IEEE Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X$_{97}$ *(IDEF$_{object}$)*

Sponsor

Software Engineering Standards Committee of the IEEE Computer Society

Approved 25 June 1998

IEEE-SA Standards Board

Abstract: IDEF1X$_{97}$ consists of two conceptual modeling languages. The key-style language supports data/information modeling and is downward compatible with the US government's 1993 standard, FIPS PUB 184. The identity-style language is based on the object model with declarative rules and constraints. IDEF1X$_{97}$ identity style includes constructs for the distinct but related components of object abstraction: interface, requests, and realization; utilizes graphics to state the interface; and defines a declarative, directly executable Rule and Constraint Language for requests and realizations. IDEF1X$_{97}$ conceptual modeling supports implementation by relational databases, extended relational databases, object databases, and object programming languages. IDEF1X$_{97}$ is formally defined in terms of first order logic. A procedure is given whereby any valid IDEF1X$_{97}$ model can be transformed into an equivalent theory in first order logic. That procedure is then applied to a meta model of IDEF1X$_{97}$ to define the valid set of IDEF1X$_{97}$ models.
Keywords: conceptual schema, data model, IDEF1X, IDEF1X$_{97}$, identity style, information model, key style, object model

---

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

Print: ISBN 0-7381-0341-1 SH94663
PDF: ISBN 0-7381-1405-7 SS94663

Introduction

[This introduction is not a part of IEEE Std 1320.2-1998, IEEE Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X$_{97}$ *(IDEF$_{object}$)*.]

Background

The need for semantic models to represent conceptual schemas was recognized by the US Air Force in the mid 1970s as a result of the Integrated Computer Aided Manufacturing (ICAM) Program. The objective of this program was to increase manufacturing productivity through the systematic application of computer technology. The ICAM program identified a need for better analysis and communication techniques for people involved in improving manufacturing productivity. As a result, the ICAM program developed a series of techniques known as the ICAM Definition (IDEF) methods, which included the following:

a) IDEF0, a technique used to produce a “function model,” which is a structured representation of the activities or processes within the environment or system.
b) IDEF1, a technique used to produce an “information model,” which represents the structure and semantics of information within the environment or system.
c) IDEF2, a technique used to produce a “dynamics model,” which represents the time-varying behavioral characteristics of the environment or system.

IDEF0 and IDEF1X (the successor to IDEF1) continue to be used extensively in various government and industry settings. IDEF2 is no longer used to any significant extent.

The initial approach to IDEF information modeling (IDEF1) was published by the ICAM program in 1981, based on current research and industry needs [B23].[1] The theoretical roots for this approach stemmed from the early work of Dr. E. F. Codd on relational theory and Dr. P. P. S. Chen on the entity-relationship model. The initial IDEF1 technique was based on the work of Dr. R. R. Brown and Mr. T. L. Ramey of Hughes Aircraft and Mr. D. S. Coleman of D. Appleton Company, with critical review and influence by Mr. C. W. Bachman, Dr. P. P. S. Chen, Dr. M. A. Melkanoff, and Dr. G. M. Nijssen.

In 1983, the US Air Force initiated the Integrated Information Support System (I^2S^2) project under the ICAM program. The objective of this project was to provide the enabling technology to integrate a network of heterogeneous computer hardware and software both logically and physically. As a result of this project and industry experience, the need for an enhanced technique for information modeling was recognized.

Application within industry had led to the development in 1982 of a Logical Database Design Technique (LDDT) by R. G. Brown of the Database Design Group. The technique was also based on the relational model of Dr. E. F. Codd and the entity-relationship model of Dr. P. P. S. Chen, with the addition of the generalization concepts of J. M. Smith and D. C. P. Smith. LDDT provided multiple levels of models and a set of graphics for representing the conceptual view of information within an enterprise. It had a high degree of overlap with IDEF1 features, included additional semantic and graphical constructs, and addressed information modeling enhancement requirements that had been identified under the I^2S^2 program. Under the technical leadership of Dr. M. E. S. Loomis of D. Appleton Company, a substantial subset of LDDT was combined with the methodology of IDEF1 and published by the ICAM program in 1985 [B15]. This technique was called IDEF1 Extended or, simply, IDEF1X.

In December 1993, the US government released a Federal Information Processing Standard (FIPS) for IDEF1X. FIPS PUB 184 [B13] was based on the ICAM program description of IDEF1X and additional features originally included in LDDT. The FIPS clarified and corrected points in the ICAM publication, sepa-

[1]The numbers in brackets correspond to those of the bibliography items listed in Annex A.

rated language syntax and semantics definition from practice and use issues, and provided a formal first-order language definition of IDEF1X.

IEEE Std 1320.2-1998 continues the evolution of the IDEF1X language. It is driven by two needs. First, development of a national standard for the language makes the definition more accessible to organizations that do not follow US government standards and allows consideration and inclusion of features needed outside the US federal government sector. Second, the needs of the users of a standard change over time as system development techniques and available technology continue to evolve. Some users adopt new concepts earlier than others. To be valuable to the widest set of users, this standard needs to support a range of practices, from those supported by the FIPS to those that are emerging as future drivers of integration.

The change in the drivers of integration is being recognized by both government and private sector organizations. Integration involves not only data but the operations performed on that data. The emerging object modeling approaches seek to treat all activities as performed by collaborating objects that encompass both the data and the operations that can be performed against that data. There is increasing interest in these approaches in both the government and private sectors. Original work done for the National Institute of Standards and Technology (NIST) in 1994 and early 1995 by Robert G. Brown of the Database Design Group (DBDG) provides the basic elements required for a graceful evolution of IDEF1X toward full coverage of object modeling [B5].

The DBDG work analyzed the 1993 definition of IDEF1X and compared to it to the emerging consensus object model. The analysis showed that

— The concepts of the current IDEF1X were a subset of those of the object model,
— The current IDEF1X contained restrictions that are unnecessary in the object model, and
— The object model contains significant new concepts.

The work also showed that if the concepts of IDEF1X were more fully developed, the restrictions dropped, and the new concepts added, the result would be an upwardly compatible object modeling technique. The evolutionary features of IDEF1X described in this standard draw heavily from the DBDG work done for NIST.

Base documents

The following documents served as base documents for the parts of IEEE Std 1320.2-1998 indicated:

a) *From IDEF1X to IDEF$_{object}$*, 1995, by Robert G. Brown, The Database Design Group, Newport Beach, CA, is the base document for the Class and Responsibility clauses. Partial financial support was provided by the National Institute of Standards and Technology (NIST) [B5].
b) *IDEF1X$_{97}$ Rule and Constraint Language (RCL)*, 1997, by Robert G. Brown, The Database Design Group, Newport Beach, CA, is the base document for the RCL clause. Partial financial support was provided by the Defense Information Systems Agency (DISA) [B6].
c) *IDEF1X$_{97}$ Formalization*, 1998, by Valdis Berzins, Naval Postgraduate School, Monterey, CA, and Robert G. Brown, The Database Design Group, Newport Beach, CA, is the base document for the Formalization clause. Partial financial support was provided by DISA and the Defense Modeling and Simulations Office (DMSO) [B7].

The IDEF1X approach

A principal objective of IDEF1X is to support integration. The “IDEF1X approach” to integration focuses on the capture, management, and use of a single semantic definition of the data resource referred to as a *conceptual schema*. The conceptual schema provides a single integrated definition of the concepts relevant to an enterprise, unbiased toward any particular application. The primary objective of this conceptual schema is to provide a consistent definition of the meanings and interrelationship of concepts. This definition can then be used to integrate, share, and manage the integrity of the concepts. A conceptual schema must have three important characteristics:

— It must be consistent with the infrastructure of the business and be true across all application areas.
— It must be extendible, such that, new concepts can be defined without disruption to previously defined concepts.
— It must be transformable to both the required user views and to a variety of implementation environments.

IDEF1X is the semantic modeling technique described by IEEE Std 1320.2-1998. The IDEF1X technique was developed to meet the following requirements:

— Support the development of conceptual schemas.
— Be a coherent language.
— Be teachable.
— Be well-tested and proven.
— Be automatable.

Organization of this document

This document begins with an explanation of the scope and purpose of this version of the IDEF1X standard. Clause 1 also describes the evolution of the IDEF1X standard. It provides a context for understanding the approach and constructs presented in the rest of this standard.

Clause 2 identifies additional references that must be on hand and available to the reader of this standard for its implementation. Other documentation and related references that might be of interest to the reader or that were used in preparing this standard are included in the bibliography (see Annex A).

This document uses words in accordance with their definitions in the *Merriam-Webster's Collegiate Dictionary* [B26]. A definitions clause (see Clause 3) is provided for the convenience of those not already familiar with the terminology in question. It also contains any terminology that has specialized meaning in the context of this standard.

Clauses 4 through 6 along with 8 discuss the meaning (semantics) of each model construct that may be used within an IDEF1X model, as well as how they shall be put together to form a valid model (the syntax). Clause 7 provides a full description of the Rule and Constraint Language (RCL) specification language for an IDEF1X model.

Clause 4 introduces the language constructs of IDEF1X. The basic constructs of an IDEF1X model are

a) Things whose knowledge or behavior is relevant to the enterprise, represented by boxes;
b) Relationships between those things, represented by lines connecting the boxes;
c) Responsibilities of those things, stated as
   1) Knowledge and behavior properties, represented by names within the boxes,
   2) Realization of those responsibilities, expressed as sentences in a declarative language, and
   3) Rules, represented as constraints over property values.

These constructs are then described in detail in Clauses 5 and 6. Clause 8 discusses how the various constructs may be put together to form a model.

Two styles of IDEF1X modeling are described in this standard. Clauses 5 through 8 present the *identity style*, which extends the conceptual schema representation capabilities of IDEF1X. Identity-style models describe the structural dimension of an object model and specify the collaborations among the objects. Identity-style models can be used in conjunction with dynamic modeling techniques such as those based on finite state machines.

Clause 9 describes the *key style*, which is backward-compatible with FIPS PUB 184 [B13]. This style may continue to be used to produce models that represent the structure and semantics of data within an enterprise, i.e., data (information) models.

In the process of producing FIPS PUB 184 [B13], the various graphical constructs of the IDEF1X language were formalized. In essence, these constructs had no more meaning than they had before, but they became more explicitly grounded than they had been. The formalization served to make obvious the fact that the graphical aspect of IDEF1X was not the language *per se* but only one external manifestation of it. Clause 10 presents the formalization of the IDEF1X language, revised to include the language features defined in IEEE Std 1320.2-1998. The formalization also provides a metamodel of IDEF1X. In addition to the metamodel diagram, the metamodel value classes and constraints are given. The reader may wish to use this model along with Annex D, which documents the built-in classes of the IDEF1X metamodel.

Additional normative and informative annexes provide convenient reference to supporting material:

— Annex A is a bibliography of relevant reference material.
— Annex B summarizes the differences and similarities between the version of IDEF1X documented in FIPS PUB 184 [B13] and this standard. The reader familiar with FIPS PUB 184 may wish to review this information before proceeding into the body of IEEE Std 1320.2-1998.
— Annex C presents a set of examples illustrating various aspects of identity-style modeling. These examples include the representation of two patterns from *Design* Patterns [B14], a business example that applies these patterns, some value class examples, and the translation of the TcCo model from FIPS PUB 184 [B13] into an initial identity-style model.
— Annex D documents the built-in classes of the IDEF1X language.

Throughout this standard, IEEE conventions for certain words are used:

— “Shall” means “required.” For example, point 5.1.2.1 a) says “A class shall be represented as a rectangle of the shape appropriate to the class.” This statement is interpreted as a mandatory requirement that a rectangle be the only acceptable way to represent a class.
— “Should” means “recommended.” For example, point 8.1.3.7 a) says “If the objective of the view is that it be internally consistent, it should be possible to demonstrate that a consistent set of instances exists.” This statement means that the presentation of a set of instances is highly recommended but not required for conformance.
— “May” means “permitted.” For example, point 5.2.3.6 c) says that “In a sample instance table, the instance identity label may be shown to the left of the row representing the instance.”
— “Can” means “is able to.” For example, 7.5.3 states that “The uniqueness conditions guarantee that a message can be resolved to at most one class responsibility.”

Reading the document

The IDEF1X$_{97}$ (IDEF$_{object}$) standard was developed to extend the practice of information modeling (IDEF1X$_{93}$) to object modeling. The readers of this standard can be broadly classified into at least at two distinct groups: management and technical. For each group, a different approach to the reading of this standard is recommended.

Management readers

This standard is written on a fairly technical plane. Managers may wish to focus on the concepts that will help them manage projects that employ this new standard. For example, modeling now will include operations on enterprise knowledge as well as rules that govern state changes. For this group of readers, Clause 1 should be read first as the key to the document. Clause 1 delimits the scope and defines the purpose of the document, providing a succinct discussion of the evolution of IDEF1X and pointing out the capabilities added in IDEF1X$_{97}$. As this clause points out, IDEF1X$_{97}$ is considered a transition language that preserves the information modeling investment, provides opportunities to simplify the data/process approach, and positions the organization to move forward.

Clause 4 should be read next. This clause provides a high level summary of the language concepts, constructs, and notation. The notation is not terribly significant to management; however, many of the concepts and constructs summarized in Clause 4 will lead a management reader to further discussions of concepts and constructs found in Clauses 5 through 8.

In the past at least two separate requirement specification languages had to be used (e.g. IDEF0, “Function Modeling” and IDEF1X$_{93}$, “Information Modeling” languages). The IDEF1X$_{97}$ identity-style language represents concepts in a more natural way by integrating data and process and by hiding implementation details that sometimes become a barrier to specifying the requirements. Hiding the implementation detail (encapsulation) simplifies the development and maintenance of databases and software.

Encapsulation is enabled by the concept that a class instance has responsibilities (see Clause 6) specified in two parts: interface and realization. By revealing only the interface specification (names, meanings, and signatures of responsibilities) to a client, IDEF1X$_{97}$ hides the complexity of realizations (the implementation detail) and their specified methods and representation properties. The realization is specified separately with the RCL so that database and software projects developed using IDEF1X$_{97}$ can focus on specifying the desired behavior and optimizing the messages requesting the services.

Managers should find the concept of modeling levels in Clause 8 of particular interest. Three technology-independent levels (survey, integration, and fully specified) and one technology-dependent level (implementation) are presented to help provide clear work product definition for management.

Clause 9 and Annex B will show management how older style information modeling (IDEF1X$_{93}$) can be supported and extended with features of the new object language.

One of the most powerful aspects of IDEF1X$_{97}$ models is that they are, with suitable automation support, directly executable to prove their correctness. Managers generally will not need to study the details, but should be aware of where to find them. The executable nature is enabled by the concepts discussed in detail in Clause 10, with a supporting RCL explained in detail in Clause 7.

Technical readers

Several groups will have primarily technical interests in the IDEF1X$_{97}$ standard.

- **Architects and Methodologists:** Readers in this group are often responsible for developing
  - The structure given to database and software components, their interrelationships, and the principles and guidelines governing their design and evolution over time (architecture) and
  - The routine procedures and practices used to produce precise, consistent, and repeatable deliverables at the end of each stage of the development process (methodology). Generally, this technical group uses modeling languages like IDEF1X$_{97}$ to develop architectures and methodologies to guide others in building consistently high quality products.

— **Data Modelers, Object Modelers, Database Designers, Software Engineers, and Other Practitioners:** Readers in this group are often responsible for defining and specifying requirements, designing and developing databases and software system solutions, and then testing and implementing those solutions as quickly and efficiently as possible. Generally, this technical group uses modeling languages like IDEF1X$_{97}$ to develop models, designs, and products to define and satisfy operational requirements with the highest quality databases and software at the lowest risk and cost of maintenance.

— **Commercial Software Vendors:** This group includes companies that build software products and tools to support the other technical groups. These software products and tools include
1) Database management systems, including relational, object-relational, and object-oriented,
2) Languages (procedural and object oriented),
3) Computer aided software engineering (CASE) tools, and
4) Data dictionary/repository systems.

Each of these technical groups will be more naturally satisfied by different reading patterns. Although Clauses 1 and 4 provide an overview, readers in these groups will be most interested in the detailed technical topical discussions in the major clauses (Clauses 5 through 10) and the annexes.

Data modelers and database designers, for example, may want to know how the new language differs from the earlier versions of IDEF1X (Clause 9 and Annex B) and, perhaps, how to begin the transition to object modeling and design. Object modelers will need to understand all features and capabilities of IDEF1X$_{97}$ identity-style modeling.

Clause 5 delineates the two types of IDEF1X$_{97}$ classes (state and value) and describes the use of generalization and relationship concepts. For the data modelers, understanding how it is possible to use value classes in place of domains will be of interest. Object technicians in all technical groups will be interested in the value class approach and in the generalization and relationship concepts: variants of these concepts exist in many currently available object modeling and design tools and in commercial software.

If a data modeler does not intend to develop object models, Clause 6 will not be of any significant interest. However, all other technical readers should carefully read Clause 6 to gain a core understanding of the object constructs and the extent of their usage by IDEF1X$_{97}$. Clause 6 introduces the concepts of responsibility, interface, realizations, requests, properties, attributes, participant properties, operations, constraints, and notes. All technical readers should study the concepts of view, view level, environment, glossary, and model presented in Clause 8.

Clause 9 is intended for data modelers who want to or must continue the practice of key-style modeling. Other technical readers will have little interest in this clause unless they support the older style practices or are planning transitions from that style of practice to object-oriented technology. In these cases, a thorough reading of Clause 9 could help with planning for changes to architectures, methodologies, and commercial software products and tools.

For all technical readers, Clause 10 and its companion Clause 7 will present the precise definition of the language. These clauses will be a key area of study for tool builders.

Participants

We note with great sadness the passing of our friend, colleague, and group Secretary, Mr. Bill Handrick of Springfield, Virginia, in 1996. Bill's contributions to the advancement of the IDEF practice and standards efforts were immense. He was an early supporter of the IDEF User Group's education efforts, contributed to the writing of the Federal Information Processing Standard for IDEF1X, and was an active member of our working group starting with the activities leading to its formation in early 1992. Grace Hopper was his sponsor in the original ACM and, when he asked whether additional sponsors were needed, replied, "I think not, sonny." We dearly miss his humor, intellect, and grounding.

Contents

IEEE Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X$_{97}$ *(IDEF$_{object}$)*

1. Overview

This standard describes the semantics and syntax of IDEF1X, a language used to represent a conceptual schema. Two styles of IDEF1X model are described. The *key style* is used to produce information models that represent the structure and semantics of data within an enterprise and is backward-compatible with the US government's Federal Information Processing Standard (FIPS) PUB 184, *Integration Definition for Information Modeling (IDEF1X)* [B13].[1] The *identity style* is used to produce object models that represent the knowledge, behavior, and rules of the concepts within an enterprise. It can be used as a growth path for key-style models. The identity style can, with suitable automation support, be used to develop a model that is an executable prototype of the target object-oriented system.

1.1 Scope

This standard defines the semantics and syntax of IDEF1X. It does so by defining the valid constructs of the language and specifying how they can be combined to form a valid model.

IDEFIX takes the approach that an enterprise manages what it knows about (its knowledge). Such knowledge consists of awareness about enterprise-pertinent actions, facts, and the relationships among them. In order to maximize the utility of this knowledge, it must be codified in a manner that makes its interpretation consistent. Without this guidance, the knowledge is either not understood at all or, worse, misused to draw unsupported or inappropriate conclusions. The guide to the interpretation and use of the enterprise knowledge has three components:

a) A grammar that dictates the kinds of actions, facts, and relationships that the enterprise is interested in recording,
b) Operations that can be performed on/with this knowledge to produce usable information, and
c) Rules about recorded knowledge that help the enterprise weed out conflicting statements and rules that govern the state changes that recorded knowledge can undergo.

[1] The numbers in brackets correspond to those of the bibliography items listed in Annex A.

For example, the sentence “The chair sings the tree” is grammatically sound in English; there is a subject, a verb, and an object in the sentence. However, the sentence itself is not useful because it states something that is nonsensical. In a natural language, rules must be established that, for instance, indicate that the subject of the sentence must be capable of taking action, if the verb is an action, and of taking the particular action specified by the verb.

Such a guide to the interpretation and use of the enterprise knowledge is, itself, captured as a set of facts. This body of facts about facts, or metaknowledge, in turn needs a guide to *its* understanding and use. This goal, in a nutshell, is the scope of IDEF1X. As part of its semantics and syntax, IDEF1X establishes just what can be said about the enterprise knowledge and what sorts of conclusions can be drawn from that meta-knowledge.

This standard does not treat methodology. A methodology is an ordered process used to produce a repeatable result. An IDEF1X methodology deals with the process of creating a model using the IDEF1X language. While critical to the practitioner, such considerations are beyond the scope of this standard. Rather, the IDEF1X constructs will be presented individually, without regard for their logical sequence of use.

1.2 Purpose

This purpose of this standard is to describe the IDEF1X language in an unambiguous manner and thereby meet two important needs. First, those who develop and use IDEF1X models need a common understanding of the modeling constructs and rules. A precise definition of the meaning of the language components allows a model developed by one individual or group to be understood by another. Second, IDEF1X users must be supported in practice by automated tools that record and validate the models. Tool developers need a precise definition of the language so that their products assist users in applying the language correctly and allow exchange of models, at the semantic level, with other tools.

The purpose of IDEF1X as a modeling technique is the same as that of all modeling techniques employed in system analysis and development efforts, that is, to plan, build, or use systems and information systems in particular, it helps to understand the meaning of the concepts involved. Modeling provides a “language” for meanings and is sometimes referred to as closing the semantic gap between the concepts of the enterprise and the capabilities of the computer systems. Figure 1 summarizes the fundamental purpose of a model: to enable accurate and useful communication among users, analysts, and developers as they all reason about the same thing.

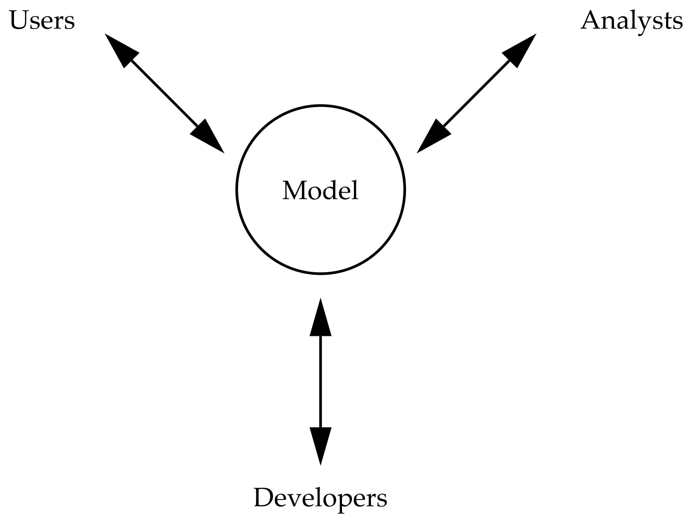


Figure 1—Communication of meanings

There are many uses for models, including process re-engineering, enterprise integration, detailed specification, implementation, and reverse engineering. Each is important.

1.3 Evolution of IDEF1X

The fundamental point of view originally adopted by IDEF1X was that the world was made up of interrelated things and that the meaning of data devolved from an understanding of these things and their relationships. This *key style* of IDEF1X modeling has been used over the past two decades to produce information models that represent the structure and semantics of data within an enterprise. The object model expands that point of view to include behavior. The evolution of IDEF1X has incorporated this goal of a broader understanding in the *identity-style* language introduced in this standard.

The transition from key-style to identity-style models involves bringing forward many earlier IDEF1X concepts, relaxing some of the restrictions, exploiting the fundamental concepts more fully, and adding important new concepts (see Figure 2). Each of the concepts used to produce an identity-style model is discussed fully in Clauses 4 through 8 of this standard. Clause 9 describes how to apply these concepts to produce a key-style model. Note that the concepts marked “unnecessary” in Figure 2 have been retained in the key-style language for backward-compatibility with existing models and for those who wish to continue producing key-style IDEF1X models.

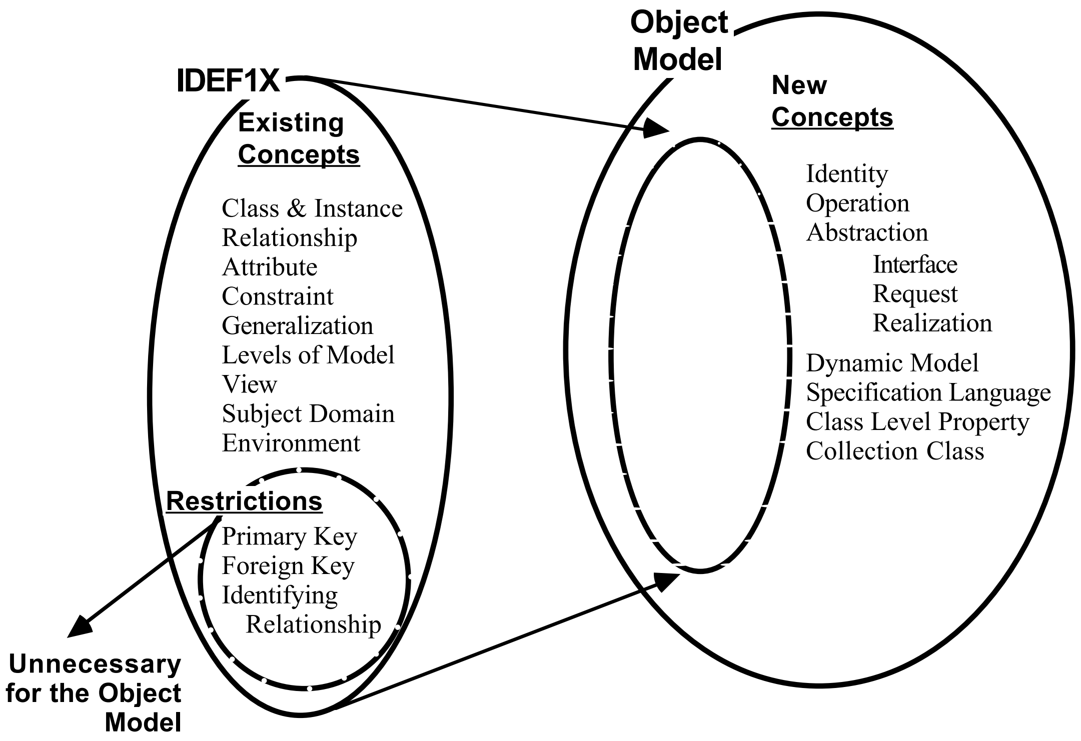


Figure 2—Correspondence of concepts

1.3.1 Understanding the data/process paradigm

The requirements for a modeling language are set largely by the way in which modelers choose to view the world. When IDEF1X was first developed in the early 1980s, the predominant system development view of the world was framed in terms of data and processes. The modeling approach of this data/process (D/P) paradigm can be summarized as follows:

a) The world is made up of activities and things.
b) Things are integrated. Activities are free-standing.
c) Activities operate on things.

Within this approach, the primary objectives of an information modeling technique are

— To provide a means for understanding and analyzing an organization's data resources,
— To provide a common means of representing and reasoning about the data,
— To provide a method for presenting an overall view of the data required to run an enterprise,
— To provide a means for defining an application-independent view of data that can be validated by users and transformed into a physical database design,
— To provide a method for deriving an integrated data definition from existing data resources.

The D/P paradigm exerted a powerful and pervasive influence over all aspects of information technology. For IDEF the result was two distinct techniques: IDEF0 for process and IDEF1X for data. Thousands of successful systems have been developed using the D/P view of the world, and many developers continue to successfully employ the techniques.

1.3.2 Understanding the emerging object-oriented paradigm

The emergence of an object-oriented (OO) view of the world has strongly influenced the evolution of IDEF1X as described in this standard. The object paradigm takes a fundamentally different view of the world. In this paradigm, the modeling approach can be summarized as

a) The world is made up of objects.
b) Objects have knowledge and behavior.
c) There are no free-standing activities. Activity is accomplished by a collaboration of objects.
d) Knowledge and behavior are different aspects of the same object, considered *together*, behind an abstraction of responsibility.

Within this approach, the primary objectives of a modeling technique are

— To provide a means for understanding and analyzing the objects that are of interest to the organization,
— To provide a common means of representing and reasoning about these objects,
— To provide a method for presenting an overall view of the objects required to run an enterprise,
— To provide a means for defining an application-independent view of objects that can be validated by users and transformed into a physical design.

1.3.3 Contrasting the paradigms

The approaches of the D/P and OO paradigms are different. Major differences are summarized below in Table 1. For IDEF1X, the concepts that emerge from the D/P and OO approaches are not entirely incompatible. Indeed, there is a high degree of correspondence in the concepts.

While an IDEF1X model has typically been called a "data model," the term has always been something of a misnomer; an IDEF1X model was never a model of "data" *per se*. The entities in an IDEF1X data model are not "data" entities. An IDEF1X entity represented a concept, or meaning, in the minds of the people of the enterprise. To emphasize their concern with meaning as opposed to representational issues, "data models" like IDEF1X models are often called "semantic data models" or "conceptual models."

Table 1—D/P and OO approaches

D/P paradigm assumptions	OO paradigm assumptions	Contrast
An entity instance is a person, place, or thing (etc.) about which the enterprise needs to keep data.	An object is a distinct thing whose knowledge (data) or behaviors (processes) are relevant.	An object combines data and process (knowledge and behavior) and hides them behind an abstraction of responsibility.
There is no free-standing data. All data is organized around the shared real-world entities of the enterprise. The data is accessed by processes and shared across applications.	There is no free-standing knowledge (data). All knowledge is organized around the shared real-world objects of the enterprise. Each object maintains its own knowledge. The knowledge is available to (modifiable by) other objects upon request, across applications.	In D/P, processes directly access and change the data of an entity. In OO, an object must be asked for its knowledge; that knowledge is not directly accessible. Only the object itself can change its knowledge. Whether the object's knowledge is by memory or derivation is known only to the object.
Processes are free-standing. Process is organized around function, accesses entities, and is unique to an application.	There are no free-standing behaviors (processes). All behavior is organized around the shared real-world objects of the enterprise. Behavior is the responsibility of the object and available to other objects upon request, across applications.	In OO, all processing is accomplished by the actions of objects. An object acts by exploiting the knowledge and behavior of itself and collaborating objects via requests. Exactly what requests are made is known only to the object.
Similar entity instances are classified into classes, and classes are related by aggregation and generalization.	Similar objects (instances) are classified into classes, and classes are related by aggregation and generalization.	Essentially the same idea, except that the object class includes behaviors.
Each entity instance in a class is distinguishable from all others by its data values.	Each object is distinct from all other objects—it has an intrinsic, immutable identity, independent of its knowledge, behaviors, or class.	The OO model can recognize as distinct what the D/P paradigm treats as indistinguishable.
There are constraints on data.	There are constraints on both knowledge and behavior.	More general kinds of constraints are needed by the object model.
Rules are incorporated by defining processes that support them.	Rules are incorporated by defining behaviors that support them.	The D/P and OO paradigms both could be improved here. It would be better if rules could be disentangled from behaviors.

An *object model* is similarly a model of meaning, but it is a richer model that is closer to the ideal of a conceptual model. An object model attempts to capture the meanings of the knowledge and behaviors of objects. Yet, even state-of-practice object models still fall short of the ideal. The objects modeled are more like clerks than executives—they do what they are told to do but are short on vision and initiative. Objects await instruction ("Chris, put the pencil down.") rather than possessing the ability to utilize their knowledge to exhibit unrequested behavior. Nevertheless, object models are, in many environments, proving to be a major advance over the combination of separate process models and data models.

1.3.4 Expanded understanding of requirements

IDEF1X continues to meet the same requirements today that it was originally chartered to meet. However, leveraging on the capabilities that the OO approach offers, the understanding of those requirements has expanded. The expanded requirements can be summarized in terms of the five points of the "IDEF1X approach":

a) *Support the development of conceptual schemas*.

The conceptual schema has been characterized as those aspects of the enterprise that are invariant across the information products of the business and implementations of the enterprise business rules (application systems and databases). Previously, this scope had been understood to include only the grammar of the data. Now the understanding of the scope of the conceptual schema can be seen to include *operations* as well as *rules*.

In addition, the scope of platforms supporting applications designed using IDEF1X has broadened. In many areas, relational database management system-based applications are slowly giving way to ones built in some form of OO environment. For IDEF1X to remain transformable into functioning systems, the OO concepts must be incorporated so that the IDEF1X language is semantically broad enough to meet the needs of its users. IDEF1X needs to provide object modeling constructs appropriate for enterprise integration—from initial survey through implementation.

b) *Be a coherent language*.

IDEF1X has a clean, coherent structure with distinct and consistent semantic concepts. Many of the IDEF1X constructs have a graphical manifestation because their semantics can be easily captured that way and the resulting diagram easily read. However, as the language has evolved, not all concepts have been forced into an iconic representation; some concepts simply cannot be easily expressed graphically in a model that remains comprehensible.

In IDEF1X, as in any language, it is important that those things that are said most often are said easily, while allowing capture of those statements that are difficult to express graphically. Some constructs are best captured in text because the semantics being represented are inherently complex. Regardless of manifestation, graphical or textual, the language as a whole remains coherent and consistent.

c) *Be teachable*.

IDEF1X data modeling has been taught and practiced for nearly two decades. The teachability of the language has always been an important consideration. IDEF1X has served well as an effective communication tool across interdisciplinary teams. This rich body of experience and familiarity will not be lost. Data models created using previous versions of IDEF1X standards will continue to be conformant under this new version in the key-style language. An upward migration path for existing IDEF1X models and skill sets is provided, and training on the newer identity-style language is expected to emerge from the marketplace.

d) *Be well-tested and proven*.

The original elements of IDEF1X were based on years of experience with predecessor techniques and have been thoroughly tested both in US government development projects and in private industry. The identity style of IDEF1X introduced in this standard has been used in a variety of industry projects. Many of the features included in this version reflect requests and suggestions from IDEF1X practitioners, while others reflect the best features of the emerging object modeling techniques.

e) *Be automatable*.

IDEF1X consists of modeling constructs that can be precisely defined. The constructs of the identity-style model provide the basis for tool support for representation and reasoning about OO conceptual models, including direct execution of the models. With the formalization of the IDEF1X language, automated reasoning about the knowledge and behavior modeled is a realistic expectation.

IEEE Std 1320.2-1998 addresses the evolutionary needs of users of earlier versions of the language. Evolution is a process of change. A new version of a “creation” emerges and becomes dominant or dies out based on its suitability to the surrounding environment. During the transition, both versions of the creation will coexist.

So, too, both versions of IDEF1X (D/P and OO) are supported by this standard. The key style of IDEF1X is fully backward-compatible with FIPS PUB 184 [B13]; the use of the identity-style features is optional. Users can migrate as needed to the expanded semantic scope characteristic of the identity-style language.

1.3.5 IDEF1X in transition

The version of IDEF1X presented in this standard is based on the object model, which is the result of the confluence of three major branches of computer science: programming, database, and artificial intelligence. As of the mid-1990s, there was no single, authoritative source for what constitutes the object model, but there was a broad consensus on the core semantic concepts. Additional semantic concepts remain in flux, and there is little consensus on syntax and methodology.

The version of IDEF1X described in FIPS PUB 184 [B13] continues to be supported by this standard and is referred to here as IDEF1X$_{93}$. Where necessary to distinguish it from this earlier version, the extended IDEF1X defined in this standard (including both identity style and key style) is referred to as IDEF1X$_{97}$.

The constructs of IDEF1X$_{97}$ were developed by

a) Framing them in terms of organizing concepts congruent with the way people think,
b) Formalizing those concepts by assigning to each a mathematical construct such that formal operations on the constructs parallel correct reasoning about the concepts,
c) Specifying a notation (diagrams or language) that actively supports representation, communication, and reasoning in terms of the concepts.

The similarities between IDEF1X$_{93}$ and IDEF1X$_{97}$ are fundamental. For both, the world consists of distinct, individual things that exist in classes[2] and are related to one another.

IDEF1X$_{97}$ was developed by relaxing some of the restrictions in IDEF1X$_{93}$, exploiting the fundamental concepts more fully, and adding some important new concepts. Each of the semantic concepts of IDEF1X$_{93}$ has a corresponding identity-style IDEF1X$_{97}$ concept, but some of the IDEF1X$_{93}$ restrictions are not needed in identity-style IDEF1X$_{97}$. These restrictions are not basically in conflict with identity-style IDEF1X$_{97}$—they could be stated if there were any reason to do so. The goals and concepts of IDEF1X$_{93}$ are subsumed by IDEF1X$_{97}$; the essential semantic constructs of IDEF1X$_{93}$ are part of IDEF1X$_{97}$. Identity-style IDEF1X$_{97}$ includes concepts that are not present in IDEF1X$_{93}$.

The identity-style IDEF1X$_{97}$ concepts are object model concepts. IDEF1X$_{97}$ includes constructs for the distinct but related components of object abstraction—interface, request, and realization. Some of the specific concepts of IDEF1X$_{97}$ that support abstraction are the principle of substitutability, declarative constraints, and declarative specifications of properties.

The identity-style IDEF1X$_{97}$ constructs model objects over varying scopes and levels of refinement. IDEF1X$_{97}$ uses both graphics and a textual specification language. Its constructs are integrated with one another by a consistent, declarative approach to object semantics.

1.3.6 Future direction

The scope of this version of the IDEF1X language covers semantic data and object modeling. Use of this standard permits the construction of data and object models that may serve to support the management of concepts as a resource, the integration of information systems, and the building of computer databases and systems.

[2]Where some say “class” and “class instance” (or, “object”), this standard adopts the terminology “class” and “instance.”

1.3.6.1 Topics for future extensions

Aspects of the object model that are topics for future extensions of IDEF1X include the following:

a) *Dynamic models.* This version of IDEF1X covers the specification of both the interface and realization of active properties (operations) of a class. However, this version of this standard does not provide a set of graphics describing individual requests or patterns of requests.
b) *Transaction models.* There are many transaction models, and this version of IDEF1X has chosen not to select one but rather provide only the most basic notions of stating a constraint and providing a way to check it. Future versions of this standard may expand on the treatment of "transaction."
c) *Exception handling.* The specification of exception handling is an important aspect of many object languages. Future versions of this standard may incorporate exception handling into the language.

1.3.6.2 Features for expanded scope

In addition, the scope of the language may be expanded to include coverage for features frequently requested by IDEF1X users. Typical examples include

a) *Rules beyond constraints.* "Rule" is a more general, and more powerful, idea than constraint. This version of the standard deals only with constraints. A future version could incorporate a fuller treatment of "rules."
b) *Technology-dependent levels/default transformations.* An important characteristic of the original IDEF1X was the existence of a default transformation from a fully attributed model to an implementation in a database system such as IMS™, IDMS™, xBase, or relational.[3] In addition to database and object database transforms, the expanded coverage of IDEF1X suggests transforms into popular object languages such as Smalltalk™, C++, and Java™. From the overall management and development point of view, providing transforms into technology-specific models encouraged building models that are actually used. It created a very useful "practical" counterbalance to "wishful modeling." Enterprise integration does not come about because of modeling *per se*—the models have to be used. The existence of default transformations encourages use.

1.3.6.3 Constructs for future versions

Specific constructs to be incorporated into future versions include the following:

a) *Importing concepts.* Allow importing a concept defined in one environment into another environment.
b) *Importing types.* Allow importing a type defined in one view into another view.
c) *Initial values.* Allow the specification of initial values for instance-level and class-level attributes.
d) *Interfaces.* A *class* consists of an interface, which is a set of responsibilities, and a realization for each of those responsibilities. An *interface* consists of just a set of public responsibilities and, if specified independently, can be realized by many classes. A *type* is either an interface or a class. Add support for interfaces and types as distinct from classes.
e) *Ordered relationships.* Support the specification of the ordering of instances participating in relationships.
f) *References.* Support one-way mappings to state classes in a way that is symmetric with attribute and relationship mappings.
g) *Visibility.* Support the specification of the visibility of types and their responsibilities outside their defining view.

[3]All trade or product names are either trademarks or registered trademarks of their respective companies and are the property of their respective holders. The mention of a product in this document is for the convenience of users of this standard and does not constitute an endorsement by the IEEE of these products.

1.4 Conformance

This document is structured to permit its use in checking a model or modeling tool for conformance to this standard.

1.4.1 Identity-style model conformance

An identity-style model is conforming when

a) The lexical rules conform to Clause 4,
b) The class (state and value), generalization, and relationship semantics, syntax and rules conform to Clause 5,
c) The class (state and value), generalization, and relationship semantics conform to the semantics defined in Clause 10,
d) The responsibility semantics, syntax, rules, requests, and realizations conform to Clause 6,
e) The responsibility and realization semantics conform to the semantics defined in Clause 10,
f) The RCL conforms to the language syntax in Clause 7,
g) The RCL semantics conform to the semantics defined in Clause 10,
h) The model infrastructure constructs conform to Clause 8, and
i) The model instantiates the language metamodel in Clause 10.

1.4.2 Identity-style modeling tool conformance

An identity-style modeling tool is conforming when

a) The lexical rules conform to Clause 4,
b) The class (state and value), generalization, and relationship semantics, syntax and rules conform to Clause 5,
c) The class (state and value), generalization, and relationship semantics conform to the semantics defined in Clause 10,
d) The responsibility semantics, syntax, rules, requests, and realizations conform to Clause 6,
e) The responsibility and realization semantics conform to the semantics defined in Clause 10,
f) The RCL semantics conform to the semantics defined in Clause 10,
g) The model infrastructure constructs conform to Clause 8,
h) It can be demonstrated that the tool's metamodel maps to the language metamodel in Clause 10, that is,
    1) There is an *onto* mapping `lang` from the set of valid populations of the tool's metamodel to the set of valid populations of the language metamodel in Clause 10,
    2) There is a total mapping `tool` from the set of valid populations of the language metamodel in Clause 10 to the set of valid populations of the tool's metamodel, and
    3) For every valid population `L` of the language metamodel in Clause 10, `L = lang(tool(L))`,
i) It can be demonstrated that the tool correctly interprets RCL as specified in Clauses 7 and 10, and
j) Any tool extensions to the graphics or RCL can be demonstrated to be reducible to the graphics or RCL specified in this standard.

1.4.3 Key-style model conformance

A key-style model is conforming when

a) The lexical rules conform to Clause 4, and
b) The model components, semantics, syntax, and rules conform to Clause 8.

1.4.4 Key-style modeling tool conformance

A key-style modeling tool is conforming when

a) The lexical rules conform to Clause 4, and
b) The model components, semantics, syntax, and rules conform to Clause 8.

2. References

This standard should be used in conjunction with the following publication. When the cited standard is superseded by an approved revision, the revision shall apply.

IEEE Std 100-1996, IEEE Standard Dictionary of Electrical and Electronics Terms.[4]

3. Definitions, acronyms, and abbreviations

3.1 Definitions

Throughout this standard, English words are used in accordance with their definitions in the latest edition of *Webster's New Collegiate Dictionary* [B26]. Technical terms not defined in *Webster's New Collegiate Dictionary* are used in accordance with their definitions in IEEE Std 100-1996. Where a definition in IEEE Std 100-1996 does not reflect usage specific to this document, or if a term used is not defined in IEEE Std 100-1996, then an appropriate definition is provided in this clause. In some cases, a term defined in IEEE Std 100-1996 is restated in this clause where it is felt that doing so enhances the usefulness of this document. Where a term applies only to the key style of modeling, it has been annotated as such.

3.1.1 abstract class: A class that cannot be instantiated independently, i.e., instantiation must be accomplished via a subclass. A class for which every instance must also be an instance of a subclass in the cluster (i.e., a total cluster) is called an abstract class with respect to that cluster.

3.1.2 abstract data type: A data type for which the user can create instances and operate on those instances, but the range of valid operations available to the user does not depend in any way on the internal representation of the instances or the way in which the operations are realized. The data is “abstract” in the sense that values in the extent, i.e., the concrete values that represent the instances, are any set of values that support the operations and are irrelevant to the user. An abstract data type defines the operations on the data as part of the definition of the data and separates what can be done (interface) from how it is done (realization).

3.1.3 aggregate responsibility: A broadly stated responsibility that is eventually refined as specific properties and constraints.

3.1.4 alias: An alternate name for an IDEF1X model construct (class, responsibility, entity, or domain).

3.1.5 alternate key: Any candidate key of an entity other than the primary key. [key style]

3.1.6 ancestor (of a class): A generic ancestor of the class or a parent of the class or an ancestor of a parent of the class. *Contrast:* **generic ancestor; reflexive ancestor**.

3.1.7 associative class: A class introduced to resolve a many-to-many relationship.

[4] IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331 USA (http://standards.ieee.org/).

3.1.8 associative literal: A literal that denotes an instance in terms of its value. The form of expression used to state an associative literal is `className with propertyName: propertyValue`.

3.1.9 attribute: (A) A kind of property associated with a set of real or abstract things (people, objects, places, events, ideas, combinations of things, etc.) that is some characteristic of interest. An attribute expresses some characteristic that is generally common to the instances of a class. **(B)** An attribute is a function from the instances of a class to the instances of the value class of the attribute. **(C)** The name of the attribute is the name of the role that the value class plays in describing the class, which may simply be the name of the value class (as long as using the value class name does not cause ambiguity).

3.1.10 attribute name: A role name for the value class of the attribute.

3.1.11 bag: A kind of collection class whose members are unordered but in which duplicates are meaningful. *Contrast:* **list**; **set**.

3.1.12 behavior: The aspect of an instance's specification that is determined by the state-changing operations it can perform.

3.1.13 built-in class: A class that is a primitive in the IDEF1X metamodel.

3.1.14 candidate key: An attribute, or combination of attributes, of an entity for which no two instances agree on the values. [key style]

3.1.15 cardinality: A specification of how many instances of a first class may or must exist for each instance of a second (not necessarily distinct) class, and how many instances of a second class may or must exist for each instance of a first class. For each direction of a relationship, the cardinality can be constrained. *See also:* **cardinality constraint**.

3.1.16 cardinality constraint: (A) A kind of constraint that limits the number of instances that can be associated with each other in a relationship. *See also:* **cardinality**. **(B)** A kind of constraint that limits the number of members in a collection. *See also:* **collection cardinality**.

3.1.17 cast: To treat an object of one type as an object of another type. *Contrast:* **coerce**.

3.1.18 categorization: *See:* **generalization**. [key style]

3.1.19 category cluster: *See:* **subclass cluster**. [key style]

3.1.20 category discriminator: *See:* **discriminator**. [key style]

3.1.21 category entity: An entity whose instances represent a subtype or subclassification of another entity (generic entity). *Syn:* subclass; subtype. [key style]

3.1.22 child entity: The entity in a specific relationship whose instances can be related to zero or one instance of the other entity (parent entity). [key style]

3.1.23 class: An abstraction of the knowledge and behavior of a set of similar things. Classes are used to represent the notion of "things whose knowledge or actions are relevant."

3.1.24 class-level attribute: A mapping from the class itself to the instances of a value class.

3.1.25 class-level operation: A mapping from the (cross product of the) class itself and the instances of the input argument types to the (cross product of the) instances of the other (output) argument types.

3.1.26 class-level responsibility: A kind of responsibility that represents some aspect of the knowledge, behavior, or rules of the class as a whole. For example, the total `registeredVoterCount` would be a class-level property of the class `registeredVoter`; there would be only one value of `registered-VoterCount` for the class as a whole. *Contrast:* **instance-level responsibility**.

3.1.27 cluster: *See:* **subclass cluster**.

3.1.28 coerce: To treat an object of one type as an object of another type by using a different object. *Contrast:* **cast**.

3.1.29 collaboration: The cooperative exchange of requests among classes and instances in order to achieve some goal.

3.1.30 collection cardinality: A specification, for a collection-valued property, of how many members the value of the property, i.e., the collection, may or must have for each instance. *See also:* **cardinality constraint**.

3.1.31 collection class: A kind of class in which each instance is a group of instances of other classes.

3.1.32 collection property: *See:* **collection-valued property**.

3.1.33 collection-valued: A value that is complex, i.e., having constituent parts. *Contrast:* **scalar**.

3.1.34 collection-valued class: A class in which each instance is a collection of values. *Contrast:* **scalar-valued class**.

3.1.35 collection-valued property: A property that maps to a collection class. *Contrast:* **scalar-valued property**.

3.1.36 common ancestor constraint: A kind of constraint that involves two or more relationship paths to the same ancestor class and states either that a descendent instance must be related to the *same* ancestor instance through each path or that it must be related to a *different* ancestor instance through each path.

3.1.37 complete cluster: *See:* **total cluster**. *Contrast:* **incomplete cluster**.

3.1.38 composite key: A key comprising of two or more attributes. [key style]

3.1.39 conceptual model: A model of the concepts relevant to some endeavor.

3.1.40 constant: (A) (As a noun) An instance whose identity is known at the time of writing. The identity of a constant state class instance is represented by #K, where K is an integer or a name. **(B)** (As an adjective) The specification that an attribute or participant property value, once assigned, may not be changed, or that an operation shall always provide the same output argument values given the same input argument values.

3.1.41 constraint: (A) A kind of **responsibility** that is a statement of facts that are required to be true in order for the constraint to be met. Classes have constraints, expressed in the form of logical sentences about property values. An instance conforms to the constraint if the logical sentence is true. Some constraints are inherent in the modeling constructs; other constraints are specific to a particular model and are stated in the specification language. **(B)** A rule that specifies a valid condition of data. [key style]

3.1.42 contravariance: A rule governing the overriding of a property and requiring that the set of values acceptable for an input argument in the overriding property shall be a superset (includes the same set) of the set of values acceptable for that input argument in the overridden property, and the set of values acceptable for an output argument in the overriding property shall be a subset (includes the same set) of the set of values acceptable for that output argument in the overridden property.

3.1.43 current extent: *See:* **extensional set**.

3.1.44 data model: A graphical and textual representation of analysis that identifies the data needed by an organization to achieve its mission, functions, goals, objectives, and strategies and to manage and rate the organization. A data model identifies the entities, domains (attributes), and relationships (associations) with other data and provides the conceptual view of the data and the relationships among data. [key style]

3.1.45 data type: (A) A categorization of an abstract set of possible values, characteristics, and set of operations for an attribute. Integers, real numbers, and character strings are examples of data types. [key style] **(B)** A set of values and operations on those values. The set of values is called the *extent* of the type. Each member of the set is called an instance of the type.

3.1.46 dependent entity: An entity for which the unique identification of an instance depends upon its relationship to another entity. Expressed in terms of the foreign key, an entity is said to be dependent if any foreign key is wholly contained in its primary key. *Syn:* identifier-dependent entity. *Contrast:* **independent entity**. [key style]

3.1.47 dependent state class: A class whose instances are, by their very nature, intrinsically related to certain other state class instance(s). It would not be appropriate to have a dependent state class instance by itself and unrelated to an instance of another class(es) and, furthermore, it makes no sense to change the instance(s) to which it relates. *Contrast:* **independent state class**.

3.1.48 derived attribute: *See:* **derived property**.

3.1.49 derived participant property: *See:* **derived property**; **participant property**.

3.1.50 derived property: The designation given to a property whose value is determined by computation. The typical case of a derived property is as a derived attribute although there is nothing to prohibit other kinds of derived property.

3.1.51 discriminator: (A) A property of a superclass, associated with a cluster of that superclass, whose value identifies to which subclass a specific instance belongs. Since the value of the discriminator (when a discriminator has been declared) is equivalent to the identity of the subclass to which the instance belongs, there is no requirement for a discriminator in identity-style modeling. **(B)** An attribute in the generic entity (or a generic ancestor entity) of a category cluster whose values indicate which category entity in the category cluster contains a specific instance of the generic entity. All instances of the generic entity with the same discriminator value are instances of the same category entity. [key style]

3.1.52 domain: *Syn:* value class.

3.1.53 dynamic model: A kind of model that describes individual requests or patterns of requests among objects. *Contrast:* **static model**.

3.1.54 encapsulation: The concept that access to the names, meanings, and values of the responsibilities of a class is entirely separated from access to their realization.

3.1.55 entity: (A) The representation of a concept, or meaning, in the minds of the people of the enterprise. **(B)** The representation of a set of real or abstract things (people, objects, places, events, ideas, combination of things, etc.) that are recognized as the same type because they share the same characteristics and can participate in the same relationships. [key style]

3.1.56 entity instance: One of a set of real or abstract things represented by an entity. Each instance of an entity can be specifically identified by the value of the attribute(s) participating in its primary key. [key style]

3.1.57 environment: A concept space, i.e., an area in which a concept has an agreed-to meaning and one or more agreed-to names that are used for the concept.

3.1.58 environment glossary: *See:* **glossary**.

3.1.59 existence constraint: A kind of constraint stating that an instance of one entity cannot exist unless an instance of another related entity also exists. [key style]

3.1.60 existence dependency: A kind of constraint between two related entities indicating that no instance of one can exist without being related to an instance of the other. The following association types represent existence dependencies: identifying relationships, categorization structures and mandatory nonidentifying relationships. [key style]

3.1.61 extensional set: The set containing the currently existing instances of a class. The instances in the extensional set correspond to the database and data modeling notion of *instance*. *Syn:* current extent.

3.1.62 foreign key: An attribute, or combination of attributes, of a child or category entity instance whose values match those in the primary key of a related parent or generic entity instance. A foreign key results from the migration of the parent or generic entity's primary key through a generalization structure or a relationship. [key style]

3.1.63 formalization: The precise description of the semantics of a language in terms of a formal language such as first order logic.

3.1.64 framework: A reusable design (models and/or code) that can be refined (specialized) and extended to provide some portion of the overall functionality of many applications.

3.1.65 function: A single-valued mapping. The mapping `M` from `D` to `R` is a *function* if for any `X` in `D` and `Y` in `R`, there is at most one pair `[ X, Y ]` in `M`. *Syn:* single-valued. *Contrast:* **multi-valued**.

3.1.66 generalization: (A) Saying that a subclass `S` generalizes to a superclass `C` means that every instance of class `S` is also an instance of class `C`. Generalization is fundamentally different from a *relationship*, which may associate distinct instances. **(B)** A taxonomy in which instances of both entities represent the same real or abstract thing. One entity (the generic entity) represents the complete set of things and the other (category entity) represents a subtype or sub-classification of those things. The category entity may have one or more attributes, or relationships with instances of another entity, not shared by all generic entity instances. Each instance of the category entity is simultaneously an instance of the generic entity. [key style]

3.1.67 generalization hierarchy: *See:* **generalization taxonomy**.

3.1.68 generalization network: *See:* **generalization taxonomy**.

3.1.69 generalization structure: A connection between a superclass and one of its more specific, immediate subclasses.

3.1.70 generalization taxonomy: A set of generalization structures with a common generic ancestor. In a generalization taxonomy every instance is fully described by one or more of the classes in the taxonomy. The structuring of classes as a generalization taxonomy determines the inheritance of responsibilities among classes.

3.1.71 generic ancestor (of a class): A superclass that is either an immediate superclass of the class or a generic ancestor of one of the superclasses of the class. *Contrast:* **ancestor**. *See also:* **reflexive ancestor**.

3.1.72 generic entity: An entity whose instances are classified into one or more subtypes or subclassifications (category entities). *Syn:* superclass; supertype. [key style]

3.1.73 glossary: The collection of the names and narrative descriptions of all terms that may be used for defined concepts (views, classes, subject domains, relationships, responsibilities, properties, and constraints) within an environment.

3.1.74 hidden: A general term covering both private and protected. *Contrast:* **public**. *See also:* **private**; **protected**.

3.1.75 IDEF1X model: A set of one or more IDEF1X views, often represented as view diagrams that depict the underlying semantics of the views, along with definitions of the concepts used in the views.

3.1.76 identifier dependency: A kind of constraint between two related entities requiring the primary key in one (child entity) to contain the entire primary key of the other (parent entity). Identifying relationships and categorization structures represent identifier dependencies. [key style]

3.1.77 identifier-dependent entity: *Syn:* dependent entity.

3.1.78 identifier-independent entity: *Syn:* independent entity.

3.1.79 identifying relationship: A kind of specific (not many-to-many) relationship in which every attribute in the primary key of the parent entity is contained in the primary key of the child entity. *Contrast:* **nonidentifying relationship**. [key style]

3.1.80 identity: The inherent property of an instance that distinguishes it from all other instances. Identity is intrinsic to the instance and independent of the instance's property values or the classes to which the instance belongs.

3.1.81 identity-style view: A view produced using the identity-style modeling constructs.

3.1.82 immutable class: A class for which the set of instances is fixed; its instances do not come and go over time. *Contrast:* **mutable class**. *See also:* **value class**.

3.1.83 incomplete cluster: *See:* **partial cluster**. *Contrast:* **complete cluster**.

3.1.84 independent entity: An entity for which each instance can be uniquely identified without determining its relationship to another entity. *Syn:* identifier-independent entity. *Contrast:* **dependent entity**. [key style]

3.1.85 independent state class: A state class that is not a dependent state class. *Contrast:* **dependent state class**.

3.1.86 inheritance: A semantic notion by which the responsibilities (properties and constraints) of a subclass are considered to include the responsibilities of a superclass, in addition to its own, specifically declared responsibilities.

3.1.87 inherited attribute: (A) An attribute that is a characteristic of a class by virtue of being an attribute of a generic ancestor. **(B)** An attribute that is a characteristic of a category entity by virtue of being an attribute in its generic entity or a generic ancestor entity. [key style]

3.1.88 input argument: The designation given to an operation argument that will always have a value at the invocation of the operation. *Contrast:* **output argument**.

3.1.89 instance: A discrete, bounded thing with an intrinsic, immutable, and unique identity. Anything that is classified into a class is said to be an instance of the class. All the instances of a given class have the same responsibilities, i.e., they possess the same kinds of knowledge, exhibit the same kinds of behavior, participate in the same kinds of relationships, and obey the same rules. Unless otherwise noted, *instance* means an existing instance, that is, a member of the current extent.

3.1.90 instance-level attribute: A mapping from the instances of a class to the instances of a value class.

3.1.91 instance-level operation: A mapping from the (cross product of the) instances of the class and the instances of the input argument types to the (cross product of the) instances of the other (output) argument types.

3.1.92 instance-level responsibility: A kind of responsibility that applies to each instance of the class individually. *Contrast:* **class-level responsibility**.

3.1.93 interface: The declaration of the meaning and the signature for a property or constraint. The interface states “what” a property (responsibility) knows or does or what a constraint (responsibility) must adhere to. The interface specification consists of the meaning (semantics) and the signature (syntax) of a property or constraint.

3.1.94 intrinsic: The specification that a property is total (i.e., mandatory), single-valued, and constant.

3.1.95 intrinsic relationship: A kind of relationship that is total, single-valued, and constant from the perspective of (at least) one of the participating classes, referred to as a *dependent class*. Such a relationship is considered to be an integral part of the essence of the dependent class. For example, a transaction has an intrinsic relationship to its related account because it makes no sense for an instance of a transaction to “switch” to a different account since that would change the very nature of the transaction. *Contrast:* **nonintrinsic relationship**.

3.1.96 key migration: The modeling process of placing the primary key of a parent or generic entity in its child or category entity as a foreign key. [key style]

3.1.97 key-style view: A view that represents the structure and semantics of data within an enterprise, i.e., data (information) models. The key-style view is backward-compatible with FIPS PUB 184 [B13].

3.1.98 knowledge: The aspect of an instance’s specification that is determined by the values of its attributes, participant properties, and constant, read-only operations.

3.1.99 label: A word or phrase that is attached to or part of a model graphic. A label typically consists of a model construct’s name (or one of the aliases) and may contain additional textual annotations (such as a note identifier).

3.1.100 level: A designation of the coverage and detail of a view. There are multiple levels of view; each is intended to be distinct, specified in terms of the modeling constructs to be used.

3.1.101 list: A kind of collection class that contains no duplicates and whose members are ordered. *Contrast:* **bag**; **set**.

3.1.102 literal: The denotation of a specific instance of a value class.

3.1.103 lowclass: If an instance is in a class `S` and not in any subclass of `S`, then `S` is the lowclass for the instance.

3.1.104 mandatory: A syntax keyword used to specify a total mapping. *Contrast:* **optional**. *See also:* **total**.

3.1.105 mandatory nonidentifying relationship: A kind of nonidentifying relationship in which an instance of the child entity must be related to an instance of the parent entity. *Contrast:* **optional nonidentifying relationship**. *See also:* **nonidentifying relationship**. [key style]

3.1.106 many-to-many relationship: A kind of relationship between two state classes (not necessarily distinct) in which each instance of one class may be associated with any number of instances of a second class (possibly none), and each instance of the second class may be related to any number of instances of the first class (possibly none).

3.1.107 mapping: An assigned correspondence between two things that is represented as a set of ordered pairs. Specifically, a mapping from a class to a value class is an attribute. A mapping from a state class to a state class is a participant property. A mapping from the (cross product of the) instances of the class and the instances of the input argument types to the (cross product of the) instances of the other (output) argument types is an operation.

3.1.108 mapping completeness: A designation of whether a mapping is complete (totally mapped) or incomplete (partial). *See also:* **partial**; **total**.

3.1.109 meaning: (of a responsibility) A statement of what the responsibility means. The statement of responsibility is written from the point of view of the requester, not the implementer. The statement of responsibility states what the requester needs to know to make intelligent use of the property or constraint. That statement should be complete enough to let a requester decide whether to make the request, but it should stop short of explaining how a behavior or value is accomplished or derived. Meaning is initially captured using freeform natural language text in a glossary definition. It may be more formally refined into a statement of *pre-conditions* and *post-conditions* using the *specification language*.

3.1.110 message: A communication sent from one object to another. *Message* encompasses requests to meet responsibilities as well as simple informative communications. *See also:* **request**.

3.1.111 metamodel: A metamodel `Vm` for a subset of IDEF$_{object}$ is a view of the constructs in the subset that is expressed using those constructs such that there exists a valid instance of `Vm` that is a description of `Vm` itself.

3.1.112 method: A statement of how property values are combined to yield a result.

3.1.113 migrated attribute: A foreign key attribute of a child entity. [key style]

3.1.114 migrated key: *Syn:* foreign key. [key style]

3.1.115 model: (A) A representation of something that suppresses certain aspects of the modeled subject. This suppression is done in order to make the model easier to deal with and more economical to manipulate and to focus attention on aspects of the modeled subject that are important for the intended purpose of the model. For instance, an accurate model of the solar system could be used to predict when planetary conjunctions will take place and the phases of the moon at a particular time. Such a model would generally not attempt to represent the internal workings of the sun or the surface composition of each planet. **(B)** An interpretation of a theory for which all the axioms of the theory are true. [logic sense]

3.1.116 model glossary: The collection of the names and definitions of all defined concepts that appear within the views of a model.

3.1.117 multi-valued: A mapping that is not a function. *Contrast:* **function**; **single-valued**.

3.1.118 multi-valued property: A property with a multi-valued mapping. *Contrast:* **single-valued property**.

3.1.119 multiple inheritance: The ability of a subclass to inherit responsibilities from more than one superclass.

3.1.120 mutable class: A class for which the set of instances is not fixed; its instances come and go over time. *Contrast:* **immutable class**. *See also:* **state class**.

3.1.121 name: A word or phrase that designates some model construct (such as a class, responsibility, subject domain, etc.).

3.1.122 named constraint: A constraint that is specific to a particular model, rather than being inherent in some modeling construct (such as a cardinality constraint.). A named constraint is explicitly named, its meaning is stated in natural language, and its realization is written in the specification language.

3.1.123 nonidentifying relationship: A kind of specific (not many-to-many) relationship in which some or all of the attributes contained in the primary key of the parent entity do not participate in the primary key of the child entity. *Contrast:* **identifying relationship**. *See also:* **mandatory nonidentifying relationship**, **optional nonidentifying relationship**. [key style]

3.1.124 nonintrinsic relationship: A kind of relationship that is partial, is multi-valued, or may change. *Contrast:* **intrinsic relationship**.

3.1.125 nonkey attribute: An attribute that is not the primary or a part of a composite primary key of an entity. [key style]

3.1.126 note: A body of free text that describes some general comment or specific constraint about a portion of a model. A note may be used in an early, high-level view prior to capturing constraints in the specification language; a note may further clarify a rule by providing explanations and examples. A note may also be used for "general interest" comments not involving rules. These notes may accompany the model graphics.

3.1.127 object: *Syn:* instance.

3.1.128 object identifier: Some concrete representation for the identity of an object (instance). The object identifier (oid) is used to show examples of instances with identity, to formalize the notion of identity, and to support the notion in programming languages or database systems.

3.1.129 object model: An integrated abstraction that treats all activities as performed by collaborating objects and encompassing both the data and the operations that can be performed against that data. An object model captures both the meanings of the knowledge and actions of objects behind the abstraction of responsibility.

3.1.130 oid: *See:* **object identifier**.

3.1.131 one-to-many relationship: A kind of relationship between two state classes in which each instance of one class, referred to as the *child* class, is specifically constrained to relate to no more than one instance of a second class, referred to as the *parent* class.

3.1.132 operation: A kind of property that is a mapping from the (cross product of the) instances of the class and the input argument types to the (cross product of the) instances of the other (output) argument types. The operations of a class specify the behavior of its instances. While an attribute or participant property is an abstraction of what an instance knows, an operation is an abstraction of what an instance does. Operations can perform input and output, and can change attribute and participant property values. Every operation is associated with one class and is thought of as a responsibility of that class. No operations are the joint responsibility of multiple classes.

3.1.133 optional: A syntax keyword used to specify a partial mapping. *Contrast:* **mandatory**. *See also:* **partial**.

3.1.134 optional attribute: An attribute that may have no value for an instance.

3.1.135 optional nonidentifying relationship: A kind of nonidentifying relationship in which an instance of the child entity can exist without being related to an instance of the parent entity. *Contrast:* **mandatory nonidentifying relationship**. *See also:* **nonidentifying relationship**. [key style]

3.1.136 output argument: An argument that has not been specified as an input argument. It is possible for an output argument to have no value at the time a request is made. *Contrast:* **input argument**.

3.1.137 override: The ability of a property in a subclass to respecify the realization of an inherited property of the same name while retaining the same meaning.

3.1.138 overriding property: A property in a subclass that has the same meaning and signature as a similarly named property in one of its superclasses, but has a different realization.

3.1.139 owned attribute: An attribute of an entity that has not migrated into the entity. [key style]

3.1.140 parallel classes: A pair of classes that are distinct, are not mutually exclusive and have a common generic ancestor class and for which neither is a generic ancestor of the other.

3.1.141 parameterized collection class: A kind of collection class restricted to hold only instances of a specified type (class).

3.1.142 parent entity: An entity in a specific relationship whose instances can be related to a number of instances of another entity (child entity). [key style]

3.1.143 partial: An incomplete mapping, i.e., some instances map to no related instance. An attribute may be declared partial, meaning it may have no value. A participant property is declared optional as part of the relationship syntax. An operation is declared partial when it may have no meaning for some instances, i.e., it may not give an answer or produce a response. *Contrast:* **total**. *See also:* **mapping completeness**; **optional**.

3.1.144 partial cluster: A subclass cluster in which an instance of the superclass may exist without also being an instance of any of the subclasses. *Contrast:* **total cluster**. *See also:* **superclass**.

3.1.145 participant property: A kind of property of a state class that reflects that class' knowledge of a relationship in which instances of the class participate. When a relationship exists between two state classes, each class contains a participant property for that relationship. A participant property is a mapping from a state class to a related (not necessarily distinct) state class. The name of each participant property is the name of the role that the other class plays in the relationship, or it may simply be the name of the class at the other end of the relationship (as long as using the class name does not cause ambiguity). A value of a participant property is the identity of a related instance.

3.1.146 path assertion: *See:* **common ancestor constraint**.

3.1.147 post-condition: A condition that is guaranteed to be true after a successful property request.

3.1.148 pre-condition: A condition that is required to be true before making a property request.

3.1.149 primary key: The candidate key selected as the unique identifier of an entity. [key style]

3.1.150 private: A responsibility that is visible only to the class or the receiving instance of the class (available only within methods of the class). *Contrast:* **protected**; **public**. *See also:* **hidden**.

3.1.151 property: A kind of responsibility that is an inherent or distinctive characteristic or trait that manifests some aspect of an object's knowledge or behavior. Three kinds of property are defined: attributes, participant properties due to relationships, and operations.

3.1.152 protected: A responsibility that is visible only to the class or the receiving instance of the class (available only within methods of the class or its subclasses). *Contrast:* **private**; **public**. *See also:* **hidden**.

3.1.153 public: A responsibility that is not hidden, i.e., visible to any requester (available to all without restriction). *Contrast:* **hidden**; **private**; **protected**.

3.1.154 RCL: *See:* **Rule and Constraint Language**.

3.1.155 read-only: A property that causes no state changes, i.e., it does no updates.

3.1.156 realization: The representation of interface responsibilities through specified algorithms and any needed representation properties. The realization states "how" a responsibility is met; it is the statement of the responsibility's method. Realization consists of any necessary representation properties together with the algorithm (if any). A realization may involve representation properties or an algorithm, or both. For example, an attribute typically has only a representation and no algorithm. An algorithm that is a "pure algorithm" (i.e., without any representation properties) uses only literals; it does not "get" any values as its inputs. Finally, a derived attribute or operation typically has both an algorithm and representation properties.

3.1.157 referential integrity: (A) A guarantee that a reference refers to an object that exists. **(B)** A guarantee that all specified conditions for a relationship hold true. For example, if a class is declared to require at least one instance of a related state class, it would be invalid to allow an instance that does not have such a relationship.

3.1.158 reflexive ancestor (of a class): The class itself or any of its generic ancestors. *See also:* **generic ancestor**. *Contrast:* **ancestor**.

3.1.159 relationship: A kind of association between two (not necessarily distinct) classes that is deemed relevant within a particular scope and purpose. The association is named for the sense in which the instances are related. A relationship can be represented as a time-varying binary relation between the instances of the current extents of two state classes.

3.1.160 relationship instance: An association of specific instances of the related classes.

3.1.161 relationship name: A verb or verb phrase that reflects the meaning of the relationship expressed between the two entities shown on the diagram on which the name appears. [key style]

3.1.162 representation: One or more properties used by an algorithm for the realization of a responsibility.

3.1.163 representation property: A property on which an algorithm operates.

3.1.164 request: A message sent from one object (the sender) to another object (the receiver), directing the receiver to fulfill one of its responsibilities. Specifically, a request may be for the value of an attribute, for the value of a participant property, for the application of an operation, or for the truth of a constraint. *Request* also encompasses sentences of such requests. Logical sentences about the property values and constraints of objects are used for queries, pre-conditions, post-conditions, and responsibility realizations. *See also:* **message**.

3.1.165 respecialize: A change by an instance from being an instance of its current subclass to being an instance of one of the other subclasses in its current cluster. *Contrast:* **specialize**; **unspecialize**.

3.1.166 responsibility: A generalization of properties (attributes, participant properties, and operations) and constraints. An instance possesses knowledge, exhibits behavior, and obeys rules. These are collectively referred to as the instance's responsibilities. A class abstracts the responsibilities in common to its instances. A responsibility may apply to each instance of the class (instance-level) or to the class as a whole (class-level).

3.1.167 role name: (A) A name that more specifically names the nature of a related value class or state class. For a relationship, a role name is a name given to a class in a relationship to clarify the participation of that class in the relationship, i.e., connote the role played by a related instance. For an attribute, a role name is a name used to clarify the sense of the value class in the context of the class for which it is a property. **(B)** A name assigned to a foreign key attribute to represent the use of the foreign key in the entity. [key style]

3.1.168 Rule and Constraint Language: A declarative specification language that is used to express the realization of responsibilities and to state queries.

3.1.169 sample instance diagram: A form of presenting example instances in which instances are shown as separate graphic objects. The graphic presentation of instances can be useful when only a few instances are presented. *Contrast:* **sample instance table**.

3.1.170 sample instance table: A form of presenting example instances in which instances are shown as a tabular presentation. The tabular presentation of instances can be useful when several instances of one class are to be presented. *Contrast:* **sample instance diagram**.

3.1.171 scalar: A value that is atomic, i.e., having no parts. *Contrast:* **collection-valued**.

3.1.172 scalar property: *See:* **scalar-valued property**.

3.1.173 scalar-valued class: A class in which each instance is a single value. *Contrast:* **collection-valued class**.

3.1.174 scalar-valued property: A property that maps to a scalar-valued class. *Contrast:* **collection-valued property**.

3.1.175 semantics: The meaning of the syntactic components of a language.

3.1.176 set: A kind of collection class with no duplicate members and where order is irrelevant. *Contrast:* **bag**; **list**.

3.1.177 shadow class: A class presented in a view that is specified in some other view.

3.1.178 signature: A statement of what the interface to a responsibility "looks like." A signature consists of the responsibility name, along with a property operator and the number and type of its arguments, if any. A type (class) may be specified for each argument in order to limit the argument values to being instances of that class.

3.1.179 single-valued property: A property with a single-valued mapping. *Contrast:* **multi-valued property**.

3.1.180 single-valued: *Syn:* function. *Contrast:* **multi-valued**.

3.1.181 specialize: A change by an instance from being an instance of its current class to being additionally an instance of one (or more) of the subclasses of the current subclass. A specialized instance acquires a different (lower) lowclass. *Contrast:* **respecialize**; **unspecialize**.

3.1.182 specification language: *See:* **Rule and Constraint Language**.

3.1.183 split key: A foreign key containing two or more attributes, where at least one of the attributes is a part of the entity's primary key and at least one of the attributes is not a part of the primary key. [key style]

3.1.184 state class: A kind of class that represents a set of real or abstract objects (people, places, events, ideas, things, combinations of things, etc.) that have common knowledge or behavior. A state class represents instances with changeable state. The constituent instances of a state class can come and go and can change state over time, i.e., their property values can change.

3.1.185 static model: A kind of model that describes an interrelated set of classes (and/or subject domains) along with their relationships and responsibilities. *Contrast:* **dynamic model**.

3.1.186 subclass: A specialization of one or more superclasses. Each instance of a subclass is an instance of each superclass. A subclass typically specifies additional, different responsibilities to those of its superclasses or overrides superclass responsibilities to provide a different realization.

3.1.187 subclass cluster: **(A)** A set of one or more generalization structures in which the subclasses share the same superclass and in which an instance of the superclass is an instance of no more than one subclass. A cluster exists when an instance of the superclass can be an instance of only one of the subclasses in the set, and each instance of a subclass is an instance of the superclass. **(B)** A set of one or more mutually exclusive specializations of the same generic entity. [key style]

3.1.188 subclass responsibility: A designation that a property of a class must be overridden in its subclasses, i.e., the designation given to a property whose implementation is not specified in this class. A property that is a subclass responsibility is a specification in the superclass of an *interface* that each of its subclasses must provide. A property that is designated as a subclass responsibility has its *realization* deferred to the subclass(es) of the class.

3.1.189 subject domain: An area of interest or expertise. The responsibilities of a subject domain are an aggregation of the responsibilities of a set of current or potential named classes. A subject domain may also contain other subject domains. A subject domain encapsulates the detail of a view.

3.1.190 subject domain responsibility: A generalized concept that the analyst discovers by asking "in general, what do instances in this subject domain need to be able to do or to know?" The classes and subject domains in a subject domain together supply the knowledge, behavior, and rules that make up the subject. These notions are collectively referred to as the subject domain's responsibilities. Subject domain responsibilities are not distinguished as sub-domains or classes during the early stages of analysis.

3.1.191 substitutability: A principle stating that, since each instance of a subclass is an instance of the superclass, an instance of the subclass should be acceptable in any context where an instance of the superclass is acceptable. Any request sent to an instance receives an acceptable response, regardless of whether the receiver is an instance of the subclass or the superclass.

3.1.192 subtype: *Syn:* subclass.

3.1.193 superclass: A class whose instances are specialized into one or more subclasses. *See also:* **partial cluster**; **total cluster**.

3.1.194 supertype: *Syn:* superclass.

3.1.195 syntax: The structural components or features of a language and rules that define the ways in which the language constructs may be assembled together to form sentences.

3.1.196 total: A complete mapping. The mapping `M` from a set `D` to a set `R` is *total* if for every `X` in `D`, there is at least one Y in R and pair `[ X, Y ]` in `M`. A property of a class is total, meaning that it will have a value for every instance of the class, unless it is explicitly declared partial. *Contrast:* **partial**. *See also:* **mandatory**; **mapping completeness**.

3.1.197 total cluster: A subclass cluster in which each instance of a superclass must be an instance of at least one of the subclasses of the cluster. *Contrast:* **partial cluster**. *See also:* **superclass**.

3.1.198 type: *See:* **class**.

3.1.199 uniqueness constraint: A kind of constraint stating that no two distinct instances of a class may agree on the values of all the properties that are named in the uniqueness constraint.

3.1.200 unspecialize: A change by an instance from being an instance of its current subclass within a cluster to being an instance of none of the subclasses in the cluster. *Contrast:* **respecialize**; **specialize**.

3.1.201 updatable argument: The designation given to an operation argument that identifies an instance to which a request may be sent that will change the state of the instance. An argument not designated as "updatable" means that there will be no requests sent that will change the state of the instance identified by the argument.

3.1.202 value class: A kind of class that represents instances that are pure values. The constituent instances of a value class do not come and go and cannot change state.

3.1.203 value list constraint: A kind of constraint that specifies the set of all acceptable instance values for a value class.

3.1.204 value range constraint: A kind of constraint that specifies the set of all acceptable instance values for a value class where the instance values are constrained by a lower and/or upper boundary. An example of the value range constraint is `Azimuth,` which is required to be between –180° to +180°. A range constraint only makes sense if there is a linear ordering specified.

3.1.205 variable: An instance whose identity is unknown at the time of writing. A variable is represented by an identifier that begins with an upper-case letter.

3.1.206 verb phrase: (A) A part of the label of a relationship that names the relationship in a way that a sentence can be formed by combining the first class name, the verb phrase, the cardinality expression, and the second class name or role name. A verb phrase is ideally stated in active voice. For example, the statement "*each project funds one or more tasks*" could be derived from a relationship showing "`project`" as the first class, "`task`" as the second class with a "one or more" cardinality, and "funds" as the verb phrase. **(B)** A phrase used to name a relationship, which consists of a verb and words that constitute the object of the phrase. [key style]

3.1.207 view: (A) A collection of subject domains, classes, relationships, responsibilities, properties, constraints, and notes assembled or created for a certain purpose and covering a certain scope. A view may cover the entire area being modeled or only a part of that area. **(B)** A collection of entities and assigned attributes (domains) assembled for some purpose. [key style]

3.1.208 view diagram: A graphic representation of the underlying semantics of a view.

3.1.209 visibility: The specification, for a property, of “who can see it?”—i.e., whose methods can reference the property. Visibility is either private, protected, or public.

3.1.210 whitespace: The nondisplaying formatting characters such as spaces, tabs, etc., that are embedded within a block of free text.

3.2 Abbreviations and acronyms

ADT	abstract data type
BNF	Backus-Naur form
DBDG	Database Design Group
DBMS	database management system
DDL	Data Definition Language
DISA	Defense Information Systems Agency
DMSO	Defense Modeling and Simulations Office
D/P	Data/Process
ER	entity-relationship
FA	fully attributed
FIPS	Federal Information Processing Standard
GUI	Graphical User Interface
I^{2}S^2	Integrated Information Support System
KB	key-based
LDDT	Logical Database Design Technique
NIST	National Institute of Standards and Technology
oid	object identifier
OO	object-oriented
RCL	Rule and Constraint Language
SQL	Structured Query Language
UOD	universe of discourse

4. IDEF1X language overview

IDEF1X is a language, and like any language it has parts of speech. For example, the classes and instances are the nouns, and the relationships are roles that instances of one class may play relative to instances of another class. The responsibilities are the knowledge that the classes and instances may possess, the behaviors that the classes and instances may exhibit, and the rules that they must obey.[5] Each of these “parts of speech” has a particular meaning and, because of that meaning, each may be combined with others only in specific ways. For example, because of what they represent, it does not make sense in IDEF1X to have relationships between relationships. This standard establishes what the valid constructs are and which possible combinations of IDEF1X modeling constructs constitute a valid model.

In Clauses 5 through 6 and Clause 8, the meaning of each basic IDEF1X construct is informally described in English, the graphic syntax for the construct (where there is one) is stated and illustrated, and any rules for using the construct are listed.[6] Clause 7 provides a full specification of the RCL syntax and semantics. Clause 10 provides a formal specification of each of the IDEF1X concepts in first-order language. Every effort has been made to insure that the English explanation of these constructs is complete and accurate. However, if there appears to be an inconsistency between the English description of a construct and the formalization or the language specification of that construct, the formalization or the language specification (in that order) is the authoritative statement.

[5]Constraint is the only form of rule discussed in this version of the standard.

[6]Rules apply to a completed model (model construct) and not one that is in development.

4.1 IDEF1X language constructs

This clause introduces the constructs of the IDEF1X language. Only a summary is given here; details and numerous examples are provided in later clauses. The language constructs of IDEF1X include

a) Class. A *class* is an abstraction of the knowledge and behavior of a set of similar things. Anything that is classified into a class is said to be an *instance* of the class. All the instances of a given class have the same *responsibilities*, i.e., they possess the same kinds of knowledge, exhibit the same kinds of behavior, and adhere to the same rules. An instance is a discrete, bounded thing with an intrinsic, immutable, and unique *identity.*

   Each class is either a state class or a value class.

   1) State Class. A *state class* represents instances with changeable state. Its instances can come and go and can change state over time, i.e., their property values can change.
   2) Value Class. A *value class* represents instances that are pure values. Its instances do not come and go and cannot change state.

b) Generalization. Classes are used to represent the notion of "things whose knowledge or actions are relevant." Since some real world things are generalizations of other real world things, some classes must, in some sense, be generalizations of other classes. A class that specifies additional, different responsibilities to those of a more general class is known as a *subclass* of that more general class (its *superclass*). Each instance of the subclass represents the same real-world thing as its instance in the superclass. The structuring of classes as a generalization taxonomy (hierarchy or network) determines the inheritance of responsibilities among classes.

c) Relationship. A *relationship* expresses a connection between two state classes that is deemed relevant for a particular scope and purpose. It is named for the sense in which the instances are related.

d) Responsibility. An instance possesses *knowledge*, exhibits *behavior*, and obeys *rules*. These notions are collectively referred to as the instance's *responsibilities*. A class abstracts the responsibilities in common to its instances. During initial model development, a responsibility may simply be stated in general terms and not distinguished explicitly as an attribute, participant property, operation, or constraint. Also, aggregate responsibilities may be specified, rather than individual properties. Broadly stated responsibilities are eventually refined as specific properties and constraints.

   1) Property. Some responsibilities are met by knowledge and behavior which, in turn, are determined by properties. A *property* is an inherent or distinctive characteristic or trait that manifests some aspect of an object's knowledge or behavior. There are three kinds of property: *attributes*, *participant properties* due to relationships, and *operations*. Classes have properties; instances have *property values*.

      i) Attribute. An *attribute* is a mapping from a class to a value class. An attribute expresses some characteristic that is generally common to the instances of a class. The name of the attribute is the name of the role that the value class plays in describing the class, which may simply be the name of the value class (as long as using the value class name does not cause ambiguity[7]).

      ii) Participant property. A *participant property* is a mapping from a state class to a related (not necessarily distinct) state class. When a relationship exists between two state classes, each class contains a participant property for that relationship. The name of each participant property is the name of the role that the other class plays in the relationship, or it may simply be the name of the class at the other end of the relationship (as long as using the class name does not cause ambiguity[8]). The value of a participant property is the identity of a related instance. For a relationship in which there may be many related instances, there is also a participant property named as described above but suffixed with `(s)`, which is a mapping from the state class to a collection class in which the members of the collection are the related instances.

[7] Ambiguity would exist if there were multiple mappings between a class and value class and different role names were not used.

[8] Ambiguity would exist if there were more than one relationship between the same pair of classes and different role names were not used.

iii) Operation. The operations of a class specify the behavior of its instances. An *operation* is a mapping from the (cross product of the) instances of the class and the instances of the input argument types to the (cross product of the) instances of the other (output) argument types. While an attribute or participant property is an abstraction of what an instance knows, an operation is an abstraction of what an instance does.

2) Constraint. Other responsibilities are met by adhering to constraints. A *constraint* is a statement of facts that are required to be true for a class or the instances of a class. Constraints are expressed in the form of logical sentences about property values or constraints. An instance conforms to the constraint if the logical sentence is true for that instance. Some constraints are inherent in the modeling constructs and can be readily represented using the graphics; other constraints are specific to a particular model and are stated in the specification language.

3) Note. A *note* is a body of free text that describes some general comment or specific constraint about a portion of a model. A note may be used in an early, high-level view prior to capturing constraints in the specification language; a note may further clarify a rule by providing explanations and examples. A note may also be used for "general interest" comments not involving rules. These notes may accompany the model graphics.

e) Request. A request is a message sent from one object (the sender) to another object (the receiver), directing the receiver to fulfill one of its responsibilities. Specifically, a request may be for the value of an attribute, for the value of a participant property, for the application of an operation, or for the truth of a constraint.

f) Realization. The realization of a responsibility specifies how the responsibility is met. A realization is stated as a logical sentence giving the necessary and sufficient conditions that the responsibility be met.

g) Model infrastructure constructs. Modeled constructs are presented in views and packaged as models that provide supporting elements of documentation such as textual descriptions.

1) View. A *view* is a collection of subject domains, classes, relationships, responsibilities, properties, constraints, and notes (and possibly other views) assembled or created for a certain purpose and covering a certain scope. A view may cover the entire area being modeled or only a part of that area.[9]

2) Level. A *level* is a designation of the coverage and detail of a view. There are multiple levels of view.

3) Environment. An *environment* is a concept space, i.e., an area in which a concept has an agreed-to meaning and one or more agreed-to names that are used for the concept. Every view is developed for a specific environment.

4) Glossary. A *glossary* is the collection of the names and descriptions of all terms that may be used for defined concepts (views, subject domains, classes, relationships, responsibilities, properties, and constraints) within an environment. A *model glossary* is the collection of the names and descriptions of all defined concepts that appear within the views of a model.

5) Model. A *model* is a packaging of one or more views along with the narrative descriptions and specification language for the view and view components (classes, responsibilities, etc.) called out in the model's views.

4.2 IDEF1X notation

The IDEF1X notation includes diagrams, free text, and the specification language.

a) Diagrams present the subject domains, classes, responsibilities, relationships, attributes, operations, and constraints of interest in a view.

b) Free text is used for labels, statements of responsibilities, narrative description, and notes.

[9] Every view automatically contains the classes, relationships, etc., of the IDEF1X$_{97}$ language metamodel, although these classes, relationships, etc., are customarily not shown in the graphics and the modeler does not declare them.

c) The specification language is used for names, requests, statements of pre- and post-conditions, and the realization of operations, constraints, and derived attributes.

4.2.1 Example IDEF1X diagram

Figure 3 illustrates some of the aspects of classes and relationships that are described in Clause 5. The classes `hotel` and `room` are state classes; `room` is a dependent state class, and `hotel` is an independent state class (see 5.2). `temperature` is a value class (see 5.3). The relationship between `hotel` and `room` says that *"each hotel contains rooms"* (see 5.5). While not shown in Figure 3, Clause 5 also discusses the generalization of classes in 5.4.

Responsibility names, if shown, are listed inside the rectangles in the graphic diagram. Responsibilities are described in Clause 6. This diagram does not distinguish graphically among attribute, participant, and operation properties. The annotations to do so are covered in 6.3.

Specifying the type of an attribute is optional. In Figure 3, types are shown only for the `temperature` class, i.e., `fahrenheit`, `celsius`, `kelvin`, and `rep` are all of type `real`. Within the `temperature` class, the attribute `rep` is marked "private" as indicated by the double bar preceding the attribute name; this is the hidden value (which could be any one of the other values or perhaps something different entirely) that the `temperature` class uses to represent temperature values.[10] The property `guest` is marked with an `(o)` to designate that a value is optional; that is, a room may be vacant. The full set of suffix annotations are covered in 6.3.

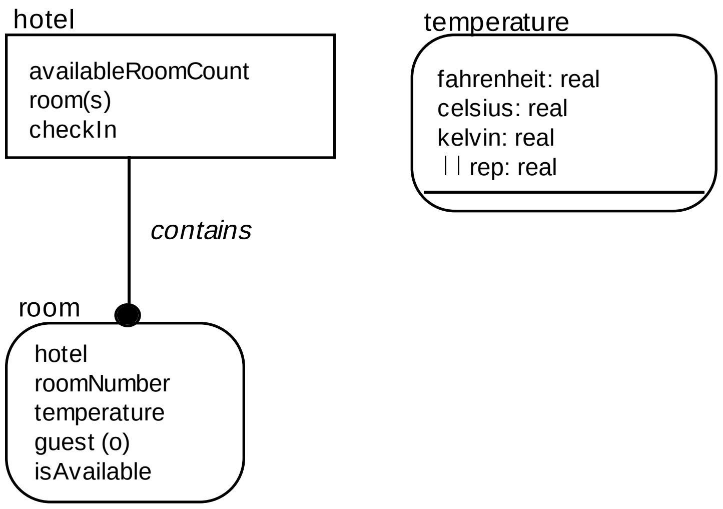


Figure 3—An example IDEF1X diagram

In addition to attributes, the property list displayed for a class may include participant properties and operations. Because of the relationship between `hotel` and `room`, the class `room` has a participant property named `hotel,` and the class `hotel` has a participant property named `room(s)`, where the `(s)` suffix reflects its plural (collection-valued) nature. While Figure 3 does display these participant properties, they are not always displayed in a diagram, as explained in 6.5. This figure also illustrates an operation, `checkIn`, for the class `hotel`.

It is sometimes useful to illustrate instance examples for classes. Two methods for doing so are shown in Figures 4 and 5. Figure 4 portrays what is referred to as a *sample instance diagram*. In this figure, sample instances of `hotel`, `room`, and `temperature` are shown as separate shapes. Above each rectangle is

[10]See 6.4.5 for a description of how representation properties are used to derive the public attribute values.

written the intrinsic object identifier (oid) of the instance. Inside the rectangles are written property names and their values. Collection-valued property instance values are shown within brackets with a literal form appropriate to the collection class (here with curly braces for `set`).

Figure 5 shows what is referred to as a *sample instance table*. The figure shows the same sample instances of `hotel`, `room`, and `temperature`, this time in the form of tables representing the classes. Columns in each table represent properties and the cells hold property values. The oid of each instance is shown to the left of the row representing the instance. When a property has no value, a "double dash" (`--`) is shown in sample instance tables and diagrams.

In the diagrams and specification language, proper nouns denoting a specific but (at the time of writing) unknown instance of a class are written with an initial capital letter. For example, `TheHotel` and `R1` each achieve the same end—standing (arbitrarily) for instances of `hotel` and `room`, respectively. Any other nouns are written with an initial lowercase letter or surrounded by single quotes. Sample instance diagrams and tables are discussed more fully in 5.2 and 5.3.

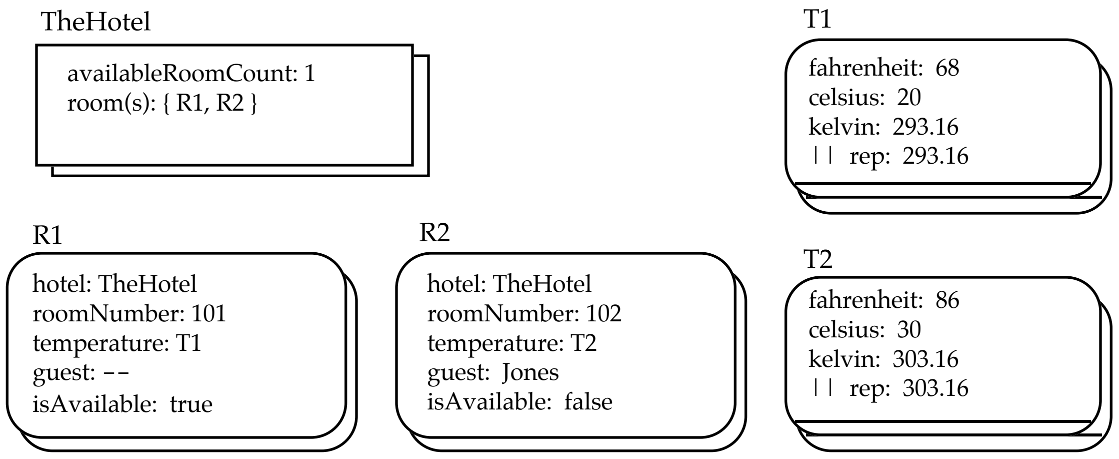


Figure 4—Sample instance diagram

4.2.2 Example specification language

IDEF1X includes a declarative specification language called the Rule and Constraint Language (RCL). Below are four examples of its use.[11] The specification language is fully described in Clause 7.[12]

4.2.2.1 Example attribute derivation specification

The specification language is used to express the realization of operations, constraints, and derived attributes and participant properties. For example, the `availableRoomCount`[13] derived attribute of `hotel` in Figure 3 has the following realization:

[11]The first and third examples also illustrate that the specific (but unknown) instance in a declaration may be named in various ways, e.g., as `TheHotel` or as `Self`. This standard does not proscribe any particular style. For example, while `Self` is a form familiar to a Smalltalk user, a C++ or Java user might have selected `This`.

[12]Several equivalent forms for RCL are described in Clause 7. The examples throughout do not attempt to show all equivalent forms.

[13]Several forms of expressing a name are allowed by the lexical rules specified in 4.2.3. This form is used for consistency throughout.

hotel

	availableRoomCount	room(s)
TheHotel	1	{R1, R2}

room

	hotel	roomNumber	temperature	guest	isAvailable
R1	TheHotel	101	T1	--	true
R2	TheHotel	102	T2	Jones	false

temperature

	fahrenheit	celsius	kelvin	|| rep
T1	68	20	293.1	293.16
T2	86	30	303.1	303.16

Figure 5—Sample instance tables

```
hotel: TheHotel has availableRoomCount: N if_def
             AvailableRooms is [ Room where
                Room is TheHotel..room(s)..member,
                Room has isAvailable ],
             N is AvailableRooms..count.
```

`TheHotel` is the instance of `hotel` that is the receiver of the request for the `availableRoomCount` derived attribute value. The specification language says that in order for `TheHotel` to have an `availableRoomCount` value of `N`, it is necessary and sufficient that

a) `AvailableRooms` is a list containing every `Room` where the `Room` is `TheHotel`'s `room`, and the `Room` `isAvailable`, and
b) `N` is the `AvailableRooms`' `count`.

The specification of derived attributes and participant properties is discussed more fully in 6.4 and 6.5, respectively.

4.2.2.2 Example operation realization specification

Another example provides a declarative specification of the operation `checkIn` in Figure 3.

```
hotel: Self has checkIn: Guest if_def
             Self has room(s)..member: R,
             R has isAvailable,
             R has guest:= Guest,
             R has isAvailable:= false.
```

When requested, a particular hotel instance will “check in” the identified guest if the hotel has an available room. A successful execution of the `checkIn` operation assigns the guest to the available room and makes that room unavailable. The specification of operations is discussed more fully in 6.6.

4.2.2.3 Example constraint specification

The specification language can be used to state constraints—statements of facts that are required to be true in order that the model conform to the world being modeled. An example of the declarative specification of the constraint `hotelOwnsTv` (see Figure 70) is

```
tvInARoom: Self has hotelOwnsTv if_def
             Self..room..hotel == Self..tv..hotel.
```

This constraint states that a television in a room has “valid ownership” if the hotel that contains the room of this television and the hotel that owns the television are precisely the same hotel. The specification of constraints is discussed more fully in 6.7.

4.2.2.4 Example query specification

The specification language is also used to make queries (requests for property values). In Figure 3, the value class `temperature` represents abstract temperatures. Each instance of `temperature` has both a `fahrenheit` and a `celsius` value (as well as a `kelvin` value). The specification language sentence

```
T is temperature with fahrenheit: 68,
C is T..celsius.
```

is a query that identifies `T` as the instance of `temperature` with a `fahrenheit` value of `68` and requests that instance’s corresponding `celsius` value. `C` is whatever the `celsius` value is for that same instance `T`. If the sentence is executed, `C` will be solved for and found to be `20`.

4.2.3 Lexical rules

A *name* is a word or phrase that designates some model construct (such as a class, responsibility, subject domain, etc.). A *label* is a word or phrase that is attached to or part of a model graphic; it typically consists of a model construct’s name (or one of the aliases) and may contain additional textual annotations (such as a note identifier). Refer to Clause 7 for the formal RCL syntax.

Free text shall be used for the names and labels of IDEF1X constructs, according to the following rules:

4.2.3.1 Naming

a) An *unquoted name* (i.e., a name not surrounded by single quotes) shall contain only alphanumeric characters and underscores.
b) A *quoted name* (i.e., a name surrounded by single quotes) may contain any character.
c) A quoted name shall specify an imbedded single quote by two adjacent single quotes.
d) A *simple name* shall be an unquoted name or a quoted name.
e) A *qualified name* shall be a series of simple names separated by colons.
f) The single quotes surrounding a quoted name may be omitted in the model graphics.
g) A name of the form `#K`, where `K` is a constant, shall be used to denote a state class instance that is known at the time of writing (see Figure C.23)
h) A name denoting a specific but (at the time of writing) unknown instance of a class shall begin with an uppercase letter.
i) Any other name shall begin with an initial lowercase letter or shall be surrounded by single quotes.
j) A name may not exceed 254 characters in length. All characters shall be treated as significant.
k) If a valid unquoted name begins with a lowercase letter, the name shall be considered equivalent to the same name surrounded by single quotes.
l) If a name used in model graphics is not a valid unquoted name, then it shall be surrounded by single quotes when used in RCL.
m) Each keyword shall be in lowercase.

4.2.3.2 Label

a) Except for the first character, a label shall be case insensitive, i.e., “`A`” and “`a`” are equivalent.
b) A label may “wrap” and be displayed as multiple lines.
c) A label displayed outside its associated graphic box (e.g., a class label outside its class box) may not extend beyond the right bounding line of the graphic box.
d) A label displayed inside its associated graphic box (e.g., a class label inside its class box or a displayed property signature) may not extend beyond the bounding lines of the graphic box.
e) On a multiline label, any annotation symbols (e.g., note numbers) shall be included at the end of the last line of the label.
f) All *whitespace* (spaces, tabs, etc.) in a label shall be preserved.

An example of a state class label that includes a note is

```
purchase-Order-Item (12)
```

An example of a multi-line relationship verb phrase label that includes a note is

```
is
assigned
to (5)
```

5. Class

People mentally classify things that are similar in some sense into a *class* named for that sense and representing all such similar things. Everyone does this classification; it is part of common sense. The things that are classified in this way are individual things, distinct from all other things. A class is an abstraction of the knowledge and behavior of a set of similar things.

5.1 Introduction

There are two kinds of class: state class and value class.[14] The distinction is introduced in 5.1.3 and more fully discussed in 5.2 and 5.3, respectively. This subclause describes the concepts that apply to the concept of *class* in general.

Anything that is categorized into a class is said to be an *instance* of the class. An instance possesses knowledge, exhibits behavior, and obeys rules. These notions are collectively referred to as the instance’s *responsibilities*. A class abstracts the responsibilities in common to its instances. Initially, a responsibility may simply be stated in general terms and not distinguished explicitly as an attribute, participant property, operation, or constraint. Also, aggregate responsibilities may be identified, rather than individual properties. Broadly stated responsibilities are eventually refined as specific properties and constraints. In addition to these instance-level responsibilities, a class may also have *class-level* responsibilities in the form of attributes, operations, and constraints. These class-level responsibilities constitute the knowledge, behavior, and rules of the class as a whole. Responsibilities are described in detail in Clause 6.

5.1.1 Class semantics

5.1.1.1 Identity

Each instance is considered to have a unique, intrinsic *identity* that is independent of its property values or the classes to which it belongs. It is an instance’s unique identity that distinguishes it from all other instances.

[14]This distinction is explicitly made in “The Evolution of Domains” [B4] and ODMG-93 [B11] where state classes and value classes are referred to mutable and immutable classes, respectively. However, this distinction is often left implicit in object model formulations.

The notion of identity is first of all a concept. Some concrete representation for the concept must be used to show examples of instances with identity, to formalize the notion of identity, or to support the notion in programming languages or database systems. This concrete representation is referred to as the intrinsic oid of the instance.

5.1.1.2 Intension/extension

The notion of class has an intensional and an extensional aspect. The *intension* reflects the sense of the class. The *intensional set* is determined by the meaning of the class. All possible things that are similar in the sense of the class are members of the intensional set.

The *extensional set* of instances contains the currently existing instances. The extensional set is always a subset of the intensional set.[15] The instances in the extensional set correspond to the database and data modeling notion of *instance*. The extensional set is sometimes called the *current extent*. Unless otherwise noted, *instance* means an existing instance, that is, a member of the current extent.

5.1.1.3 State class/value class

A *state class* is one in which the extensional set of instances is a time-varying subset of the intensional set of instances. An instance changes state when it is born, when it takes on attribute or participant property values, when it changes those values, or when it dies. A class of such instances is called a state class. The class `registeredVoter` is an example of a state class. The extensional set of registered voters varies over time. `Chris Jones` could be an instance of the state class `registeredVoter` at any particular time.

Instances that do not change state are pure values. A class of such instances is called a *value class*. A value class is one in which the extensional set of instances is fixed and equal to the intensional set of instances. The instances act as pure values, like an integer or a mathematical set. It makes no sense to have duplicate instances—there is only one `17`, only one `0`, only one empty set, and so on. Because the instances of a value class act like values, a value class instance is sometimes called a *value*, but it is still an instance.

Value class instances cannot be created, updated, or deleted. It makes no sense to update `17`; it would not be `17` any more. If a mathematical set has a member removed or a new one is added, it is not the same set any more. Everyday examples of value classes include `date` and `time`. Again, “updating” a date or a time makes no sense; it would yield a different date or time, not the same one changed in some way.

Everything that is said about classes, instances, and properties applies to both state and value classes unless specifically restricted. For example, value classes do not participate in relationships. Also, the representation of identity is typically different for state and value classes—for state classes, the identity is represented by the intrinsic identifier; for value classes, the identity is represented by the value. The details of state classes and value classes are discussed in detail in 5.2 and 5.3, respectively.

5.1.1.4 Abstract data type

A class can be considered an abstract data type. Traditionally, an *abstract data type* (ADT) is a data type for which the user of the data type can create instances of the data type and operate on those instances, but for which the range of valid operations available to the user does not depend in any way on the internal representation of the instances or the way in which the operations are realized. The data is abstract in the sense that values in the extent, i.e., the concrete values that represent the instances, are

— Any set of values that support the operations, and
— Irrelevant to the user.

[15]This fact forms the foundation of *information*. Otherwise, everything that could possibly be true is in the database rather than just those things that are presently true.

An ADT identifies the operations on the data as part of the specification of the data and separates what can be done (the interface) from how it is done (the realization).[16]

For example,[17] the interface declaration of the value class `temperature` specifies

— A way to denote a unique temperature by its Fahrenheit or Celsius or Kelvin value, and
— Operations to obtain the Fahrenheit, Celsius and Kelvin values.

The realization of `temperature` specifies

— Whether the temperature is to be represented by its Fahrenheit or Celsius or Kelvin value, and
— The appropriate rules for each operation, depending on the representation choice.

The nature of the abstract values in the extent of the `temperature` value class is exemplified by the sentence "32 Fahrenheit is the same thing as 0 Celsius." That thing is the instance of temperature. When the `fahrenheit` operation is applied to the instance, it yields `32`. When the `celsius` operation is applied to the same instance, it yields `0`.

For another example, the interface declaration of the value class `vector` specifies

— A way to denote a unique vector by its coordinate values or by its magnitude and direction,
— Operations to obtain the values of the coordinates, magnitude, and direction, and
— Operations such as adding a vector to a vector.

The realization of the value class `vector` specifies

— Whether the vector is to be represented by coordinate values, or by magnitude and direction values, and
— The rules for each operation.

5.1.1.5 Built-in class/user-defined class

A few classes, such as `object`, `class`, `integer`, `real`, `string`, and `list,` are assumed to be *built-in* to the IDEF1X language. These classes provide properties such as instance creation, instance deletion, access to the instances of a class, and numeric, string, and list operations. Classes that are not built-in are *user-defined*. The built-in state classes and built-in value classes are presented in Clause 10 and Annex D.

5.1.1.6 Collection class

The typical class represents instances that are atomic. Some classes, however, have instances that are themselves collections of other instances. Such a class is called a collection class. A *collection class* is a kind of class in which each instance is a group of instances of other classes. Examples of collection classes are `list`, `set`, and `bag`.

5.1.1.7 Parameterized collection class

The built-in `list`, `set`, and `bag` collection classes are untyped, i.e., their instances can be collections of anything. However, a collection class can be restricted to hold only instances of a specified type (class). This kind of collection class is called a *parameterized collection class*, a class in which a parameter specifies the class of the instances that the collection may contain.

[16] "Interface" and "realization" are explained in 6.1. See also [B12].
[17]These examples are developed further in 5.3, 6.4, and 6.6.

The built-in collection class generators `list ( T )`, `set ( T )`, and `bag ( T )` permit typed collection classes to be specified. A class is “generated” in the sense that the classes `set ( real )` and `set ( integer )` are two distinct classes, not a single class with a parameter variable. For example, the instances of `set ( real )` are restricted to be sets of real numbers. The collection class `set ( integer )` is a different collection class; its instances are restricted to be sets of integers.

The parameter, `T`, can be any built-in or user-defined class. All the properties and constraints of the built-in collection classes apply to the generated collection classes. (See Clause 7 for a further discussion of collection classes.)

5.1.1.8 Parameterized pair class

The built-in `pair` class is untyped, i.e., its instances can be pairs of anything. However, a pair class can be restricted to hold only instances of specified types (classes). This kind of pair class is called a *parameterized pair class*, a class in which two parameters specify the classes of the instances that the pair may contain.

The built-in pair class generator `pair ( T1, T2 )` permits typed pair classes to be specified. A class is “generated” in the sense that the classes `pair ( real, integer )` and `pair ( integer, string )` are two distinct classes, not a single class with parameter variables.

The parameters, `T1` and `T2`, can be any built-in or user-defined classes. All the properties and constraints of the built-in pair class apply to the generated pair classes. (See Clause 7 for a further discussion of pair classes.)

5.1.2 Class syntax

5.1.2.1 Graphic

a) A class shall be represented as a rectangle of the shape appropriate to its class.

The shapes for state class are specified in 5.2.2.1. The shape for value class is specified in 5.3.2.1.

5.1.2.2 Label

a) Each class displayed in a view shall be assigned a label.
b) The label of a class in a view shall consist of the class name or one of its aliases.

The syntax for state class labeling is specified in 5.2.2.2. The syntax for value class labeling is specified in 5.3.2.2.

5.1.2.3 Sample instances

a) For every model it shall be possible to present sample instances that validate the model.
b) When provided, sample instances shall be presented in one of two forms, as a *sample instance diagram* or *sample instance table*.

The representation of sample instances of a state class is specified in 5.2.2.3 through 5.2.2.6. The representation of sample instances of a value class is specified in 5.3.2.3 through 5.3.2.6.

5.1.3 Class rules

5.1.3.1 Naming

a) A class shall have both a simple (unqualified) name and a fully qualified name.

b) The simple name of a class shall be a noun or noun phrase.

c) A class shall be given a simple name as one would refer to a single instance of the class. Typically that name is singular in form, not plural.

   For example, an instance of a state class may represent a set of things. If a state class instance represents a collection of things, as in a set of playing cards, a plural noun (e.g., `cards`) could be used for the state class name (although a singular form such as `deck` would also be appropriate).

   For example, an instance of a value class may have a list as its internal representation. If a value class instance is plural, as in a set of coordinates, a plural noun (e.g., `coordinates`) would be appropriate for the value class name.

d) A class shall have a fully qualified name,[18] as follows:
   1) The fully qualified name of a class with a simple name `Csn` in a view named `Vn` shall be `Vn:Csn`.
   2) The fully qualified name of a class with the simple name `Csn` in a view with no parent view shall be just `Csn`.

5.1.3.2 Responsibilities

a) A class may have any number of responsibilities.

5.2 State class

A state class[19] represents instances with changeable state. Its instances can come and go, and can change state over time, i.e., their property values can change.

A state class is a class that represents a set of real or abstract objects (people, places, events, ideas, things, combinations of things, etc.) that have common knowledge and behavior and adhere to common constraints. An individual member of the set is referred to as a *state class instance* (simply, *instance*). A real world thing may be represented by more than one state class. For example, `John Doe` can be an instance of both the state class `employee` and the state class `buyer`. Furthermore, an instance may represent a concept involving a combination of real world things. For example, `John` and `Mary` could be the participants in an instance of the state class `marriedCouple`.

5.2.1 State class semantics

5.2.1.1 Instance identity

For a state class, an oid represents the concept of identity. In terms of a representation system (i.e., the examples, formalization, or software), the oid stands for the instance. In a sample instance diagram or sample instance table of state class instances, each row has an associated oid. For example, oids are shown in the sample instance diagram in Figure 13 and the sample instance tables in Figure 14. Note that an oid is not an attribute; the oid is always hidden from the client.

5.2.1.2 Independent state class/dependent state class

A state class instance is distinguished from all other instances of its class because of its intrinsic identity; that is, the typical state class is considered to be an *independent state class*. Its instances have existence, knowledge, and behavior independent of other instances. However, there may be cases where it makes no sense to have a class instance by itself and unrelated to an instance of another class(es) and, furthermore, where it makes no sense to change the instance(s) to which it relates. This type of class is referred to as a

[18] See also 8.1.3.1 and 10.7.2.

[19] “State class” is not to be confused with the notion of a “state machine.”

dependent state class. A dependent state class instance is by its very nature *intrinsically* related to certain other state class instance(s).[20]

Both cases are illustrated in Figure 6. In this example, `room` is a dependent state class, dependent on the independent state class `hotel`. A room of the hotel cannot exist without its hotel. It makes no sense to separate the room from the hotel; the room would not exist. Furthermore, a hotel room is intrinsically a part of some one specific hotel (at least in the example), and it makes no sense to change it to a different hotel—to do so would yield a different hotel room, not the same one changed.

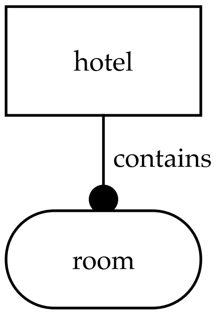


Figure 6—Independent and dependent state classes

State class dependency can be expressed more precisely. A state class `d` is dependent on another state class `c` if and only if every instance `D` of `d` is related to exactly one instance of `c`, and `D` cannot be updated to be related to any other instance of `c` or to no instance.

The distinction between "independent" and "dependent" has historically proven useful in IDEF1X modeling. In building a model, the independent state classes usually emerge first. In seeking to understand a model, a familiarity with the independent classes is required before the meaning of the dependent classes can be understood.

5.2.2 State class syntax

5.2.2.1 Graphic

a) A state class shall be represented as a rectangle.
b) An independent state class rectangle shall have square corners, as illustrated in Figure 7.
c) A dependent state class rectangle shall have rounded corners, as illustrated in Figure 7.

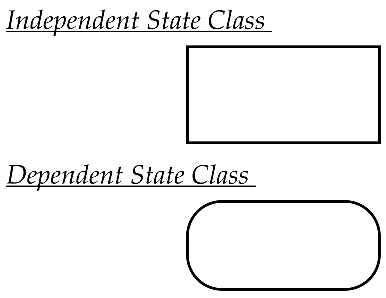


Figure 7—State class syntax

[20]Although an instance of a state class may depend on another instance, it still has its own identity, as indicated previously.

5.2.2.2 Label

a) As shown in Figures 8 and 9, the state class label shall be placed either
   1) Above or inside the rectangle, when no names (responsibilities, property names, or constraint names) are shown, or
   2) Above the rectangle, when names (responsibilities, property names, or constraint names) are shown inside the rectangle.

b) When placed outside the box, the state class label shall be left-justified and aligned with the left side of the box.

c) When placed inside the box, the state class label shall be centered inside the box.

d) Responsibilities, property names, and constraint names placed inside the state class box shall be left-justified.

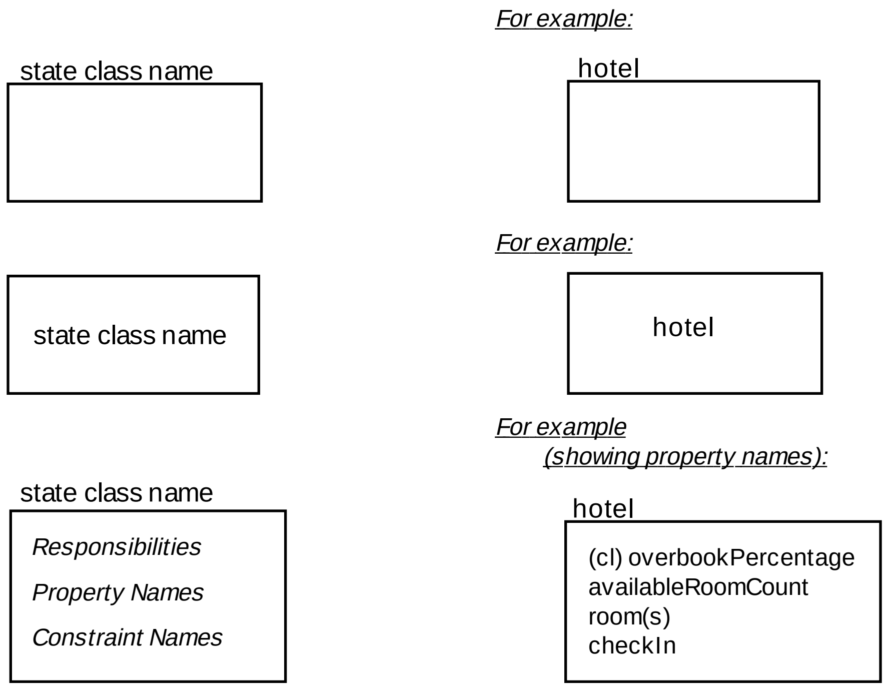


Figure 8—Alternatives for independent state class labeling

5.2.2.3 Sample instance identity label

In providing illustrative examples of instances, it is useful to have a way of representing an instance's identity.

a) The identity of an instance that is a *variable*—i.e., an instance unknown at the time of writing[21]—shall begin with an uppercase letter, such as

```
X
TheHotel
Self
```

[21]A *variable* represents a named but unknown value. This concept can be illustrated by analogy. In stylized English, the phrase "...the sale consummated by the seller (Seller) and the purchaser (Purchaser)..." may be used in writing a standard contract where the identity of the specific Seller was unknown at the time of writing. The originally indefinite references are definite on a signed contract, where Seller and Purchaser are identified as real parties. Similarly, the use of a variable denotes a specific individual—just which individual is unknown at the time of writing.

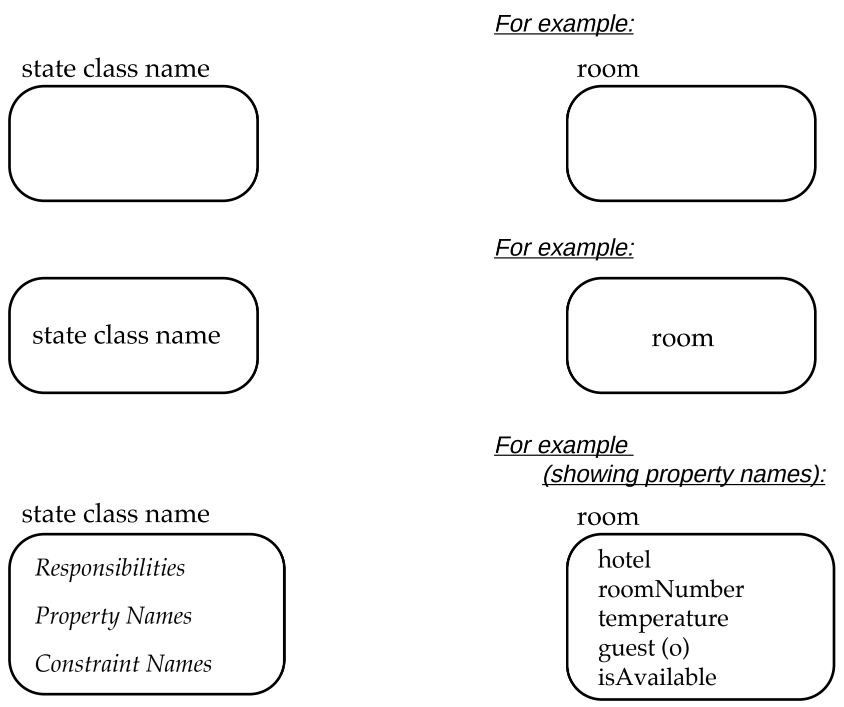


Figure 9—Alternatives for dependent state class labeling

b) The identity of an instance that is a *constant*—i.e., an instance known at the time of writing—shall be represented by `#K`, where `K` is a constant. This representation of a constant oid is most generally used in sample instance tables or diagrams, as shown in Figure C.23.
c) The state class name of the instance may be included along with its identity. In this form, the class name shall precede its identity and shall be followed by a colon, such as
   `hotel: TheHotel`
d) An unnamed, unknown instance shall be indicated by omitting the identity portion and including only the state class name of the instance, such as
   `hotel`

5.2.2.4 Sample instance property

a) Any relevant property of the instance, either direct or inherited, may be shown for a sample instance.
b) The sample instance property shall have two parts:
   1) A sample instance property label, and
   2) A sample instance property value.
c) While the sample instance property label is typically the property name, the sample instance property label may be any RCL expression over properties of that class, e.g., `principal + interest`.
d) The sample instance property value shall be the value to which the expression evaluates for the instance. If the expression value pair is `E: V`, then `Self has E: V` shall hold where
   1) `Self` is the identity of the instance (for instance properties), or
   2) `Self` is the class (for class properties).
e) A sample instance property having no value shall be indicated by “`--`” (a double dash).
f) A collection-class-valued sample instance property value shall be represented by a collection class literal (such as that for a set using curly brackets), with the multiple values separated by commas (see Figures 13 and 14).
g) A multi-valued sample instance property value shall be represented by multiple values separated by spaces.

5.2.2.5 Sample instance diagram

In a sample instance diagram, instances are shown as separate shapes.

a) In a sample instance diagram, an instance shall be represented by an open-stacked rectangle of the kind appropriate to its class, as shown in Figure 10. Each rectangle is either
   1) An open-stacked rectangle (if the instance belongs to an independent state class), or
   2) An open-stacked rounded rectangle (if the instance belongs to a dependent state class).
b) In a sample instance diagram, the instance's identity label shall be placed either
   1) Inside the rectangle (when no responsibilities, property names, or property name value pairs are shown), or
   2) Above the rectangle (when sample instance properties are shown inside the rectangle).

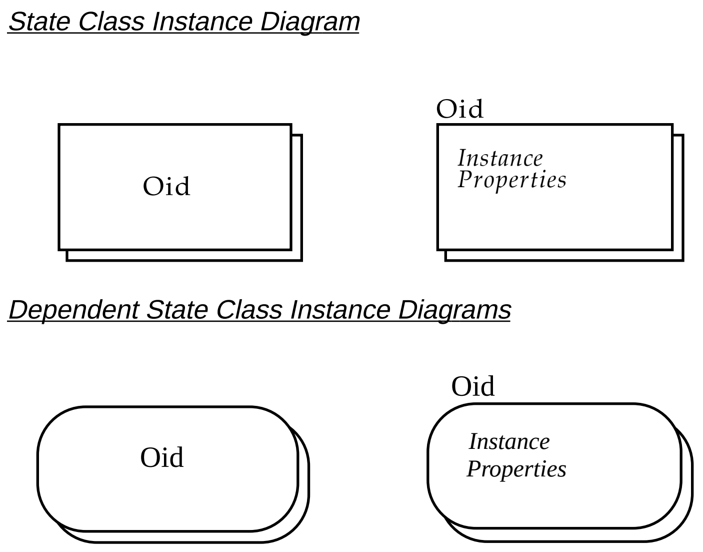

State Class Instance Diagram
Oid
Oid
Instance Properties
Dependent State Class Instance Diagrams
Oid
Oid
Instance Properties

Figure 10—State class instance diagram syntax

c) In a sample instance diagram, a sample instance property shall be written as
   `property label: sample property value`
d) The values of class-level properties shall be shown in an unstacked box, labeled only by the class name, as shown in Figure 11.
e) An alternative form of syntax shall be available for a state class instance diagram. This form shall use only a reference to the state class to represent an unnamed, unknown instance, as illustrated in Figure 12.

The example in Figure 13 shows two instances of `room` and a single instance of `hotel` (with the value of `hotel`'s class-level property).

5.2.2.6 Sample instance table

A second form of showing sample state class instances is as a sample instance table. This tabular presentation of instances can be useful when several instances of one class are to be presented. The conventions for a sample instance table are illustrated in Figure 14, which depicts the class `hotel`, one instance of `hotel`, and two instances of `room`.

a) In a sample instance table, the class name shall be placed above the table.
b) In a sample instance table, instances shall be shown as rows in a table representing the class.

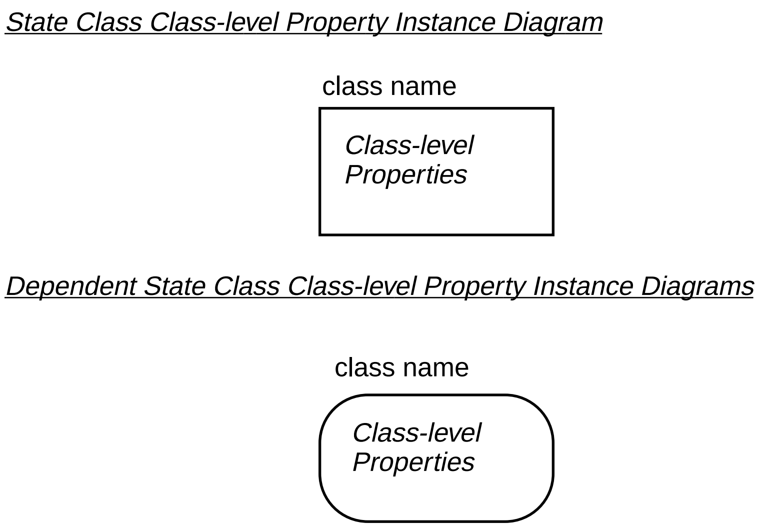

State Class Class-level Property Instance Diagram
class name
Class-level Properties
Dependent State Class Class-level Property Instance Diagrams
class name
Class-level Properties

Figure 11—Class-level property instance diagram syntax

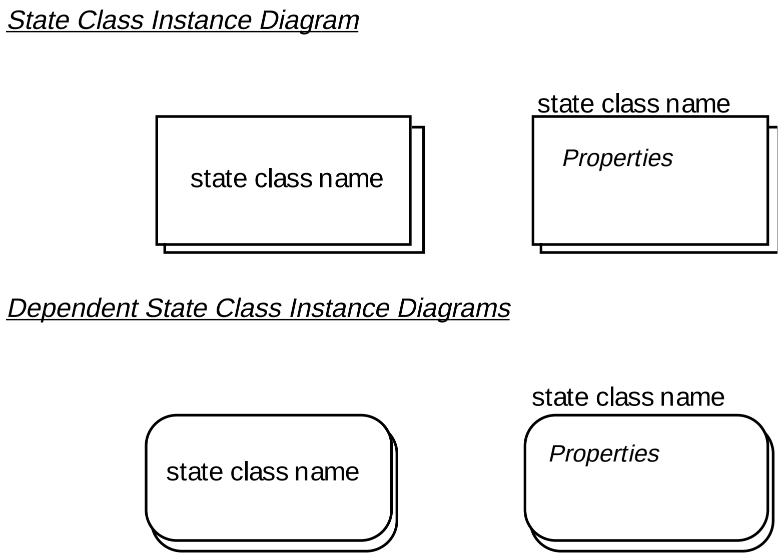


Figure 12—State class instance diagram alternative syntax

c) In a sample instance table, the instance identity label may be shown to the left of the row representing the instance.

d) In a sample instance table, each property shall be represented by a column where the column is named by the sample instance property label. Each cell shall display the sample instance property value that is associated with the row (instance).

e) When the instance table displays class instances as its rows, a double line shall separate the property name column headings from the first instance row.

f) When the instance table displays class-level properties, a single line shall separate the property name column headings from the class property row, and the values of class-level properties shall be shown in this single row without an oid.

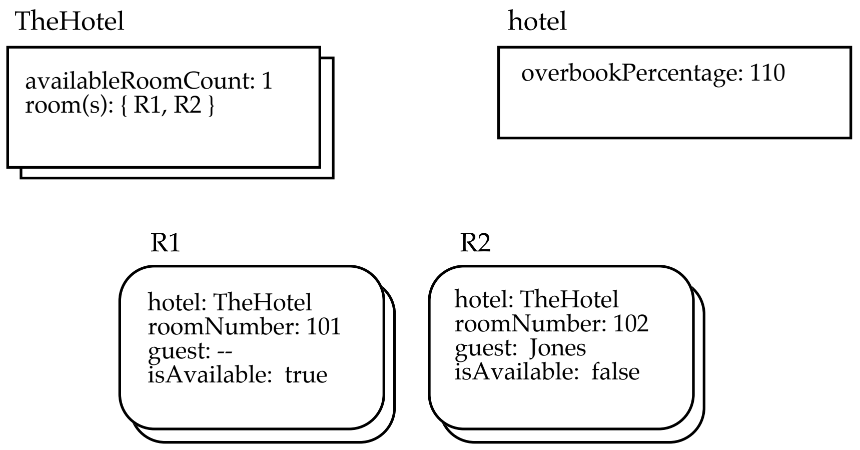


Figure 13—Sample state class instance diagram

hotel

	availableRoomCount	room(s)
TheHotel	1	{ R1, R2 }

hotel

overbookPercentage
110

room

	hotel	roomNumber	guest	isAvailable
R1	TheHotel	101	--	true
R2	TheHotel	102	Jones	false

Figure 14—State class sample instance tables

5.2.3 State class rules

There are no rules for *state class*, beyond those that apply to *class* in general.

5.3 Value class

A *value class* is a kind of class representing instances that are pure values; there are no duplicate values, and it makes no sense to update a value. The instances of a value class do not come and go and cannot change state, i.e., a value class is an *immutable* class. A value class has a fixed, and possibly infinite, set of instances. By contrast, a state class is a time-varying (*mutable*) class; the instances of a state class vary over time as the data is modified and maintained.

As instances of an immutable class, value class instances always exist in principle. For example, in the value class `date`, all instances of `date` exist, although some particular value of `date` might not be mapped to by an instance of any state class. Another example of a value class is `temperature`; the set of allowable values for this value class would satisfy the definition of “a degree of heat or cold.”

5.3.1 Value class semantics

5.3.1.1 Instance identity

The identity of an instance within the value class is equivalent to its value. More precisely, the identity of an instance is equivalent to the abstract value of the instance. The abstract value is represented by the values of hidden, encapsulated attributes. These values need be unique only within the class. Globally, an instance of a value class can be identified by the combination of its class and its representation. In terms of a representation system (i.e., examples, formalization, or software), the combination of the class and its representation uniquely identifies at most one instance of the class.

A value class uses one or more other value classes as its hidden, encapsulated representation of an instance. The `temperature` class may use a real number as its hidden representation, known only to the `temperature` class to be a Kelvin temperature. The `date` class may use an integer as its hidden representation, known only to the `date` class to be the number of days from a time zero (also known only to the `date` class). The representation hierarchy eventually terminates at a few predefined value classes, such as `real`, `integer`, `string`, `boolean`, etc.

5.3.1.2 Literal

In IDEF1X, a *literal* denotes a specific instance of a value class. A primitive (base) value class such as `integer` has a standard form of literal symbol (e.g., `17` or `3`) that is readily recognized and understood.

5.3.1.3 Associative literal

For a user-defined value class, there must be a provision for referring to an instance. An *associative literal* denotes an instance in terms of its value. The form of expression used to state an associative literal is either

```
  className with propertyName: propertyValue
```

or

```
  className(propertyName(propertyValue))
```

where `propertyName` is the sole constituent of a uniqueness constraint. In other words, no two instances of the class are permitted to have the same value for the named property.

The meaning of an associative literal is that the instance being referenced is the instance that has the value for the named property. For example,

```
  temperature with fahrenheit: 68
```

denotes a specific instance of the value class `temp`, and

```
  temperature with celsius: 20
```

denotes the same instance. They are both ways of denoting an instance (and the <u>same</u> instance). In fact, even though `7` by itself is a readily understood literal, it is simply a shorthand way to say

```
  integer with arabic: 7
```

which is equivalent to

```
integer with base2: 111.
```

A literal may also be stated using a literal expression, in the form of either

```
className with ( propertyName1: propertyValue1, ...propertyNameN:
   propertyValueN )
```

or

```
className(propertyName1(propertyValue1), ...
propertyNameN(propertyValueN))
```

where `propertyName1` through `propertyNameN` are the constituents of a *uniqueness constraint*. In other words, no two instances of the class are permitted to have the same values for the properties. For example,

```
vector with ( x: 100, z: 17 )
```

denotes a specific instance of the value class `vector`, and

```
date with ( month: 1, day: 10, year: 1995 )
```

denotes a specific instance of the value class `date`.[22]

5.3.1.4 Atomic/complex

A value class may represent either atomic data or complex data.[23] *Atomic data* is an indivisible whole and contains no constituents. Atomic data include things like `bit`, `integer`, `real`, or `character`. An atomic value, like `17`, entails no additional data and represents itself.

Complex data is data that contains data where

a) The constituent data is atomic or complex, and
b) Both the data as a whole and its constituents are accessed and operated on.

In this respect, alternative representations are considered constituent properties.

Complex data include things like `temperature`, `vector`, `time`, and `timeSpaceState`.

— "`temperature`" is complex because it must be represented by other data—Fahrenheit, Celsius, or Kelvin temperatures. Any one of them can be used to represent the abstract temperature. Whichever is used, the other two can be derived and all three are considered constituents of temperature. Any data in which the unit of measure is abstracted away is complex data that includes among its constituents its value in each unit of measure. One is used as representation; the others are derived.
— "`vector`" is complex because it includes, by definition, the constituents magnitude and direction. It may also contain the x,y coordinates. Either the magnitude and direction or the x,y coordinates can be used to represent the vector, and the others derived.
— "`time`" is complex because it has alternate units of measure.
— "`timeSpaceState`" is complex because it contains constituents, each of which is complex.

[22]Associative literals may be used for a state class as well. For a state class, any property may be used in the associative literal.

[23]There are many variations of complex data. Value classes provide a good solution for certain kinds of complex data. State classes provide a good solution for other kinds of complex data, such as bill-of-materials data or engineering design data.

Whether a given value class is atomic or complex depends on the context. An integer would usually be considered atomic. But in a model of the arithmetic unit of a computer, an integer might be considered complex (consisting of bits) in order to specify addition in terms of bit operations.

Both atomic and complex value classes are immutable in the sense that "changing" a value is logically impossible—that would make it a different value. Just as it makes no sense to change `17`, it makes no sense to change the `vector with ( x: 100, y: 17 )`; if any coordinate value is changed, the result is a different vector. In other words, the combination of x,y coordinate values is unique to a single vector. The combination of magnitude and direction is also unique to a single vector.

Each kind of value class can have operations that apply to it. Operations on complex data are carried out by operating on their constituent data. For example, the vector `add` operation can be carried out by adding the x,y coordinates of the two vectors.

5.3.1.5 Instance value constraint

A value class may have a declared value constraint. An *instance value constraint* specifies the acceptable values of a value class. Two examples of an instance value constraint are the value list constraint and the value range constraint.

a) The *value list constraint* specifies the set of all acceptable instance values for a value class. Attributes that represent a mapping into a value class with a value list constraint are only valid if their instance values are a part of the value list. A common use of this constraint is to enumerate a list of coded values such as `dayOfWeek`, i.e., `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, `Sunday`.
b) The *value range constraint* specifies the set of all acceptable instance values for a value class where the instance values are constrained by a lower and/or upper boundary. An example of the value range constraint is `Azimuth`, which is required to be between `-180`º to `+180`º. Another example is `temperature`, which must be above absolute zero. A range constraint only makes sense if there is a linear ordering specified.[24]

The instance value constraint for a value class is optional; it may be left unspecified. In this case, the value class is only constrained by the constraints associated with its data type or those of its superclass, if the value class has either (see 5.4). "`title`" is an example of a value class without an instance value constraint; it can take on any allowable character string.

5.3.2 Value class syntax

5.3.2.1 Graphic

a) A value class shall be represented as a rounded rectangle with a double base line, as illustrated in Figure 15.

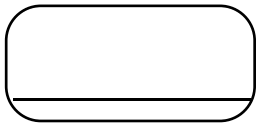


Figure 15—Value class syntax

[24]The typical *value range constraint* orders numbers and strings, but there can be other things (e.g., color) where some form of range constraint might be desired. However, if there is no ordering of the values, no "range" can be expressed. Therefore, to specify a range constraint, the notion of ordering, achieved by a `less than` (or `greater than`) operation in the value class, is required.

5.3.2.2 Label

a) As shown in Figure 16, the value class label shall be placed either
   1) Above or inside the rectangle when no names (responsibilities, property names, or constraint names) are shown, or
   2) Above the rectangle when names (responsibilities, property names, or constraint names) are shown inside the rectangle.

b) When placed outside the box, the value class label shall be left-justified and aligned with the left side of the box.

c) When placed inside the box, the value class label shall be centered inside the box.

d) Responsibilities, property names, and constraint names placed inside the value class box shall be left-justified.

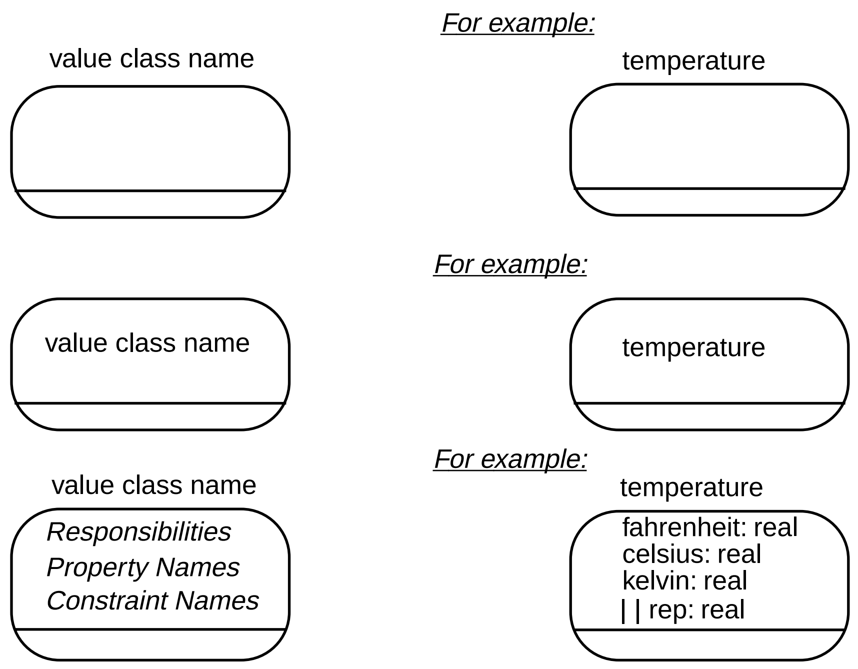


Figure 16—Alternatives for value class labeling

5.3.2.3 Sample instance identity label

In providing illustrative examples of instances, it is useful to have a way of representing an instance's identity.

a) The identity of a value class instance may be represented by a *literal*.
   The following are example literals for the built-in value classes `integer`, `real`, and `string` (respectively):

```
7
3.142
'Hello World'
```

   The following are example associative literals for the user-defined value classes `temperature`, `date`, and `point` (respectively):

```
temperature with celsius: 0
date with ( month: 1, day: 10, year: 1995 )
point with ( x: 100, y: 0 )
```

b) The identity of an instance may be represented by a *variable*, denoting an individual instance that is unknown at the time of writing. A variable name shall begin with an upper-case letter, such as:

```
Y
T1
Self
```

c) Along with the identity of the value class instance, the value class name may be stated. In this form, the class name shall precede the identity label literal or variable and shall be followed by a colon, such as:

```
temperature: T1
```

d) An unnamed, unknown instance shall be indicated by omitting the identity portion and including only the value class name of the instance, such as:

```
temperature
```

5.3.2.4 Sample instance property

a) Any relevant property of the instance, either direct or inherited, may be shown for an example instance.
b) The sample instance property shall have two parts:
   1) A sample instance property label, and
   2) A sample instance property value.
c) While the sample instance property label is typically the property name, the sample instance property label may be any RCL expression over properties of that class.
d) The sample instance property value shall be the value to which the expression evaluates for the instance. If the expression value pair is `E: V`, then `Self has E: V` shall hold, where `Self` is the identity of the instance.

5.3.2.5 Sample instance diagram

In a sample instance diagram, instances are shown as separate shapes.

a) In a sample instance diagram, an instance shall be represented by an open-stacked rounded rectangle with a double baseline.
b) In a sample instance diagram, the instance's identity label shall be placed either
   1) Inside the rectangle (when no responsibilities, property names, or property name value pairs are shown), as shown in Figure 17, or

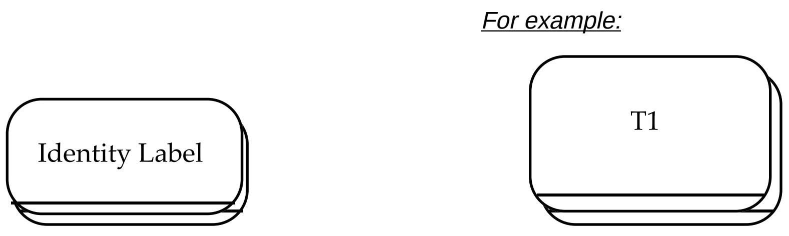


Figure 17—Value class sample instance diagram (1)

   2) Above the rectangle (when sample instance properties are shown inside the rectangle), as shown in Figure 18.
c) In the form of sample instance diagram shown in Figure 18, a sample instance property shall be written as:

```
property name: property value
```

d) An alternative form of syntax shall be available for a value class instance diagram. This form shall use the name of the value class along with its identity, as illustrated in Figures 19 and 20.
e) The values of class-level properties shall be shown in an unstacked box that is labeled by only the class name, as shown in Figure 21.

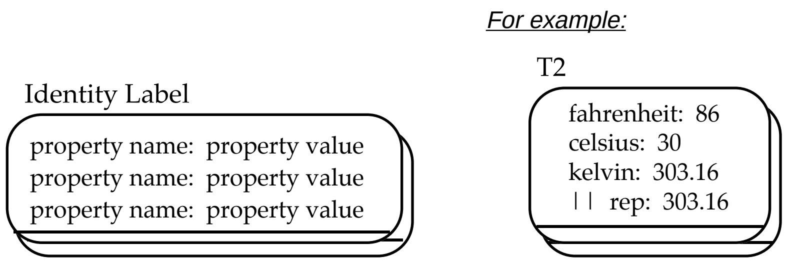

Identity Label
property name: property value
property name: property value
property name: property value
For example:
T2
fahrenheit: 86
celsius: 30
kelvin: 303.16
| | rep: 303.16

Figure 18—Value class sample instance diagram (2)

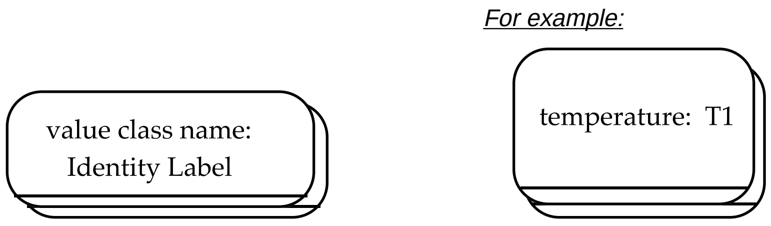


Figure 19—Alternative syntax for a value class instance diagram (1)

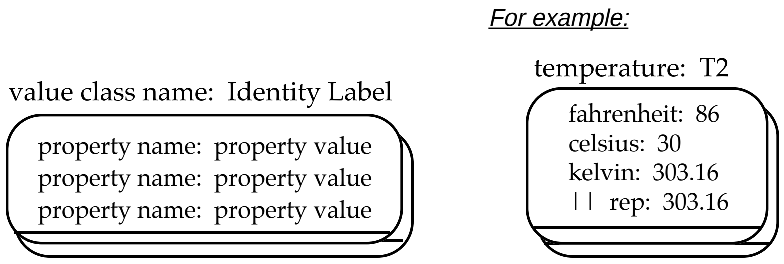

value class name: Identity Label
property name: property value
property name: property value
property name: property value
For example:
temperature: T2
fahrenheit: 86
celsius: 30
kelvin: 303.16
| | rep: 303.16

Figure 20—Alternative syntax for a value class instance diagram (2)

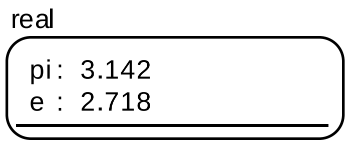


Figure 21—Value class class-level property instance diagram

5.3.2.6 Sample instance table

A second form of showing sample value class instances is as a sample instance table. This tabular presentation of instances is useful when several instances of one class are to be presented. The conventions for a sample instance table are illustrated in Figure 22, which depicts two instances of `temperature` and two class-level properties values of `real`.

a) In a sample instance table, the class name shall be placed above the table.

b) In a sample instance table, instances shall be shown as rows in a table representing the class.

temperature

	fahrenheit	celsius	kelvin	|| rep
T1	68	20	293.1	293.16
T2	86	30	303.1	303.16

real

pi	e
3.142	2.718

Figure 22—Value class sample instance table

c) In a sample instance table, the instance identity label may be shown to the left of the row representing the instance.

d) In a sample instance table, each property shall be represented by a column where the column is named by the sample instance property label. Each cell shall display the sample instance property value associated with the row (instance).

e) When the instance table displays class instances as its rows, a double line shall separate the property name column headings from the first instance row.

f) When the instance table displays class-level properties, a single line shall separate the property name column headings from the class-level property row, and the values of class-level properties shall be shown in this single row without an identity label.

5.3.3 Value class rules

5.3.3.1 Instance value constraint

a) Any number of instance value constraints may be specified for a value class.

5.3.3.2 Uniqueness constraint

a) At least one uniqueness constraint shall be specified for a value class.

b) The property name(s) stated in an associative literal shall be the sole constituent(s) of a uniqueness constraint.

5.4 Generalization

People mentally abstract a *generalization* between two classes when they realize that every instance of one class is an instance of another class. Everyone does this generalizing; it is part of common sense. In this way the generalization of citizen over registered-voter is abstracted from the realization that every registered voter is a citizen.

Classes are used to represent the notion of “things whose knowledge or behaviors are relevant.” Since some real world things are generalizations of other real world things, some classes must, in some sense, be generalizations of other classes. A class that specifies additional, different responsibilities to those of a more general class is known as a *subclass* of that more general class (its *superclass*). Each instance of the subclass represents the same real-world thing as its instance in the superclass.

For example, suppose employees are something with knowledge and behavior. Although there is some information needed about all employees, for salaried employees additional responsibilities might be needed that differ from the additional responsibilities needed for hourly employees. In this example, the classes `salariedEmployee` and `hourlyEmployee` are considered subclasses of the class `employee`. In another case, a subclass may be needed to express a relationship that is valid for only a specific subclass or to document the relationship differences among the various subclasses. For example, a `fullTimeEmployee` may qualify for a `benefit`, while a `partTimeEmployee` may not. In a third case, a subclass may be needed to reflect different behavior. For example, a `partTimeEmployee` may change days or hours of employment while a `fullTimeEmployee` may not.

5.4.1 Generalization semantics

5.4.1.1 Instance

Saying that a subclass `S` generalizes to a superclass `C` means that every instance of class `S` is also simultaneously an instance of class `C`. For example, every instance of `fullTimeEmployee` is also an instance of `employee`, and every instance of `partTimeEmployee` is an instance of `employee`. Note that there is only one instance; this concept is fundamentally different from a *relationship*, which associates distinct instances.

5.4.1.2 Generalization structure

A *generalization structure* is a connection between a superclass and one of its more specific, immediate subclasses. A *generic ancestor* of a class is a superclass that is either an immediate superclass of the class or a generic ancestor of one of the superclasses of the class.

A generalization structure is not explicitly named. It is important to remember that a generalization structure is not a relationship. It cannot be said that a subclass instance "is related to" its intrinsic superclass since an instance of the subclass and its superclass are one and the same instance. However, in reading the generalization structure, implicit verb phrases may be used:

— *"is a/an"* (from the subclass to the superclass), and
— "*can be a/an*" or "*must be a/an*" (from the superclass to the subclass).

For example, reading the generalization structure in Figure 25 from the subclass to the superclass direction, *"each salariedEmployee is an employee.*" The generalization structure is read as *"each employee can be a salariedEmployee.*" in the reverse direction. If each instance of the superclass must be an instance of one of its subclasses, the structure should be read as "*must be a/an*" with an *"or a/an"* conjunction. For example, *"each employee must be a fullTimeEmployee or a partTimeEmployee.*"

5.4.1.3 Generalization taxonomy

The set of generalization structures with a common generic ancestor forms a *generalization taxonomy* (or *generalization hierarchy*). In a generalization taxonomy every instance is fully described by one or more of the classes in the taxonomy. For every instance, at least one of these classes is its *lowclass*, the lowest subclass in its declaration. Specifically, if an instance is in a class `S` and not in any subclass of `S`, then `S` is the lowclass for the instance. *Lowclass* is important in understanding property inheritance and request response handling.[25]

5.4.1.4 Substitutability

Since each instance of a subclass is an instance of the superclass, an instance of the subclass should be acceptable in any context where an instance of the superclass is acceptable. This is the principle of *substitutability* [B19]. When substitutability holds throughout a model, reasoning about the model is simplified since it can be done on the basis of what a property value means for the superclass in which it is specified. If substitutability did not hold, then reasoning about a model would require examination of every overriding property. For instance, referring to the example in Figure 23, an instance of `assembledPart` can be reasoned

[25]See 5.4.16 for an explanation of the circumstances where there can be multiple lowclasses for an instance. See 5.4.1.11 for a discussion of the possible impacts on lowclass when specializations change.

about simply as a `part`. This more generalized notion of `part` has a property `cost` that does not lose any meaning in its more specialized form as `assembledPart`'s `cost`.[26]

5.4.1.5 Inheritance

Generalization implies inheritance of responsibilities.[27] This is an old idea with new terminology. For example,

> "*Every mammal has a date of birth.*
> *Every human is a mammal.*
> *Therefore every human has a date of birth.*"

is one of the syllogisms of Aristotle's logic. The class `human` inherits the date of birth property from the superclass `mammal`.

Because every instance of a subclass is also an instance of its superclass, each instance of the subclass has the responsibilities (properties and constraints) of its superclass as well as its own. The subclass is said to *inherit* the responsibilities of its superclass. The subclass may also declare responsibilities that are specific to that subclass. Finally, the subclass may specify properties that have different realizations from the realizations specified in its superclass. Figure 23 illustrates these three aspects of inheritance.

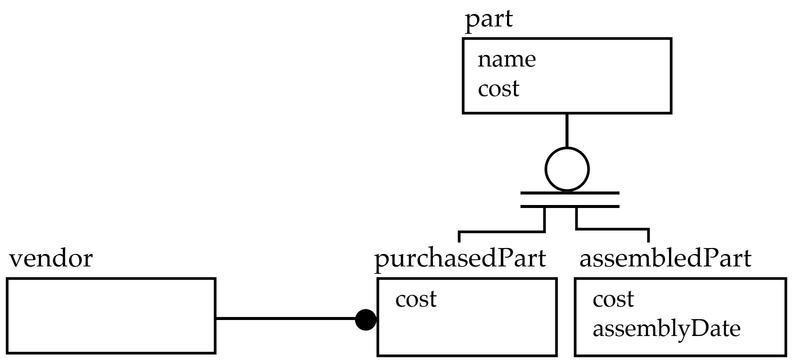


Figure 23—Inheritance

In the view in Figure 23, a part is categorized according to whether it is an assembled or a purchased part. Every part has a name, so this property is stated at the level of the superclass, `part`. The specification of each subclass (`purchasedPart` and `assembledPart`) inherits this property from its superclass. Every instance of `purchasedPart` and `assembledPart` will have a name value because each is itself a `part`.

Furthermore, a subclass may declare additional responsibilities that are specific to the subclass. An `assembledPart` has a date on which it was assembled, and a `purchasedPart` is associated with the vendor that supplies it.

Finally, while generalization implies the inheritance of the specification and meaning of a property (i.e., its *interface*), it does not necessarily imply the inheritance of its *realization* (see Clause 6). When both the subclass and its superclass have a property of the same name, the property in the subclass *overrides* the inherited property (Figure 24). Overriding is intended to preserve substitutability. Continuing with the example in Figure 23, all parts have a cost, but an assembled part has a cost that depends upon the cost of each of its constit-

[26] For a further discussion of typing rules for overrides, see 7.4.4.
[27] See Clause 7 for a full discussion of inheritance.

uents plus the cost of assembly. While a purchased part knows its cost directly, an assembled part must calculate its cost when requested. The superclass `part` has a property called `cost` that is inherited by both `purchasedPart` and `assembledPart`. The realization of `cost` (i.e., the method that implements the operation) differs in each of the subclasses, but its meaning (interface) does not. Generalization and its consequent inheritance are a semantic notion, not an aspect of the realization.

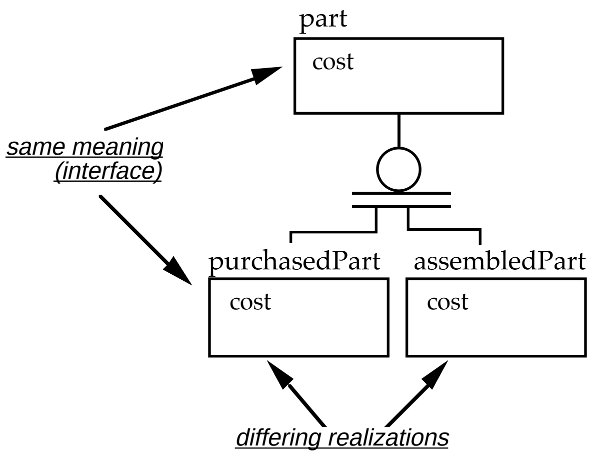


Figure 24—Inheritance overriding

5.4.1.6 Subclass cluster

A *subclass cluster* (simply, *cluster*) is a set of one or more generalization structures in which the subclasses share the same superclass and in which an instance of the superclass is an instance of no more than one subclass. A cluster exists when an instance of the superclass can be an instance of only one of the subclasses in the set, and each instance of a subclass is an instance of the superclass. Since an instance of the superclass cannot be an instance of more than one of the subclasses in the cluster, the subclasses in a cluster are mutually exclusive. However, a class can be the superclass in more than one cluster, and the subclasses in one cluster are not mutually exclusive with those in other clusters.

Expanding on the earlier example of employee, `salariedEmployee`, `hourlyEmployee`, `fullTimeEmployee`, and `partTimeEmployee` are all subclasses of the superclass `employee`. These are four generalization structures: one between `employee` and `salariedEmployee`, a second one between `employee` and `hourlyEmployee`, a third between `employee` and `fullTimeEmployee`, and a fourth between `employee` and `partTimeEmployee`.

In this example, an employee cannot be both salaried and hourly. Likewise, an employee cannot be both full-time and part-time. However, an employee could be hourly and full-time, or hourly and part-time, etc. Thus, there are two clusters specified for `employee`—one including `salariedEmployee` and `hourlyEmployee`, and one including `fullTimeEmployee` and `partTimeEmployee`. An instance of `employee` can be simultaneously an instance of either `salariedEmployee` or `hourlyEmployee` and an instance of either `fullTimeEmployee` or `partTimeEmployee` (see Figure 25).

5.4.1.7 Total cluster/partial cluster

If each instance of a superclass must be an instance of at least one of the subclasses of a cluster, the cluster is said to be a *total cluster* (*complete cluster*). In a total cluster, each superclass instance is always an instance of one of its subclasses. For example, in Figure 25 each employee is either full-time or part-time, so that cluster is total.

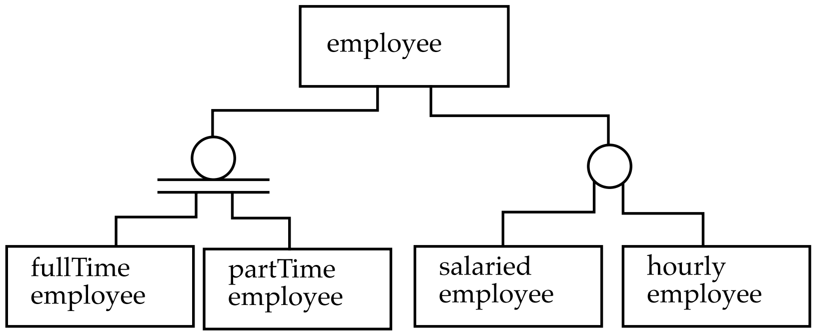


Figure 25—Multiple subclass clusters

In a *partial cluster* (*incomplete cluster*), an instance of the superclass may exist without also being an instance of any of the subclasses. For example, if some employees are unpaid and, therefore, have none of the additional properties of salaried or hourly employees, that cluster is partial.

5.4.1.8 Abstract class

A class for which every instance must also be an instance of a subclass in the cluster (i.e., a total cluster) is called *abstract* with respect to that cluster. A class is an *abstract class* if it is abstract with respect to any cluster.

An abstract class cannot be instantiated independently, i.e., instantiation must be accomplished via a subclass. In Figure 25, `employee` is abstract with respect to the two subclasses, `fullTimeEmployee` and `part-TimeEmployee`; every `employee` is either a `fullTimeEmployee` or a `partTimeEmployee`. However, because the salaried/hourly cluster is partial, `employee` is not abstract with respect to that cluster.

5.4.1.9 Parallel classes

Two subclasses are *parallel classes* if they are distinct, are not mutually exclusive, and have a common generic ancestor and for which neither is a generic ancestor of the other. Figure 26 illustrates these ideas:

a) There are two clusters under `c1`: `c2` and `c3`.
b) `c1` is abstract with respect to the cluster `c2`, but not with respect to `c3`.
c) The generic ancestors of `c4` are `c2`, `c3`, and `c1`.
d) The parallel class pairs are (`c2, c3`) and (`c2, c5`).

5.4.1.10 Inheritance disambiguation

While the kind of construct shown in Figure 26 rarely occurs, it is used here to illustrate the disambiguation of inheritance conflicts. If any class has multiple superclasses, inheritance conflicts could occur. Inheritance conflicts arise when a class inherits a responsibility of a given name from two distinct generic ancestors. Such conflicts shall be avoided by imposing the rule under “Uniqueness” in Clause 7.5.3.

5.4.1.11 Changing state class specialization

Every instance is an instance of one or more classes. The instance’s lowclass in a cluster, if any, is the lowest class in that cluster of which it is an instance. In a generalization taxonomy, it is possible for a state class instance to change the nature of its specialization.[28] Specifically, an instance whose lowclass in a cluster is at

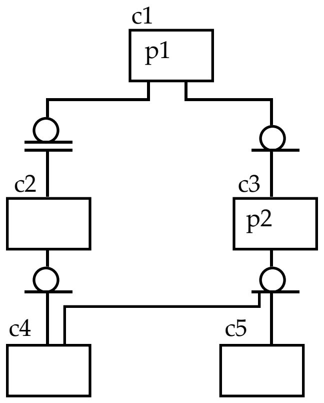


Figure 26—Parallel classes and inheritance disambiguation

one level in the taxonomy could become (or be discovered to be) a more specialized form of that class. In this case, the instance would *specialize*, i.e., become an instance of one (or more) of the subclasses specified for the class of its current lowclass and thereby have a different (lower) lowclass. For example (referring to Figure 27), an instance `AA` with a current lowclass of `b1` could specialize as `c`; its new lowclass would be `c`. This is not restricted to only one level; an instance `AB` with a current lowclass of `a` could specialize as `c`.

Alternatively, a state class instance could become (or be discovered to be) a less specialized form of its class. In this case, the instance would *unspecialize*, i.e., cease being an instance of one (or more) of the subclasses specified for the class of its current lowclass and thereby have a different (higher) lowclass. For example, an instance `AC` with a current lowclass of `c` could unspecialize as `b1`; its new lowclass would be `b1`. This is not restricted to only one level; an instance `AD` with a current lowclass of `c` could unspecialize as `a`.

It is also possible for a state class instance to change "laterally" within a cluster. In this case, the instance would *respecialize*, i.e., become an instance of one of the other subclasses in its current cluster. For example, an instance `AE` that is currently in subclass `b1` could respecialize as `b2`. The lowclass of `AE` could have been either `b1` or `c`; in either case its new lowclass would now be `b2`. Conversely, if instance `AF` might respecialize from `b2` to `b1`; its new lowclass would be `b1`. It could further specialize as `c`.

5.4.1.12 Discriminator

A *discriminator*, which is a property of the superclass, may optionally be specified for a cluster. Since the value of the discriminator (when a discriminator has been declared) is equivalent to the identity of the subclass to which the instance belongs, there is no requirement for a discriminator.

If a discriminator is identified, the value of the discriminator determines the subclass of an instance of the superclass. For example, in Figure 25 shown earlier, the discriminator for the cluster including the full- and part-time subclasses might be an attribute named `employeeTimeType` (see Figure 30).

In a total cluster using a discriminator, there will always be a value of the discriminator. In a partial cluster, an instance's discriminator has no value if the class is the instance's *lowclass* within the cluster.

5.4.1.13 Value class hierarchy

While value classes may exist in a generalization hierarchy, it should be emphasized that representation is not the same as generalization. For example in Figure 16, the `temperature` value class has a representa-

[28]The built-ins supporting these operations are described in Annex D.

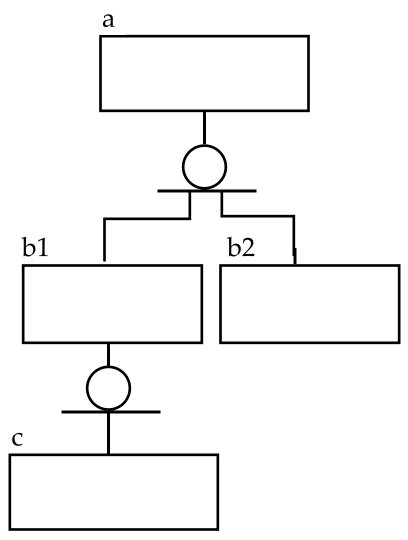


Figure 27—Changing specialization

tion of `real`, but it is not specified as a subclass of `real`. A given representation value may occur in many value subclass instances, but that does not make the instances identical.

5.4.2 Generalization syntax

5.4.2.1 Subclass cluster

a) The subclass cluster symbol shall be an underlined circle.
b) A cluster shall be shown as a line extending from the superclass to the subclass cluster symbol accompanied by separate lines extending from the bottom-most subclass cluster underline to each subclass in the cluster, as shown in Figure 28.

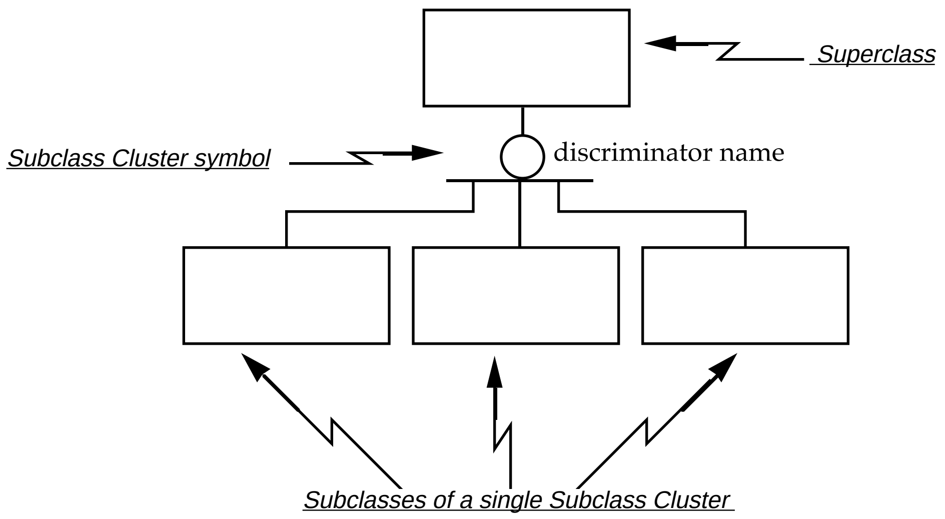


Figure 28—Subclass cluster syntax

5.4.2.2 Abstract class

a) A double-underlined circle, as shown in Figure 29, shall designate that the superclass is an abstract class with respect to the cluster, i.e., the cluster is a *total cluster*. Note that this shall not mean that all of the subclasses are depicted in the diagram.

b) A single-underlined circle shall denote a *partial cluster*, i.e., the superclass is not abstract with respect to the cluster.

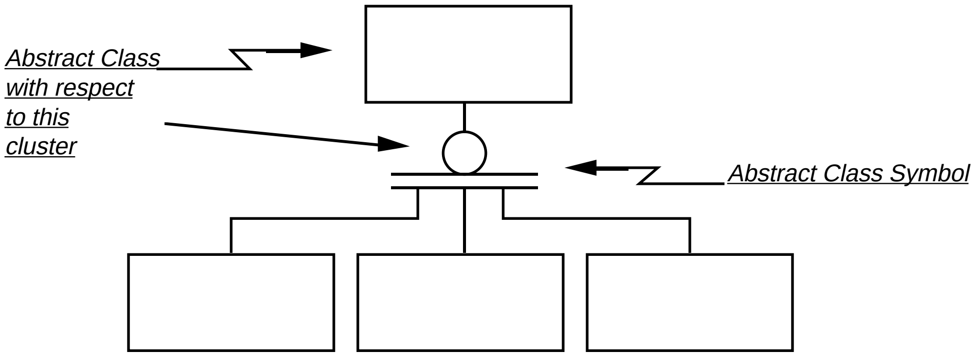


Figure 29—Abstract class syntax

5.4.2.3 Generalization structure

a) In a diagram, each line pair (from the superclass to the subclass cluster symbol, and from the bottom-most underline of the subclass cluster symbol to the subclass) shall represent one of the generalization structures in a cluster.

5.4.2.4 Discriminator

a) If a discriminator property has been specified, that property's name shall be written with the subclass cluster symbol, as shown in Figure 30.

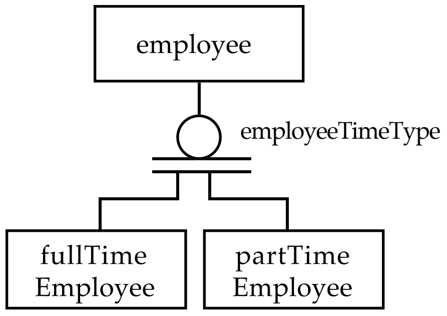


Figure 30—State class discriminator

5.4.3 Generalization rules

5.4.3.1 Generalization structure

a) A class may have more than one generalization structure in which it is the subclass, i.e., a class may have more than one superclass.
b) A subclass in one generalization structure may be a superclass in another generalization structure.
c) The subclass and superclass in a generalization structure shall both be state classes or both be value classes.
d) A state class subclass shall inherit the nature of its superclass, i.e.,
   1) If a superclass is a dependent state class, its subclasses shall be dependent.
   2) If a superclass is an independent state class, its subclasses shall be independent.

5.4.3.2 Generalization taxonomy

a) All the superclasses of a class shall have a common generic ancestor.
b) No class may be its own generic ancestor, i.e., no class may have itself as a superclass nor may it participate in any series of generalization structures that specifies a cycle.

5.4.3.3 Inheritance overriding

a) A subclass shall *inherit* the responsibilities of its superclasses.
b) A subclass may have additional responsibilities beyond those of its superclasses.
c) A subclass may *override* one or more of the responsibilities of its superclasses.
d) A property `P′` of a class `C′` that overrides a property `P` of a superclass `C` may do so in one of two ways:
   1) As a substitution for `P`, or
   2) As a specialization of `P`.
      Whether `P′` is a *substitute* or *specialization* is a matter of intent. It shall be up to the modeler to choose whichever best models the “real world” under study. (See also 7.4.4.)
e) If `P′` *substitutes* for `P`, then `P′` shall be used for all messages to instances of `C′`.
f) If `P′` *specializes* `P`, then `P′` shall be used for some messages to instances of `C′` and `P` shall be used for other messages to instances of `C′`, depending on the (dynamic types of the) argument values in the message.

5.4.3.4 Subclass cluster

a) The subclasses in a cluster shall be mutually exclusive.
b) Subclasses in distinct clusters of a superclass need not be mutually exclusive.
c) A class may have any number of subclass clusters in which it is the superclass.
d) A subclass cluster shall be classified as either
   1) “total” (“complete” or “abstract”), or
   2) “partial” (“incomplete” or “concrete”).
e) A view may present all, or only some, of the subclasses of a class.

5.4.3.5 Discriminator

a) A discriminator shall be a property of the superclass.
b) If a discriminator is declared for a total cluster, the discriminator shall have a value for every instance.
c) The value of the discriminator property shall be either
   1) The intrinsic identifier of the superclass, or
   2) One-to-one mappable to that identifier.

d) If a cluster has a discriminator declared, the discriminator shall be distinct from all other discriminators for the superclass, i.e., no two clusters of a superclass may have the same discriminator.

5.4.3.6 Value class hierarchy

a) A subclass value class shall not be restricted to having the same representation as its superclass (since representation is by encapsulated attributes).
b) The instances of a subclass value class may be a subset of the instances of the superclass value class.[29]
c) A value class subclass may have additional or different properties in its representation.

   For example, there are two classic representations for `point`: Cartesian and polar. For certain operations on points, there are "better" (faster/more efficient) implementations, e.g., addition is better using Cartesian and multiplication is better using polar. This means that the representation in the subclass could be different from that in the superclass. Alternatively, the superclass could be declared abstract, with the representation stated only in the subclasses.
d) Parallel value classes shall be abstract.
e) Every pair of parallel value classes shall have a common subclass.

The result of these rules is that a value class instance shall always have exactly one lowclass, but a state class instance may have multiple lowclasses.

5.5 Relationship

People mentally abstract relationships between classes in the sense that individual instances of the classes are related in a similar way. Everyone does this relating; it is part of common sense. A *relationship* expresses a connection between two (not necessarily distinct) classes that is deemed relevant to a particular scope and purpose. It is named for the sense in which the instances are related. For example, a *"votes at"* relationship between the `registeredVoter` class and the `pollingPlace` class is abstracted from the understanding that individual instances of registered voters vote at a polling place.

5.5.1 Relationship semantics

5.5.1.1 Relationship/relationship instance

An IDEF1X diagram depicts the type of relationship between two state classes. A relationship is the result of mental classification, but a relationship itself is not treated as a state class or value class. An instance of the relationship associates specific instances of the related classes. It is a time-varying binary relation between the instances (in the current extents) of two state classes. For example, "customer Mary owns account number 123" could be an instance of the relationship shown in Figure 34.

5.5.1.2 Identity

A relationship instance does not have its own intrinsic identity; rather, its identity comes from the identity of the participating state classes. The relation can be visualized in the usual tabular way (two columns each reflecting an oid) as illustrated in Figure 31, which shows the instances of the `vendor/boughtPart` relationship of TcCo (see C.7). Each row is an ordered pair of oids for the related objects. An instance (occurrence) of a relationship is uniquely determined by the identity of the participants. A relationship instance does not have an identity independent of its property values; its identity is equivalent to its property values.

[29]Subclassing by subsetting preserves substitutability and allows static type checking only for value classes. See also "Fundamentals of Object-Oriented Databases" [B21].

standardVendor boughtPart

vendor	boughtPart
#201	#4
#101	#5
#301	#6
#301	#7

Figure 31—Relationship instances

Relationship instances can also be presented in a sample state class instance diagram. The same TcCo relationships that are shown in Figure 31 can be depicted in a diagram of the instances participating in this relationship (and omitting, for now, the display of the other properties of the classes), as in Figure 32.

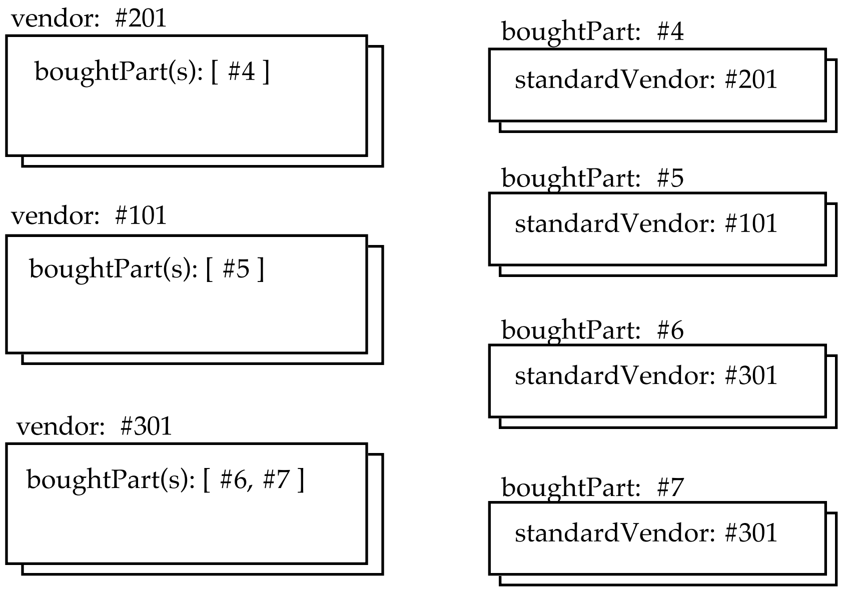


Figure 32—Alternative presentation of relationship instances

This presentation is meant simply to be a way to visualize the instances of a relationship type. One way to implement the relationship is to form a table (as in Figure 31); another way is for each participant to maintain the identity value (or a “list” of such values) of the other participants as reflected in Figure 32. This standard does not specify any particular way to achieve the implementation of relationships.

5.5.1.3 One-to-many relationship

A *one-to-many relationship* (sometimes referred to as a *parent-child relationship*) is a relationship between two state classes in which each instance of one class (referred to as the *child class*) is specifically constrained to relate to no more than one instance of a second class (referred to as the *parent class*). Each instance of the parent class may be associated with zero, one, or more instances of the child class. For example, a one-to-many relationship would exist between the classes `account` and `transaction` if each transaction is incurred by a single account and an account incurs zero, one, or more transactions.

The term “one-to-many” includes the special case of a *one-to-one relationship* in which each instance of the parent class is also specifically constrained to relate to no more than one instance of the child class. In this case, the terms “parent” and “child” lose their intuitive meaning.

5.5.1.4 Many-to-many relationship

A *many-to-many relationship* is a relationship between two state classes in which each instance of one class may be associated with any number of instances of a second class (possibly none), and each instance of the second class may be related to any number of instances of the first class (possibly none). Such a relationship would exist between the classes `account` and `customer` if each customer may own any number of accounts (zero, one, or more) and each account may be owned by any number of customers. In the initial development of any model, it is often helpful to identify many-to-many relationships between classes. Many-to-many relationships are often used in survey-level models (see 8.2) to represent general associations between state classes.

Many-to-many relationships may be replaced in later phases of model development.[30] For example, the many-to-many relationship between customer and account discussed above might be replaced by a pair of one-to-many relationships by introducing a third state class, such as `accountOwnership`, which is a common child class in parent-child relationships with the `customer` and `account` classes. These new relationships specify that an account has zero, one, or more account ownerships (constrained, in this example to one or more) and that each customer has zero, one, or many account ownerships. Each account ownership is for exactly one customer and exactly one account. A class introduced to resolve a many-to-many relationship is sometimes called an *associative class*. A many-to-many relationship is replaced with an associative class when the association is itself an object of interest, i.e., it has responsibilities of its own (perhaps including relationships to other classes).

5.5.1.5 Cardinality

A relationship specification includes a statement of the *cardinality* of the relationship. Cardinality specifies how many instances of the second class may or must exist for each instance of the first class, and how many instances of the first class may or must exist for each instance of the second class. For each direction of a relationship, the cardinality can be constrained to be at most one, at least one, or both.

The following cardinalities may be expressed from the perspective of each participating class:

a) Each instance of one class shall have exactly one associated instance of the other class.
b) Each instance of one class shall have no more than one (i.e., zero or one) associated instance of the other class.
c) Each instance of one class shall have at least one (i.e., one or more) associated instance of the other class.
d) Each instance of one class shall be associated with some exact number of instances of the other class.
e) A more specific cardinality shall be expressed using either a constraint or a note.

If no cardinality is specifically declared for the perspective of a participating class, the following cardinality applies by default: Each instance of one class shall have zero or more associated instances of the other class.

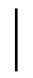
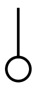
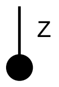
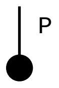
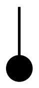
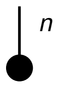
These cardinality variations can be summarized by stating that a relationship can be specified as

— *Single-valued* (i.e., at most one) or *multi-valued* (i.e., possibly more than one), and
— *Total* (i.e., at least one) or *partial* (i.e., possibly none)

[30] In models that are not identity-based, all relationships must eventually be expressed as one-to-many relationships (see 9.10).

in any combination, in both directions. With no constraints stated explicitly, a relationship is simply a many-to-many relationship (multi-valued) that is partial in each direction. The graphics for expressing relationships and cardinality are provided in Table 2.

Table 2—Relationship cardinality syntax

Cardinality	Graphic	Cardinality expression	Single-valued participant type	Multi-valued participant type
The absence of a dot shall indicate “exactly one.”		exactly one	scalar	NA
A hollow dot shall indicate zero or one.		at most one	scalar	NA
A `Z` beside a solid dot shall indicate a collection constrained to no more than one (and possibly none).		at most one[a]	collection-valued with `(s)` suffix	scalar
A `P` (for “positive”) beside a solid dot shall indicate one or more, i.e., at least one, possibly more.		one or more	collection-valued with `(s)` suffix	scalar
A solid dot shall indicate there is no cardinality constraint, i.e., zero, one, or more.		zero or more	collection-valued with `(s)` suffix	scalar
A positive nonzero integer beside the dot shall indicate a cardinality of an exact number.		exactly *n*	collection-valued with `(s)` suffix	scalar

[a]This alternate form for “at most one” cardinality is provided for modelers who wish to represent consistently all relationships as collection-valued, constrained appropriately. For this audience, it is possible to represent a collection-valued cardinality of “exactly one” using the last option in Table 2 using a value of `1`.

5.5.1.6 Deletion

When a state class instance is deleted (using the built-in deletion property[31]), the deletion propagates along relationships to the extent needed to comply with the cardinality constraints. For example, in Figure 34, the deletion of an account deletes all related transactions, and the deletion of a customer deletes any account owned only by that customer, which in turn deletes all related transactions. On the other hand, the deletion of a transaction does not propagate.

The default deletion property is built around cardinality constraints. It offers the following advantages:[32]

a) Elimination of race conditions[33] (and all associated complexity)
b) Direct use of the cardinality constraints
c) Provision for exception handling via overrides

[31]See 10.2.

[32]However, it has the disadvantage that, if all that is needed is a simple restrict, it is necessary to override the default (built-in) deletion property with a user-written deletion property.

[33]A *race condition* exists when the outcome depends on the order of execution and the order of execution is not specified, thus making the outcome unspecified.

5.5.1.7 Verb phrase

A relationship may be labeled with a *verb phrase* such that a sentence can be formed by combining the name of the first class, the verb phrase, the cardinality expression, and the name of the second class. A verb phrase is ideally stated in active voice. For example, the statement “each project funds one or more tasks” could be derived from a relationship showing `project` as the first class, `task` as the second class with a “one or more” cardinality, and “funds” as the verb phrase.

A relationship may be labeled with up to two verb phrases, one for each “direction” of the relationship. A second verb phrase is sometimes a simple restatement of the first verb phrase in passive voice, e.g., “Customer <u>owns</u> account.” restated as “Account <u>is owned by</u> customer.” In fact, from the previous example, it could be inferred that “each task <u>is funded by</u> exactly one project.” without the direct statement of a verb phrase. The second direction is simply represented as “is funded by” as the passive voice form of the “funds” verb phrase. A relationship shall still hold true in both directions even if no verb phrase is explicitly assigned.

5.5.1.8 Role name

Two additional labels can be designated for each relationship. These labels are the relationship’s role names (see also 6.5). A relationship *role name* is a name given to a class in a relationship to clarify the participation of that class in the relationship, i.e., connote the role played by a related instance. This naming scheme results in up to four labels for each relationship, a role name and a verb phrase in each direction.

An example that includes two one-to-many relationships with all four role names is shown in Figure 33. This example has the following full reading, with the role names highlighted in bold and the verb phrases underlined:

a) Each **`origin`** <u>is the start of</u> many (zero or more) **`outboundFlight(s)`**.
b) Each **`outboundFlight`** <u>takes passengers from</u> exactly one **`origin`**.
c) Each **`destination`** <u>is the end of</u> many (zero or more) **`inboundFlight(s)`**.
d) Each **`inboundFlight`** <u>brings passengers to</u> exactly one **`destination`**.

If a class in a relationship has a role name `r`, the name of the corresponding participant property in the related class is either `r` or `r(s)` depending on the cardinality of the relationship (see 6.5).

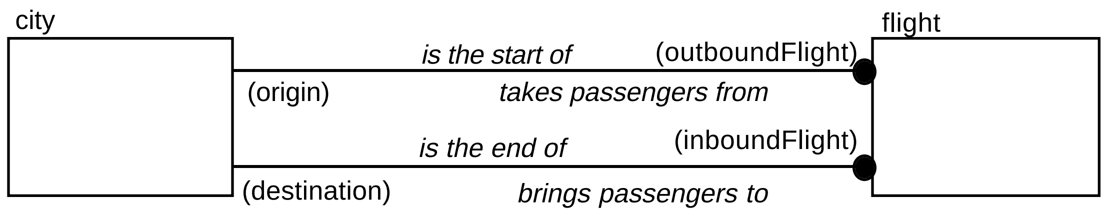


Figure 33—Relationship with verb phrases and role names

5.5.1.9 Intrinsic relationship

A relationship is an *intrinsic relationship* if it is *total* (i.e., not partial), *single-valued* (i.e., not multi-valued), and *constant* (i.e., unchanging once established) from the perspective of (at least) one of the participating classes, referred to as a *dependent class* (see 5.2). Such a relationship is considered to be an integral part of the essence of the dependent class. For example, in Figure 34 a `transaction` has an intrinsic relationship to its related `account` because it makes no sense for an instance of a `transaction` to switch to a different `account`. That would change the very nature of the `transaction`.

A dependent state class may also participate in one or more nonintrinsic relationships. A *nonintrinsic relationship* is a relationship that, from the dependent class perspective, is partial, is multi-valued, or may change. For example, a `transaction` can also be related to an initiating `location`; this relationship is shown as nonintrinsic in Figure 34.

5.5.2 Relationship syntax

5.5.2.1 Graphic

a) A relationship shall be depicted as an arc (line) connecting the associated state class rectangles.

5.5.2.2 Cardinality

a) As illustrated in Table 2, a “dot” (possibly annotated) at one end of the relationship line, or the absence of a dot, shall depict the cardinality from the perspective of the class at the other end of the relationship. The entries under “single-valued participant type” and “multi-valued participant type” are explained further in 6.5. Other cardinalities may be expressed using a constraint or a note reference annotated beside the dot, e.g., “from 2-12,” “more than 3,” “exactly 7 or 9,” etc.

5.5.2.3 Nonintrinsic relationship

a) A dashed line shall be used for the relationship arc of any nonintrinsic relationship of a dependent class.

b) A solid line shall be used for all kinds of relationship arc other than the nonintrinsic relationship of a dependent class.

In Figure 34, the dependent state class, `transaction`, has an intrinsic relationship with `account`; this relationship arc is drawn as a solid line. “`transaction`” is also related to an initiating `location`. Since this relationship is not intrinsic, the relationship arc is drawn as a dashed line.

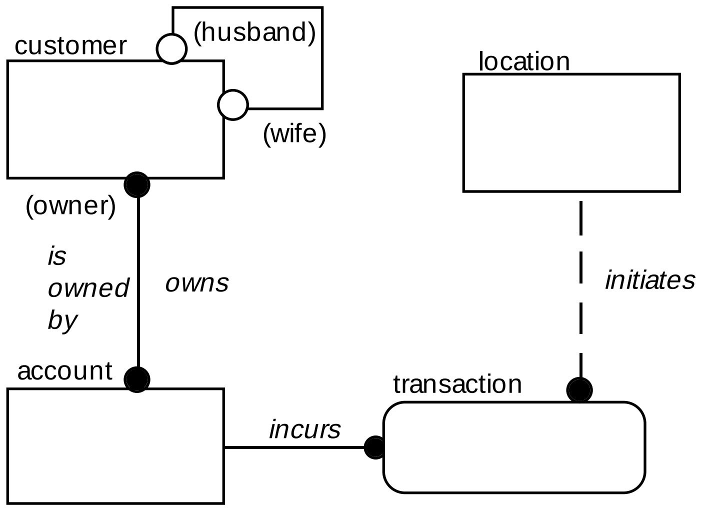


Figure 34—Relationships

5.5.2.4 Label

a) When a verb phrase is provided, it shall be written beside the relationship line on whichever side allows the diagram to be read in a clockwise direction. For example, in Figure 34, the relationship from customer to account reads: “Each customer owns accounts.”
b) When a role name is specified, it shall be written in parentheses next to the class whose instance it names, i.e., at the opposite end of the relationship from the class in which the participant property occurs (see also 6.5). For example, in Figure 34, the relationship between `account` and `customer` is read as “Each account is owned by owners.” Figures 33 and 34 illustrate the proper placement of verb phrases and role names.

5.5.3 Relationship rules

5.5.3.1 Composition

a) A relationship shall always be between exactly two state classes.
b) The two related classes need not be distinct, i.e., a relationship may be recursive (see Figure 34).
c) A state class may participate in any number of relationships.

5.5.3.2 Verb phrase

a) Verb phrases shall be optional (although they should be used for clarity).
b) When a verb phrase is omitted, “has” shall be used to read the relationship.

5.5.3.3 Role name

a) A role name shall be specified when needed to disambiguate two or more relationships to the same state class.
b) When there is more than one relationship between the same pair of classes, the associated role names shall be distinct.
c) A role name shall conform to the rules of state class naming.
d) If a role name is omitted, the related class name shall be used as the role name when reading the relationship.

5.5.3.4 Naming

a) The name of a relationship shall be composed of the names of the related classes, along with their respective relationship role names, if any. For example, in Figure 33 the two relationships between city and flight are named `city (origin)`, `flight (outboundFlight)` and `city (destination)`, `flight (inboundFlight)`, respectively.

5.5.3.5 Intrinsic relationship

a) A dependent state class shall participate in at least one intrinsic relationship.
b) A dependent state class may participate in any number of nonintrinsic relationships.

5.5.3.6 Cardinality

a) Upon read, all cardinality (including “cardinality N”) shall be checked.

5.5.3.7 Deletion

a) When a state class instance is deleted with the built-in deletion property, the deletion shall propagate along relationships to the extent needed to comply with the cardinality constraints other than the "exactly N" constraint. There are three variations in support of differing cardinality specifications. To illustrate this, let classes `c1` and `c2` be related and an instance `I` of `c1` be deleted.

1) In the first case, if the cardinality specification states that each `c2` shall have exactly one `c1` (see Figure 35), then when an instance `I` of `c1` is deleted, every `c2` related to `I` shall also be deleted. In other words, given this cardinality specification, if an instance of `c2` is related to `c1` and `c1` is deleted, it would violate the cardinality constraint to leave `c2` by itself so `c2` shall also be deleted.

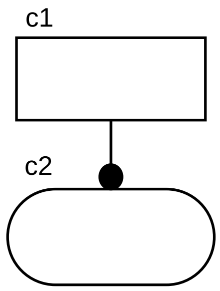


Figure 35—Deletion case 1

2) In the second case, if the cardinality specification states that each `c2` shall have at least one `c1` (see Figure 36), then when an instance `I` of `c1` is deleted, every `c2` related to only `I` shall also be deleted. In other words, given this cardinality specification, if an instance of `c1` is deleted and it is the last `c1` related to `c2` , then the deletion shall propagate to `c2`.

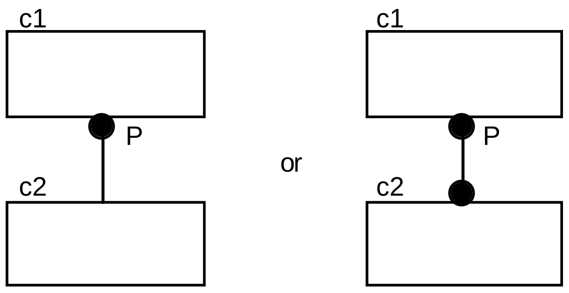


Figure 36—Deletion case 2

3) In all other cases, there shall be no propagation. Figure 37 illustrates only one of the many variations of cardinality specification for this case. When an instance `I` of `c1` is deleted, there shall be no propagation to `c2`. Likewise, when an instance `J` of `c2` is deleted, there shall be no propagation to `c1`.

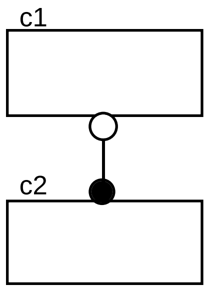


Figure 37—Deletion case 3

b) The default deletion property may be overridden
    1) To prevent deletion,
    2) To carry out additional or alternative action.

6. Responsibility

An instance possesses knowledge, exhibits behavior, and obeys rules. These notions are collectively referred to as the instance's *responsibilities*.

6.1 Introduction

A class abstracts the responsibilities common to its instances. During initial model development, and in survey-level and integration-level models in particular (see 8.2), a responsibility may simply be stated in general terms and not distinguished explicitly as an attribute, participant property, operation, or constraint. Also, aggregate responsibilities may be identified, rather than individual properties. Broadly stated responsibilities are eventually refined as specific properties and constraints.

In addition to these instance-level responsibilities, a class may also have *class-level responsibilities* in the form of attributes, operations, and constraints. These responsibilities constitute the knowledge, behavior, and rules of the class as a whole. For example, `name` and `lastDateVoting` would be instance-level properties of the class `registeredVoter`. The total `registeredVoterCount` would be a class-level property of the class `registeredVoter`. While each registered voter would have a value of `name` and `lastDateVoting`, there would be only one value of `registeredVoterCount` for the class as a whole.

6.1.1 Separation of interface from realization

According to the concept of abstraction, each responsibility (property or constraint) can be

a) Realized by stored data or by computation.
b) Understood without knowing how it is realized.
c) Requested in the same way, regardless of whether it is an attribute, participant property, operation, or constraint—and independent of how the responsibility is realized.

The specification of a responsibility has two parts: an interface and a realization,[34] each of which in turn may have two parts, as shown in Figure 38.

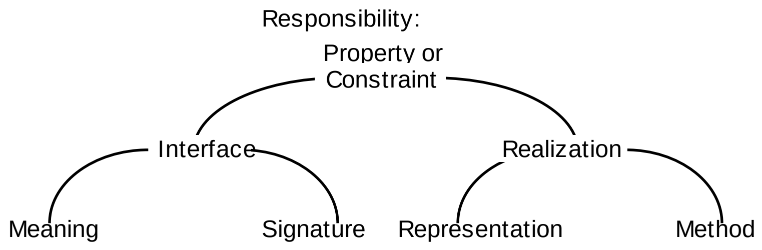


Figure 38—The elements of a responsibility

[34]The term "implementation," which may be more familiar to an object-oriented audience, has consciously not been used in order to avoid confusion since implementation tends to connote a particular language implementation. "Realization" in this document refers to the requests made to fulfill a responsibility in the specification language, independent of implementation language.

A sharp distinction is made between interface and realization. The *interface* encompasses the declaration of meanings and the signatures for properties and constraints. *Realization* encompasses the representation of these interface responsibilities through specified methods and any needed representation properties.

A class is *encapsulated* to the extent that access to the names, meanings, and values of the responsibilities is entirely separated from access to their realization. Encapsulation always hides the realization of a responsibility from the requester. In a model, this encapsulation takes the form of providing only the external interface of the classes, i.e., the meanings and signatures of the responsibilities. The specification language enforces encapsulation by not providing any way to access the representation of any responsibility directly.

Encapsulation is an enforcement mechanism for the concept of abstraction and is used to prevent one from seeing behind the abstraction. Encapsulation is most important as an implementation-time enforcer of abstraction. During modeling, the main concern is specifying classes and responsibilities according to the concept of abstraction. If the abstraction is done well, encapsulation will be able to enforce it. If the abstraction is not done well, some way will be found around the encapsulation out of necessity.

6.1.1.1 Interface

The interface states what an object is responsible to know or do (property) or what constraints it is responsible to adhere to (constraint). The interface specification consists of the *meaning* (semantics) and the *signature* (syntax) of a property or constraint.

The *meaning* of a responsibility is just that, what it means. The statement of responsibility is written from the point of view of the requester, not the implementer. It states what the requester needs to know to make intelligent use of the property or constraint. That statement should be complete enough to let a requester decide whether to make the request, but it should stop short of explaining how a behavior or value is accomplished or derived. Meaning is initially captured using freeform natural language text in a glossary description (see 8.4). It may be more formally refined into a statement of pre- and post-conditions using the specification language (see also 6.1.1.4 and 7.10.2).

A *signature* states what the responsibility “looks like.” It specifies the name of the responsibility, the arguments (if any), and the type of the result. A qualified responsibility name is the qualified class name followed by a colon (“:”) followed by the property name. A type (class) may be specified for each argument in order to limit the argument values to being instances of that class. Typing the arguments helps one to reason about the responsibility. However, insisting on typing too soon during model development is counter-productive. Therefore, both typed and untyped arguments are supported (see 7.4).

6.1.1.2 Request

A *request* encompasses the requests for properties and constraint checks and the sentences of such requests. The related concept of encapsulation is generally applied to the interface between a given instance and its requester clients. If the instance is well-encapsulated, then the client knows nothing about the internal implementation (realization) of any of the functionality of the instance. A request is simply made to the interface of the instance (see Figure 39). The requester does not need to know anything about the internals of the instance.

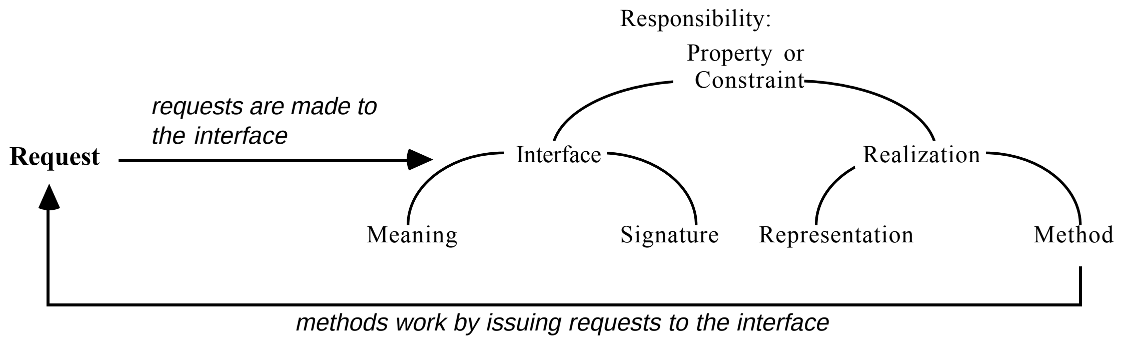


Figure 39—Request and interface

A responsibility of a class can itself be *hidden*, i.e., declared as either visible only to the instances of the class and its subclasses (protected), or visible only to requests issued by realizations of the responsibilities of the same class (private). Any interface that is not hidden is visible to any requester, i.e., *public*. Declaring a property or constraint hidden restricts the visibility of the interface to only specified requesters.

6.1.1.3 Realization

To meet its responsibilities, an instance may request the knowledge or behavior of other instances (see 6.2). The realization states “how” a responsibility is met. A realization specification consists of any necessary representation property(s) together with the method (if any). *Representation* provides the value of the value class instance or the state of the state class instance. Representation consists of one or more attributes or participant properties. A *method* is a statement of how property values are combined to yield a result.

A realization may involve representation properties or a method, or both. For example, an attribute may have only a representation and no method.[35] Figure 40 illustrates the `temperature` class using a real number as its hidden representation, known only to the `temperature` class to be a Kelvin temperature. (See 6.3 for an explanation of the graphic syntax used in these diagrams.)

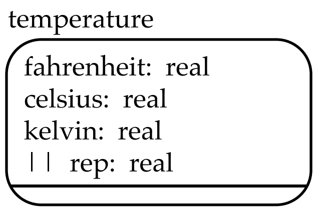


Figure 40—Hidden representation property

A derived attribute has a method and, typically, representation properties. If `hotel`’s `averageTemp` is derived, then it has a method,[36] such as illustrated in Figure 41 (the use of the multi-valued participant property `room` is illustrated):

[35]A realization that was a “pure method” (i.e., without any representation properties) would use only literals; it would not “get” any values as its inputs.

[36]Note this allows for hotels with no rooms and allows for rooms that have no temperature value. `Ts` is the list of (valued) `roomTemps`. In order for the hotel to have an `averageTemperature`, `Ts` must not be empty.

```
hotel: Self has averageTemp: T if_def
             Ts is [ Troom where ( Self has room..roomTemp: Troom ) ],
             not ( Ts == [ ] ),
             T is Ts..sum/Ts..count.
```

The method's *representation properties*, i.e., the properties on which the method operates, are `room` and `roomTemp` (see Figure 41).

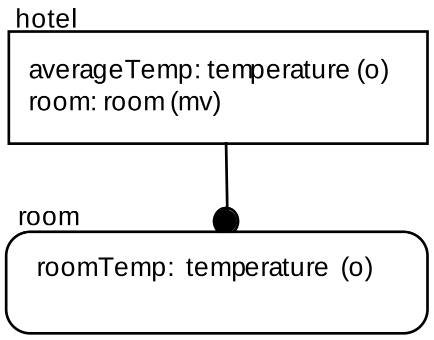


Figure 41—averageTemp (derived) with representation properties

If, on the other hand, `averageTemp` is cached rather than derived, it makes use of an internal (private), stored representation property, such as `savedTemp`, as shown in Figure 42. The realization method makes use of this representation property in its method (the use of the collection participant property `room(s)` is illustrated):

```
hotel: Self has averageTemp: T if_def
   (if not Self has savedTemp: T
   then
     Ts is [T where (Self has room(s)..member..roomtemp: T)],
     not ( Ts == [ ] ),
     Self..savedTemp:= Ts..sum/Ts..count
   endif),
   Self has savedTemp: T.
```

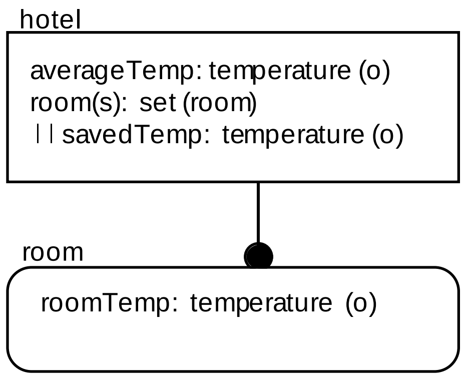

hotel
averageTemp: temperature (o)
room(s): set (room)
| | savedTemp: temperature (o)
room
roomTemp: temperature (o)

Figure 42—averageTemp (cached) with representation properties

In either case, the realization of `hotel`'s `averageTemp` is not known to the requester. In fact, the realization may be changed behind the scenes without impact on its requesters.

6.1.1.4 Pre-condition/post-condition

The meaning of a property can be further specified with pre-conditions and post-conditions. Each condition is a logical sentence about the property values of the instance to which the property request was directed and the values of the property arguments.

The meaning of the property is interpreted as being what the requester of the property can rely on; which is if the pre-conditions are true before the property request, then the post-conditions will be true after the property request.

Such pre- and post-conditions form the basis for *contracts* between the instance and those who send requests. In essence the contract is an assurance by the instance to the requester that "if you adhere to these pre-conditions, then you can depend on me to fulfill your request and ensure that the post-conditions are met." Thus, both the pre-conditions and post-conditions are considered to be "on the interface," i.e., visible to the requester of the property.

In the case of inheritance hierarchies, the effective pre-condition is a disjunction of the pre-conditions of overridden properties and the pre-condition of the overriding property. The effective post-condition is the conjunction the post-conditions of overridden properties and the post-condition of the overriding property. The *effective pre-condition*, *effective post-condition*, and the *realization* each consists of a sentence in the specification language that, when evaluated, is true or false. The overall logic is as follows:

```
if effective pre-condition
then
    if realization
    then
        if effective post-condition
        then
            request is true
        else
            post-condition failure
        endif
    else
        request is false
    endif
else
    pre-condition failure
endif
```

A "pre-condition failure" or "post-condition failure" is an exceptional condition. This standard does not specify what happens when an exceptional condition is encountered; conceptually "everything stops." An exception condition is simply a flaw in the model. (See also Clause 7.)

6.1.2 Responsibility specification and use

The specification and use of responsibilities involve the following:

a) A responsibility is specified as part of the interface of a class. The responsibility is named, its meaning is stated in natural language, and various declarations are made about the responsibility to permit requests for it to be made correctly.

b) A realization (method) specifies how an instance maps a responsibility's input arguments to its output arguments. Methods lie behind the interface of a class. Methods are stated with the specification language (see Clause 7). Methods consist of requests for other interface responsibilities.

c) A specific request for a responsibility is a request for the receiver to map the specific input argument values to the corresponding output argument values.
d) Solving for a specific responsibility request, i.e., doing the mapping, typically involves sending specific requests to other instances, as determined by the method, the identity of the receiver, and the value of the input arguments.

This version of the standard provides graphics and a specification language for the semantics and syntax of class interfaces, requests, and realizations. This version of the standard does not provide a set of graphics describing individual requests or patterns of requests.

6.2 Request

To meet its responsibilities, an instance may request the knowledge or behavior of other instances by sending them messages. A *request* is a message sent from one object (the sender) to another object (the receiver), directing the receiver to fulfill one of its responsibilities. Specifically, a request may be for the value of an attribute, for the value of a participant property, for the application of an operation, or for the truth of a constraint. Request also encompasses sentences of such requests. Logical sentences about the property values and constraints of objects are used for queries, pre-conditions, post-conditions, and responsibility realizations.

A request is the only way to access a property value, apply an operation, or check a constraint. A request is made by a sender to a receiver’s interface; the sender does not need to know anything about the realization. See Clause 7 for details on specific requests such as `create`, `delete`, `display`, and `query`.

6.2.1 Request semantics

A request is defined to be a *logical proposition*.[37] The declarative nature of a logical proposition gives the request a dual nature. First, as a request, it is asking some designated instance `I` for the value `V` of a specified property `P`. Secondly, as a proposition, it is asserting that some designated instance `I` has a specified property `P` with a value `V`.

The instance that receives a request for a property value will either use a previously saved value or derive a value using its method. That method itself consists of requests for the property values (or actions or constraint checks) of the other instances with which the instance receiving the original request decides to collaborate. Thus, a request encompasses both the individual requests for properties and constraint checks and the sentences of such requests.

6.2.1.1 Requests for properties and constraint checks

A property is defined as a mapping from the receiver and the input arguments to the output arguments. If the mapping is visualized as a table with a column for the object identity (`I`) and a column for each argument (`A1,...An`), then the request is satisfied if there is a row in the table matching on the receiver (`I`) and all the arguments that had values when the request was made. The values in the row for the other arguments (those without a value at time of request) are the *solution* for those arguments. An example of this for operation is shown in Figure 64.

In addition to having a solution (the output argument values set), a request also has a *truth value*. If the mapping succeeds (there is a row in the visualized table), then the request is true; if it fails (there is no row), then the request is false.

A request for an operation without arguments has no solution *per se*; it is simply true or false. The mapping in this case can be visualized as a table with a single column for the object identity (`I`). The table contains

[37]See Clause 7 for a discussion of propositions.

only the instances for which the operation is true. The request is true if there is a row matching on the request receiver (`I`). Alternatively, the mapping can be visualized as a table with two columns, one for the object identity and the other containing either "true" (succeeds) or "false" (fails), as the operation is either true or false for that instance.

A request for a constraint is the same as for an operation without arguments. The constraint check is simply true or false.

A request for an attribute has no input arguments and only one output argument (the identity of the related value class instance). A request for a participant property has no input arguments and only one output argument (the identity of the related state class instance).

6.2.1.2 Logical sentences of requests

Logical sentences over property values are used for the realization of responsibilities (stating the methods behind properties and constraints) and for stating pre-conditions and post-conditions. Together, a set of requests forms a sentence specifying the necessary and sufficient conditions for the property to have the value V (or for the constraint, pre-condition, or post-condition to be satisfied). Every instance of a class uses the same method to obtain the value of a given property or to check a named constraint.[38]

A request is the atomic formula out of which such sentences are constructed. A logical sentence is formed from requests (logical propositions) combined using *logical connectives* such as `and`, `or`, `not`, `if then`, `forall`. The truth of the sentence depends on the truth of its constituent propositions. If the sentence is true, then the property has the value V (or the constraint or condition holds); otherwise it does not.

If a request involves multiple updates, and the request fails (i.e., is false), the state of the view "rolls back" to the state prior to the request. If a request is of the form `forall F: G`, only `G` is allowed to perform updates.

As an example of the use of `forall`, the `commonOwner` constraint of every `checkingAccount` can be checked by the sentence:

```
forall ( checkingAccount has instance: CA): ( CA has commonOwner ).
```

This sentence can be read as "for every `checkingAccount` instance, `CA`, the `commonOwner` constraint is true for `CA`" or in a more natural fashion, "every checking account satisfies the common owner constraint" (see 6.7). The sentence is false if for any instance `CA`, the `commonOwner` constraint is not true.

6.2.2 Request syntax

The form of a request is the same for attributes, participant properties, operations, and constraints; it depends only on the number of arguments and the nature of the request (get, set, etc.). The general syntax is explained here and illustrated individually in 6.4.4 (for attribute), 6.5.4 (for participant property), (6.6.4 for operation), and 6.7.4 (for constraint).

The operators `:=` (set value) and `:!=` (unset value) are valid only for single-valued properties. The operators `:+=` (insert value) and `:-=` (remove value) are valid only for multi-valued properties and single-valued collections.

6.2.2.1 Single-argument request: get value

a) A "get value" request with one argument shall have the form:

[38]See Clause 7 for a discussion of sentences and messages.

```
I has P: V
```

where `I` is an instance, `P` is a property of `I`, and `V` is the assumed value of that property.

This syntax covers two cases:

1) The variable `V` does not have a value when the request is made. In this case,
   i) If `I` has a value for its property `P`, then the request shall be true, and the request shall set `I`’s value for its property `P`—i.e., the solution shall be value `V`.
   ii) Otherwise, the request shall be false and shall have no solution.
2) The variable `V` has a value when the request is made. In this case,
   i) If `I`’s property `P` value is `V`, then the request shall be true.
   ii) Otherwise, the request shall be false, and `V`’s value shall be unchanged.

b) A “get value” request without any argument has the form:

```
I has P
```

where `I` is an instance and `P` is a `boolean` attribute or a single `boolean` argument operation. This request is equivalent to the request

```
I has P: true
```

where “`true`” is an instance of `boolean`.

6.2.2.2 Single-argument request: set value

a) A “set value” request with one argument shall have the form:

```
I has P:= V
```

where `I` is an instance, `P` is a property of `I`, and `V` is the value for the property. This syntax covers two cases:

1) `V` has no value when this request is made. In this case, an exception shall be raised.
2) `V` has a value when this request is made. In this case, if P is allowed to be the value V,[39] if P is constant and has no value when the request is made, or if `P` is not constant, then `I`’s `P` value shall be set to `V` and the request shall be true. Otherwise, the request shall be false.

6.2.2.3 Single-argument request: unset value

a) An “unset value” request with one argument shall have the form:

```
I has P:!= V
```

where `I` is an instance, `P` is a partial, nonconstant property of `I`, and `V` is the value of that property.

This syntax covers two cases:

1) `V` has no value when this request is made. In this case, if `I`’s property `P` has a value, then `V` shall be set to `I`’s current `P` value, `I`’s current `P` value shall be cleared, and the request shall be true. If `I` does not have property `P` value, then the request shall be false.
2) `V` has a value when this request is made. In this case, if `I`’s property `P` value is currently `V`, then `I`’s `P` value shall be cleared and the request shall be true. Otherwise, the request shall be false.

6.2.2.4 Single-argument request: insert value

a) An “insert value” request with one argument shall be used for collection-valued or multi-valued attributes and participant properties to add an element to the collection. It shall have the form:

```
I has P:+= V
```

where `I` is an instance, `P` is a nonconstant, collection-valued or multi-valued property of `I`, and `V` is the value to add to the collection.

This syntax covers two cases:

1) `V` has no value when this request is made. In this case, an exception shall be raised.

[39]The value could be disallowed for various reasons, such as the attribute being declared `constant` and already having a value, the proposed new value causing a uniqueness constraint to fail, a value constraint to be violated, etc.

2) `V` has a value when this request is made. In this case, if the collection is allowed to contain the value `V`, `V` shall be added to `I`'s `P` collection and the request shall be true. Otherwise, the request shall be false.

b) An attempt to add an element to a collection shall fail if it would result in a violation of a declared collection cardinality restriction.

6.2.2.5 Single-argument request: remove value

a) A "remove value" request with one argument shall be used for collection-valued or multi-valued attributes and participant properties of state classes to remove an element from the collection. It shall have the form:

```
I has P:-= V
```

where `I` is an instance, `P` is a nonconstant, collection-valued or multi-valued property of `I`, and `V` is an element to be removed from the collection.

This syntax covers two cases:

1) `V` has no value when this request is made. In this case, if `I`'s property `P` has a value, then `V` shall be set to the first of `I`'s current `P` values, that value shall be removed from `I`'s current `P` values, and the request shall be true. If `I` does not have property `P` value, then the request shall be false.

2) `V` has a value when this request is made. In this case, if `I`'s property `P` includes that value, then that value shall be removed from `I`'s current `P` values, and the request shall be true. Otherwise, the request shall be false.

b) An attempt to remove an element from a collection shall fail if it would result in a violation of a declared collection cardinality restriction. For example, an attempt to remove the last element from a collection not allowed to be empty shall fail.

6.2.2.6 No-argument request

a) The form of a constraint check request or a request for a property without arguments is:

```
I has P
```

where `I` is an instance and `P` is either a named constraint or an operation of the class of `I` .

For example, (referring to Figure 69) if instance `CCA` of `creditCardAccount` has a `limit` of $15,000 and a `balance` of $10,000, then the request (constraint check)

```
CCA has balanceUnderLimit
```

is true. However, an attempt to increase the `balance` by $7,000 violates the constraint (i.e., the constraint "fails.")

To illustrate an operation, assuming that `creditCardAccount` has an operation `closeAccount` that terminates an active account, then the request

```
CCA has closeAccount
```

terminates the account and is true, if `CCA` is currently active. Otherwise, it is false.

6.2.2.7 Multiargument request

a) Only operations may use multiargument requests.

See 6.6.4 for specific examples.

6.2.2.8 Alternate syntax forms

a) Alternate forms of the request syntax may be used. Table 3 illustrates two examples of typical, useful syntax equivalents. The full syntax for requests is covered in Clause 7.

Table 3—Alternative forms of request syntax

Instead of	Equivalent syntax	Example
`I has P : V`	`V is I..P`	`'Tom' is Cust#1..name`
`I has P: V1, V1 has P2: V2`	`I has P1..P2: V2`	`Account has owner..name: 'Joe'`

6.2.3 Request rules

6.2.3.1 Evaluation

The following rules summarize how a request shall be satisfied. (See 7.5 for the details of message resolution.)

a) A request

```
I has P: A
```

shall be true if each of the following steps is true:

1) Class `C` is the most specific class (lowest in the hierarchy) for which `I` as an instance of `C` and `P` is a directly specified (not inherited) property of `C`.
2) The method is specified as

   `C: Self has P: V if`$_{def}$ `Sentence.`
3) If the type of the argument is `T` and `A` has a value, then `A` is an instance of `T`.
4) The effective pre-condition is `Pre`.
5) The effective post-condition is `Post`.
6) The conjunction

`Self = I`	– binding the formal parameter `Self` to the receiver instance `I`
`V = A`	– binding the formal argument `V` to the actual argument value `A`
`Pre`	– evaluating the effective pre-condition (which must be true)
`Sentence`	– evaluating the method (which must be true)
`Post`	– evaluating the post-condition (which must be true)

   is true.
7) If the type of the argument is `T` and `A` has a value, then `A` is an instance of `T`.

Steps 1 and 2 find the right method. Step 3 type checks the input arguments. Steps 4 and 5 find the pre-conditions and post-conditions, setting the variables `Pre` and `Post`, respectively. Step 6 solves for the truth of the request. Step 7 type checks the output arguments.

b) If a request for a property fails, no updates shall be done.

6.3 Property

Some responsibilities[40] are met by knowledge and behavior that, in turn, are determined by properties. A *property* is an inherent or distinctive characteristic or trait that manifests some aspect of an object's knowledge or behavior. There are three kinds of property:

a) *Attributes*
b) *Participant properties* due to relationships
c) *Operations*

[40]Other responsibilities are met by adhering to imposed constraints; these are discussed in 6.7.

A class has properties; a class instance has property values. A class instance's knowledge is determined by the values of its attributes, participant properties, and constant, read-only operations. A class instance's behavior is determined by the state-changing operations it can perform.

The concepts that are common to properties in general are described in this subclause. The specializations of property are discussed and illustrated individually in 6.4 (attributes), 6.5 (participant properties), and 6.6 (operations). Only the distinctions will be discussed in these latter subclauses. Statements that apply generally to all three are made here and are not repeated in the more specific subclauses.

6.3.1 Property semantics

6.3.1.1 Naming/signature

All properties are named. The detail of property naming is discussed individually in 6.4.1.1 and 6.4.2.3 (for attribute), 6.5.1.1, 6.5.2.2, and 6.5.3.1 (for participant), and 6.6.1.1 and 6.6.2.3 (for operation). For a multi-valued property, a corresponding single-valued collection property is often needed, and vice versa. The rules for name construction of such properties is discussed in 6.3.1.6 through 6.3.1.8 and 6.3.2.4.

In certain cases, properties with the same name may occur in multiple classes (or even the same class). In one case, a name may have one consistent meaning but differing signatures[41] (in the same or different classes). In another case, properties of the same name may have differing meanings in different classes. Furthermore, a property of a given name and the same meaning (with the same or different signatures) is permitted to be a property of a class as well as any of its subclasses. In this case, the property in the subclass is said to *override* the property in the superclass, i.e., the property has the same name and same meaning but a different realization.

This facility is powerful, but it can easily be abused.[42] The semantic concept used to constrain overriding is the principle of substitutability (see 5.4.1).

6.3.1.2 Visibility

Encapsulation always hides the realization of a property. However, the interface of a property can also be hidden by declaring it as either

a) *Protected*—visible only to the class or the receiving instance of the class (available only within methods of the class or its subclasses), or
b) *Private*—visible only to the class or the receiving instance of the class (available only within the methods of the class)

A property that is not hidden is considered *public*, i.e., visible to any requester (available to all). The abstract interface for a requester includes whatever is visible to the requester.

Initial modeling is concerned primarily with public properties. However, protected or private properties are typically introduced in a fully specified model (see 8.2) in support of a constraint or property realization.

6.3.1.3 Instance-level/class-level

A property can be an instance-level property or a class-level property. A property is at the instance level if it applies to each instance individually. An example of an instance-level property of the customer class might be `customerName` (an attribute). Each instance of customer may have a name specified.

[41]See 6.3.2.1 for the definition of *signature*. See 7.5.1 for a discussion of signature matching.

[42]Using the same name with the same meaning but with different *signatures* or *realizations* can be appropriate. Using the same name but with different *meanings* is problematic, especially if both names appear in the same view, since using the same name for two distinct concepts makes it more difficult to reason about the concepts.

A property is at the class-level if it applies to the class as a whole. An example of a class-level property of the customer class might be `numberOfCustomers`, specified as the count of the number of instances in the class. As another example, a single maximum credit limit (e.g., $100) might apply equally to all instances of the `customer` class.

6.3.1.4 Mapping

All properties are mappings between things.[43] As depicted in Figure 43, a mapping `M` from a set `D` to a set `R` is a set of ordered pairs `[ X, Y ]` where `X` is in `D` and `Y` is in `R`. An attribute is a mapping from a class or the instances of a class to the instances of a value class. A participant property is a mapping from the instances of a state class to the instances of a state class. An operation is a mapping from the (cross product of the) class or instances of the class and the instances of the input argument types to the (cross product of the) instances of the other (output) argument types. Examples of each of these types of mapping are given in 6.4, 6.5, and, 6.6, respectively.

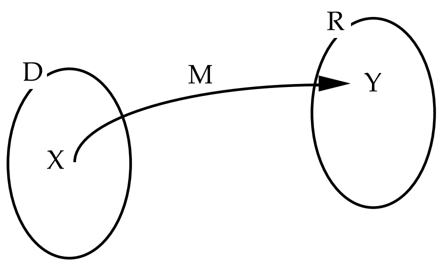


Figure 43—Mapping

6.3.1.5 Mapping completeness

A mapping is either a *total* mapping (every element in `D` maps to an element in `R`) or a *partial* mapping (some elements in `D` are unmapped). Referring to Figure 43, the mapping `M` is *total* if for every `X` in `D`, there is at least one pair `[ X, Y ]` in `M`. A property declaration of `mandatory` constrains a mapping to be a total mapping (see 6.3.2).

If a mapping is *partial*, the property is allowed to have no value.[44] A property declaration of `optional` allows a mapping to be a partial mapping (see 6.3.2).

6.3.1.6 Single-valued/multi-valued

A mapping `M` is *single-valued* if for any `X` in `D`, there is at most one pair `[ X, Y ]` in `M`. If the mapping is single-valued, it is a *function.* A property declaration of `single-valued` constrains the mapping to be a function (see 6.3.2). A property with a single-valued mapping is referred to as a *single-valued property.*

[43]In mathematics, this mapping is called a “relation.”

[44]The precise interpretation of “no value” has bedeviled database theorists for decades. There is as yet no definitive interpretation. Various popular options have attempted to make distinctions, such as “there is a value but it is unknown,” “such a value is not applicable,” and “it may be applicable but there is no such value for this instance.” For example, if a bank provides for tracking when disputes occur for accounts, a checking account might have a `lastDisputeDate` attribute. For accounts that are never involved in a dispute, there will never be a value for such a date; it is inapplicable to these instances. As another example, an employee's birthdate may be unknown when the instance is created, and the attribute may have no value for some period of time. The option this standard adopts is that the lack of a value means simply “there is no such value.” There is no implication that “no value” means there is a value but it is not known; there is no insistence that “no value” means that such a value is inapplicable. The interpretation is simply that, if there is no value, then there really is no value. It means that, according to our model, an employee with no birthdate value has no birthdate. This is analogous to the unexamined assumption that, if the value recorded for an attribute is 3, then the value in the real world really is 3.

If a mapping is not a function, it is *multi-valued* (i.e., elements in `D` map to multiple elements in `R`). A property declaration of `multi-valued` allows the mapping to be multi-valued (see 6.3.2). The default is `single-valued`. In terms of Figure 43, single-valued and multi-valued describe the mapping `M`. A property with a multi-valued mapping is referred to as a *multi-valued property.*

6.3.1.7 Scalar-valued/collection-valued

The class to which a property maps may be a scalar-valued class or collection-valued class. A *scalar-valued class* is a class in which each instance is a single, atomic value, such as an integer. Each instance of a *collection-valued class* is a collection of values, such as a set of integers.

For a property that maps to a scalar-valued class, a property value `V` is a single, atomic value. For a property that maps to a collection-valued class, a property value `V` is itself a collection of values. In both cases, the property value `V` is a single instance of the class to which the property maps. Note that in terms of Figure 43, scalar-valued and collection-valued describe the class `R`.

A property that maps to a scalar-valued class is referred to as a *scalar-valued property* (simply *scalar property*). A property that maps to a collection-valued class is referred to as a *collection-valued property* (simply *collection property*).

6.3.1.8 Implicit properties

If a class has a collection property `p(s),` then it implicitly has a corresponding multi-valued property `p`. If a class has a multi-valued property `p,` then it implicitly has a corresponding collection property `p(s)`. For every instance `I` and value `V`, the properties `p` and `p(s)` are related according to the following:

`I has p: V` if and only if `I has p(s)..member: V.`

6.3.1.9 Collection cardinality

For a property that maps to a collection class, the values of the property can be constrained to a specific cardinality by a declaration of its *collection cardinality.* For example, a declaration of `cardinality Positive` prohibits the empty collection. If no collection cardinality is specified, a collection-valued property may map to a collection of any number of members, including zero (the empty collection).

6.3.1.10 Constant

An attribute or participant property is a *constant* if it is unchanging once assigned. For example, the `openedDate` of a checking account is assigned when the account instance is created and not changed during the life of the account. An operation is a constant if the same set of input values always yields the same set of output values.

A property that is declared to be both `mandatory` and `constant` must be assigned when the instance is created. A property that is declared to be `optional` and `constant` may be left unassigned when the instance is created and then assigned later. Once assigned, it may not be changed.

6.3.1.11 Read-only

A property can be declared to be `read-only`. A property is *read-only* if it causes no state change, i.e., it does no updates.

6.3.1.12 Intrinsic

A property can be declared to be `intrinsic`, specifying that the property is constant and has a single-valued, total mapping. Single-valued specifies the mapping; the mapped-to value could be a collection.

6.3.1.13 Subclass responsibility

A property of an abstract class that must be overridden in its subclasses is declared to be a *subclass responsibility*. A property that is a subclass responsibility is a specification in the superclass of an interface that each of its subclasses must provide; its realization is not specified in this class. Instead, its realization is deferred to the subclass(es) of the class.

6.3.1.14 Uniqueness constraint

A *uniqueness constraint* is a specification that no two distinct instances of the class may agree on the values of all the properties named in the uniqueness constraint. An attribute or participant property can be declared to be a part of a uniqueness constraint.

With the concept of identity there is no need that every class declare a restriction that forbids any two instances of a class from agreeing on all property values, i.e., there is no inherent requirement to declare a primary key. (See also 6.7.2.)

6.3.1.15 Derived

A property whose value is determined by computation is *derived*. An attribute or participant property can be derived. A derived property has no implicit realizations; the modeler must provide the realizations.

6.3.2 Property syntax

As an aid to reasoning about properties and arguments, various elaborations on the nature of the property may be declared using the constructs of `Visibility`, `PrefixCommaList`, `Arguments`, and `SuffixCommaList`. Figure 44 illustrates several combinations of properties, arguments, and declarations for both a state class and a value class.

The syntax that is common to properties in general is described here. The specializations of property syntax are discussed and illustrated individually in 6.4.2 (for attribute), 6.5.2 (for participant property), and 6.6.2 (for operation). Only the distinctions will be discussed in these latter subclauses. The common syntax is presented here and is not repeated in the more specific subclauses.

6.3.2.1 Naming/signature

a) The *signature* of a property shall consist of (in this order):
   1) The class name,
   2) A colon,
   3) The property name,
   4) A property operator, and
   5) The number and type of its arguments.
b) A *fully qualified property name* shall consist of (in this order):
   1) The fully qualified class name,[45]
   2) A colon,
   3) The property name,
   4) The property operator,

[45]See 5.1.3.1 and 8.1.3.1 for a description of fully qualified naming.

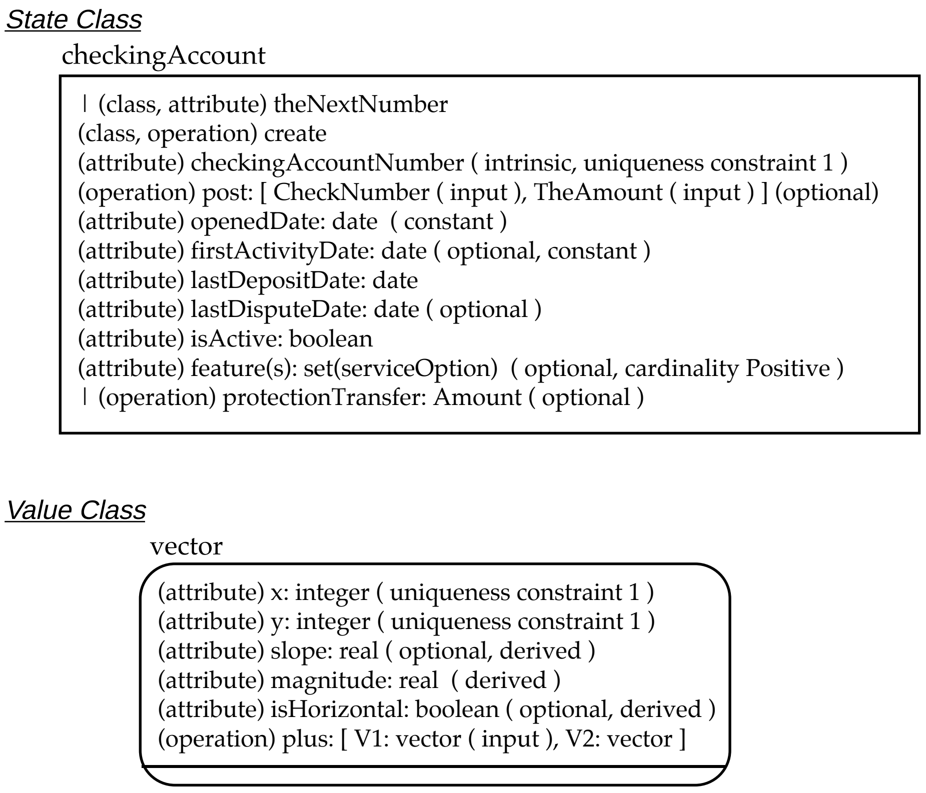

State Class
checkingAccount
| (class, attribute) theNextNumber
(class, operation) create
(attribute) checkingAccountNumber (intrinsic, uniqueness constraint 1)
(operation) post: [CheckNumber (input), TheAmount (input)] (optional)
(attribute) openedDate: date (constant)
(attribute) firstActivityDate: date (optional, constant)
(attribute) lastDepositDate: date
(attribute) lastDisputeDate: date (optional)
(attribute) isActive: boolean
(attribute) feature(s): set(serviceOption) (optional, cardinality Positive)
| (operation) protectionTransfer: Amount (optional)
Value Class
vector
(attribute) x: integer (uniqueness constraint 1)
(attribute) y: integer (uniqueness constraint 1)
(attribute) slope: real (optional, derived)
(attribute) magnitude: real (derived)
(attribute) isHorizontal: boolean (optional, derived)
(operation) plus: [V1: vector (input), V2: vector]

Figure 44—Properties with declarations

5) The type, where type is
   i) the type of that argument (for a single-argument property), or
   ii) a list of the types, one per argument (for a multiargument property).

For example, in Figure 44, the `vector` value class shows a signature for the property `slope`. In this signature, the property name is `slope`, the property operator is `:`, and its single argument is of type `real`.

Similarly, `lastDepositDate` in `checkingAccount` shows a signature in which `lastDepositDate` is the property name, `:` is the property operator, and its single argument is of type `date`. There are two additional, implied signatures for this property as follows:

— `viewName:checkingAccount:lastDisputeDate:= date` (to set its value), and

— `viewName:checkingAccount:lastDisputeDate:!= date` (to clear its value).

c) In a diagram, the property signature may be shown as an *annotated property signature* (the signature with additional keyword annotations) inside the class rectangle.

d) The general form of an annotated property signature[46] is

Visibility	(PrefixCommaList)	PropertyName	Arguments	(SuffixCommaList)

e) With the exception of the `PropertyName`, each of the elements of the annotated property signature shall be optional.

f) As in all labels (see 4.2.3), *whitespace* (spaces, tabs, etc.) in the annotated property signature shall be maintained.

[46]Informal diagrams like this and Figure 45 are used throughout this clause to illustrate signature syntax. The precise syntax is provided by the BNF in Clause 7.

6.3.2.2 Visibility annotation

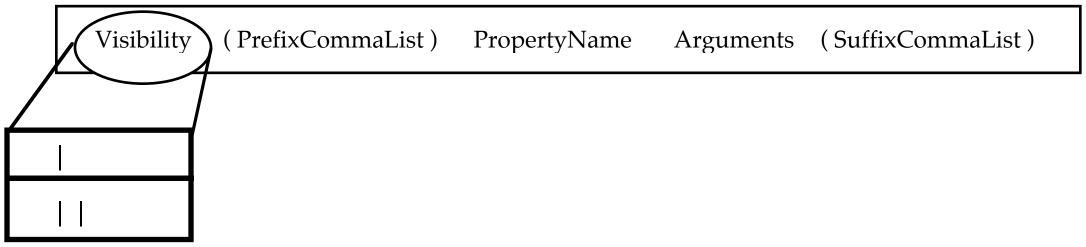


Figure 45—Property visibility annotation

a) The visibility annotation of a property shall specify whose methods may reference the property—i.e., “who can see it?” (see Figure 45). The interpretation of the visibility annotation shall be as presented in Table 4.

b) For a property declared with the get property operator (`:`), the default visibility for the update operators (`:=, :!=, :+=, :-=`) is private if the get is private, otherwise protected.

Table 4—Interpretation of visibility notation

If the property is	Meaning that the property is accessible to	Then this visibility prefix shall be used
Public	All (accessible without restriction)	*unannotated*
Protected	The class or the receiving instance of the class within methods of the class and its subclasses	|
Private	The class or the receiving instance of the class within methods of the class only	||

6.3.2.3 PrefixCommaList clause

`PrefixCommaList` is a comma-separated list of one or more keywords (see Figure 46).

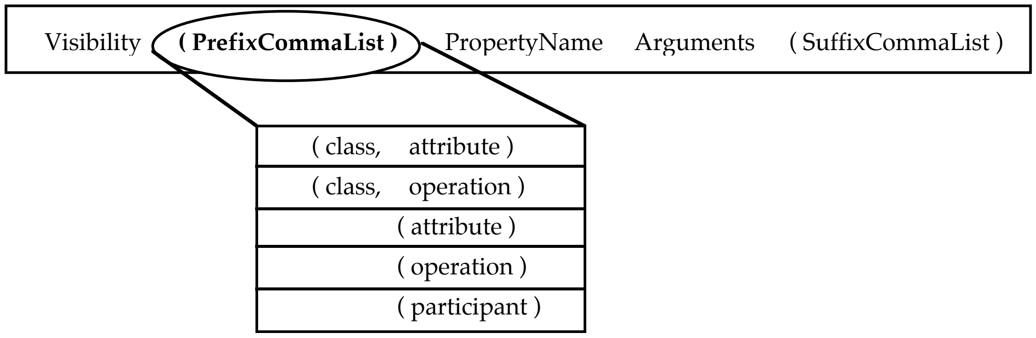


Figure 46—PrefixCommaList

a) A property may be designated as one of the keywords in Table 5.

For example, in Figure 44, `checkingAccountNumber` is an attribute property, and `post` is an operation property. Figure 61 illustrates participant properties.

b) “`class`” shall designate a property as a class-level property.

c) If "`class`" is not specified, the property shall be an instance-level property.

d) The keywords in a `PrefixCommaList` may come in any order—i.e., keyword order shall not be meaningful.

e) Multiple `PrefixCommaLists` shall be equivalent to a single `PrefixCommaList` with the keywords separated with commas. For example, `(class) (attribute)` is equivalent to `(class, attribute)`.

Table 5—PrefixCommaList keywords

Keyword	Meaning
`attribute`	The property is an attribute
`participant`	The property is a participant property
`operation`	The property is an operation

6.3.2.4 Property name clause

a) The property name shall be suffixed with `(s)` for any single-valued, collection property. This suffix shall be used only with a single-valued collection property.

6.3.2.5 Arguments clause

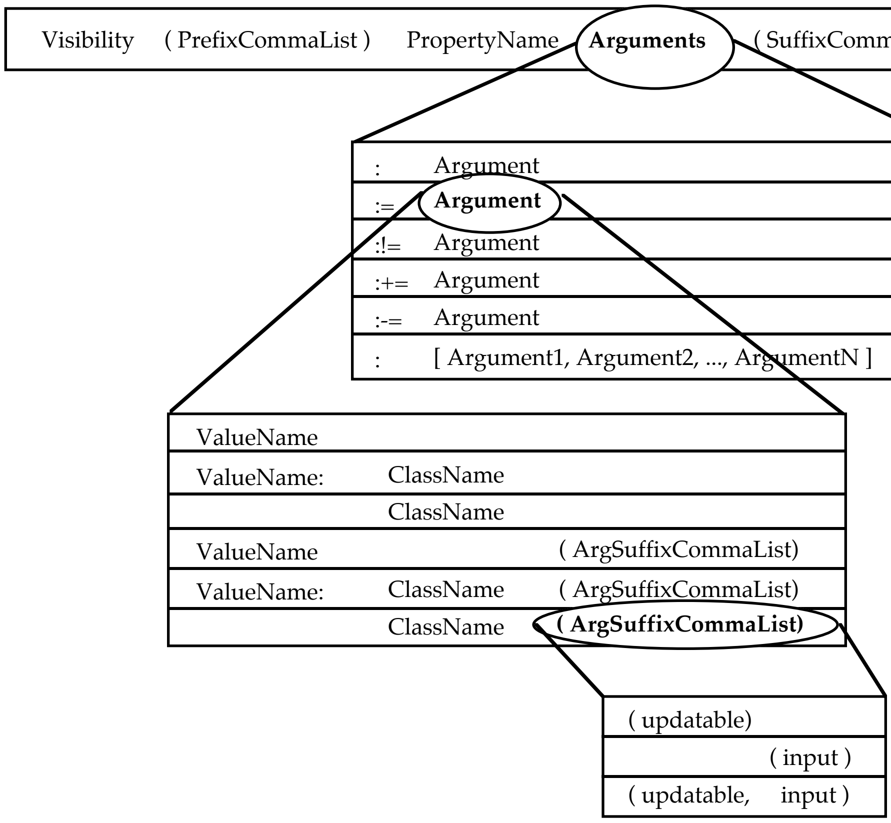


Figure 47—Arguments

a) The `Arguments` clause shall be either:

   1) One of:

`: Argument`
`:= Argument`
`:!= Argument`
`:+= Argument`
`:-= Argument`

   where the property operators[47] shall have the meaning:

`:`	get value
`:=`	set value
`:!=`	unset value
`:+=`	insert value (to a collection or multi-valued property)
`:-=`	remove value (from a collection or multi-valued property)

   or

   2) A list of arguments (for an operation property only):[48]

```
  : [ Argument1, Argument2, ..., ArgumentN ]
```

b) `Argument` shall be one of:

`ValueName`		
`ValueName:`	`ClassName`	
	`ClassName`	
`ValueName`		`( ArgSuffixCommaList )`
`ValueName:`	`ClassName`	`( ArgSuffixCommaList )`
	`ClassName`	`( ArgSuffixCommaList )`

c) `ValueName` shall be the name of the argument value.

d) `ValueName` shall have its first letter capitalized.

e) `ClassName` shall be the argument type.

f) `ClassName` shall either begin with a lower-case letter or be enclosed in single quotes.

g) `ArgSuffixCommaList` shall only be applicable for operation properties (see also 6.6.3).

h) `ArgSuffixCommaList` shall be a comma-separated list of one or more keywords, including

   1) `updatable`, and

   2) `input`.

i) The items in an `ArgSuffixCommaList` may come in any order—i.e., order shall not be meaningful.

[47]See Clause 7 for a full explanation of these property operators.

[48]Arguments are not named. The realization is matched to the interface by the position of the arguments. See Clause 7 for the detailed rules for typing.

6.3.2.6 Overriding the property operator built-ins

a) The property operators are, in essence, built-in methods for which IDEF1X implements a special syntax. It shall be possible to override the built-in semantics of these operators to allow (for example) the modeler to specify a trigger on a "set value" statement (see also Clause 7).

6.3.2.7 SuffixCommaList clause

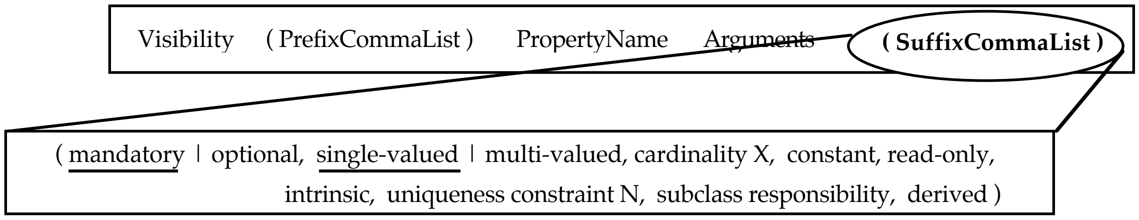

Visibility
(PrefixCommaList)
PropertyName
Arguments
(SuffixCommaList)
(mandatory | optional, single-valued | multi-valued, cardinality X, constant, read-only, intrinsic, uniqueness constraint N, subclass responsibility, derived)

Figure 48—SuffixCommaList

a) `SuffixCommaList` shall be a comma-separated list of one or more of the following keywords:
   1) `mandatory` | `optional`,
   2) `single-valued` | `multi-valued`,
   3) `cardinality X`,
   4) `constant`,
   5) `read-only`,
   6) `intrinsic`,
   7) `uniqueness constraint N`,
   8) `subclass responsibility`,
   9) `derived`,

   where "`|`" denotes alternative keywords and underlining designates the default keyword if none is explicitly specified.
b) "`cardinality X`" means that the collection property's value (a collection) shall have a collection cardinality restriction where `X` shall be one of the options in Table 6:

Table 6—Options for coordinating restriction

Option	Meaning
`Positive`	Collection may not be empty
`Zero`	Collection shall have a maximum of one member
`N`	Collection shall contain exactly `N` members, where `N` is any positive integer

c) "`read-only`" means the property shall cause no state changes, i.e., shall do no updates.
d) "`uniqueness constraint N`" means the property shall be part of "uniqueness constraint N" where `N` shall be an unsigned nonzero integer.
e) Indicating that a property is `derived` shall be part of the realization—i.e., it shall be supported by the graphics only as a convenience for the author of the class.
f) Order shall not be meaningful in a `SuffixCommaList`—i.e., the items in a `SuffixCommaList` may appear in any order.
g) Repeated, contiguous occurrences of the same whitespace character within a `SuffixCommaList` may be collapsed into a single occurrence of that character.

h) Multiple `SuffixCommaLists` shall be equivalent to a single `SuffixCommaList` with the keywords separated with commas. For example, `(subclass responsibility) (derived)` is equivalent to `(subclass responsibility, derived)`.

6.3.2.8 Keyword combinations

a) Table 7 summarizes the interpretation of the keyword combinations for mapping completeness and collection cardinality as they shall apply to single-valued and multi-valued, scalar, and collection properties.

Table 7—Keyword combination for mapping completeness and collection coordinality

Property type	**Mapping completeness**	**Collection cardinality**	***Specification***
Single-valued, scalar property	`mandatory`	–	Shall always be mapped (to exactly one value).
	`optional`	–	May be unmapped; When mapped, shall be mapped to exactly one value.
Single-valued, collection property	`mandatory`	–	Shall always be mapped (to exactly one collection) with no restriction on the collection cardinality.
	`mandatory`	`ca X`	Shall always be mapped (to exactly one collection) with a restriction to a specified collection cardinality.
	`optional`	–	May be unmapped; When mapped, shall be mapped to exactly one collection with no restriction on collection cardinality.
	`optional`	`ca X`	May be unmapped; When mapped, shall be mapped to exactly one collection with a restriction to a specified collection cardinality.
Multi-valued, scalar property	`mandatory`	–	Shall always be mapped (to at least one value).
	`optional`	–	May be unmapped; When mapped, shall be mapped to any number of values.
Multi-valued, collection property	`mandatory`	–	Shall always be mapped (to at least one collection) with no restriction on collection cardinality.
	`mandatory`	`ca X`	Shall always be mapped (to at least one collection) with a restriction to a specified collection cardinality.
	`optional`	–	May be unmapped; When mapped, shall be mapped to any number of collections with no restriction on collection cardinality.
	`optional`	`ca X`	May be unmapped; When mapped, shall be mapped to any number of collections with a restriction to a specified collection cardinality.

b) If a single-valued, collection property `x(s)` is declared
   1) `mandatory`, and
   2) `ca P or ca N` (for `N` greater than `0`),

   then the cardinality of the implicit multi-valued property `x` shall be `mandatory`; otherwise it shall be `optional`.
c) If a multi-valued property `x` is declared `mandatory`, then the cardinality of the implicit single-valued, collection property `x(s)` shall be
   1) `mandatory`, and
   2) `ca P`.
d) If a multi-valued property `x` is declared `optional`, then the cardinality of the implicit single-valued, collection property `x(s)` shall be `optional`.

6.3.2.9 Keyword abbreviation

a) Each of the keywords in the syntax may be abbreviated by the first one or more letters, so long as no ambiguity results. For example,

   ```
   ( class, operation ) create
   ```
   may be abbreviated as
   ```
   ( cl, o ) create.
   ```
b) If the keyword is a phrase, it may be abbreviated using the first letter(s) of each word in the phrase and omitting intervening spaces. For example,
   ```
   name ( uniqueness constraint 1, subclass responsibility )
   ```
   may be abbreviated as
   ```
   name ( uc1, sr ).
   ```
   For another example,
   ```
   feature(s) ( cardinality P )
   ```
   may be abbreviated as
   ```
   feature(s) ( ca P ).
   ```
c) If the keyword is a hyphenated phrase, it may be abbreviated using the first letter of each word in the phrase and omitting the hyphen. For example,
   ```
   feature ( multi-valued )
   ```
   may be abbreviated as
   ```
   feature ( mv ).
   ```
   Figure 49 illustrates the use of keyword abbreviations.

6.3.3 Property rules

6.3.3.1 Naming/signature

a) A property of a given signature may appear in more than one class in a view.
b) No two properties with the same signature may appear in the same class.

A more complete explanation of the signature uniqueness requirements is given in 7.5.3.

6.3.3.2 Mapping completeness

a) If a property is not mandatory, then `optional` shall be declared.

6.3.3.3 Collection cardinality

a) Specification of *collection cardinality* shall be used only for an attribute or participant property.

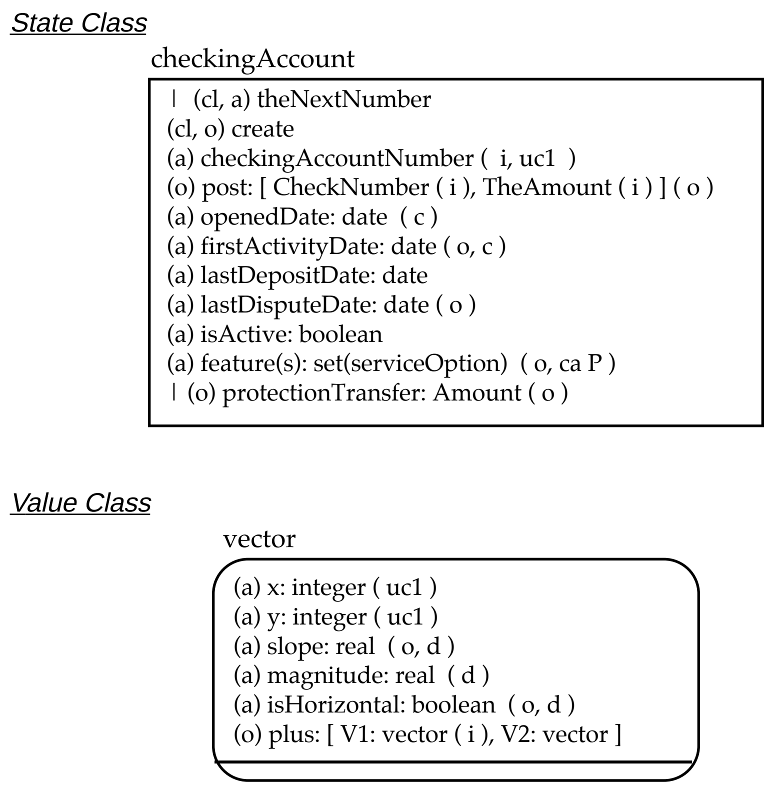

State Class
checkingAccount
| (cl, a) theNextNumber
(cl, o) create
(a) checkingAccountNumber (i, uc1)
(o) post: [CheckNumber (i), TheAmount (i)] (o)
(a) openedDate: date (c)
(a) firstActivityDate: date (o, c)
(a) lastDepositDate: date
(a) lastDisputeDate: date (o)
(a) isActive: boolean
(a) feature(s): set(serviceOption) (o, ca P)
| (o) protectionTransfer: Amount (o)
Value Class
vector
(a) x: integer (uc1)
(a) y: integer (uc1)
(a) slope: real (o, d)
(a) magnitude: real (d)
(a) isHorizontal: boolean (o, d)
(o) plus: [V1: vector (i), V2: vector]

Figure 49—Property declarations using abbreviated keywords

6.3.3.4 Read-only

a) A property declared `read-only` shall cause no state change, i.e., it shall do no updates.

6.3.3.5 Intrinsic

a) If a property is single-valued, mandatory, and constant, then `intrinsic` should be declared (and, because they are redundant, the keywords `constant` and `mandatory` should be omitted from the `SuffixCommaList`).

6.3.3.6 Subclass responsibility

a) A property declared to be a subclass responsibility shall also be declared (as an override) in the appropriate subclasses.

6.3.3.7 Uniqueness constraint

a) A uniqueness constraint shall be declared for a value class in order to use an associative literal (see 5.3.1).

6.3.3.8 Property ordering

a) In any context in which the ordering of properties is important, the order specified in the graphic representation of the properties shall be used.

6.3.4 Property realization

a) The realization of a property shall be stated using the specification language (see Clause 7) in one of three forms depending on the number of arguments, as follows:
   1) `class: Self has property if`$_{def}$ `sentence.`
   2) `class: Self has property: V if`$_{def}$ `sentence.`
   3) `class: Self has property: [ V1, V2, ..., Vn ] if`$_{def}$ `sentence.`

   where

`class`	shall be the class name,
`Self`	shall be a variable name denoting the receiver of the property request (typically `Self` is used),
`property`	shall be the property name (which may be in the form of `p` or `p(s)`),
`V, Vi`	shall be variables denoting the values of the arguments (optionally with argument suffixes), and
`sentence`	shall be a sentence giving the necessary and sufficient conditions for the instance `Self` to map to the argument values or, if there is no argument, for the property to be true for the instance.

b) If a property realization for a collection property `p(s)` is given but none is given for the corresponding multi-valued property `p`, then the realization

   `class: Self has p: V if`$_{def}$ `Self has p(s)..member: V.`

   shall be assumed.

c) If a property realization for a multi-valued property `p` is given but none is given for the corresponding single-valued collection property `p(s)`, then a default realization shall be assumed, as follows:
   1) For a participant property,

      `class: Self has p(s): Vs if`$_{def}$ `Vs is { V where (Self has p: V) }.`

      i.e., the corresponding collection property shall be set-valued.
   2) For all other kinds of property,

      `class: Self has p(s): Vs if`$_{def}$ `Vs is [ V where (Self has p: V) ].`

      i.e., the corresponding collection property shall be list-valued.

   This default implies a cardinality of `mandatory`.

6.4 Attribute

People mentally abstract attributes of a class from the sense that individual instances of the class are described by values in a similar way. Everyone does this abstraction; it is part of common sense.

An *attribute* expresses some characteristic that is generally common to the instances of a class, representing a kind of property associated with a set of real or abstract things (people, objects, places, events, ideas, combinations of things, etc.) that is some characteristic of interest. It is named for the sense in which the instances are described by the values. For example, the `registeredVoter` class acquires a `dateOf-Registration` attribute by abstracting from the individual instances of registered voter being described by specific values of their date of registration. Figure 50 illustrates a state class `checkingAccount` that has three attributes: `balance`, `lastDepositDate`, and `checkingAccountNumber`.

Any class can have attributes, including value classes. Since generalization is based on common attributes, relationships, and operations, having attributes available for value classes strongly affects the generalization and classification of the value classes. Value classes that incorporate unit of measure, such as Fahrenheit and Celsius, can be consolidated into a single class (such as a `temperature` value class) with attributes such as `fahrenheit` and `celsius`. Each is an attribute with a value class (type) of `real`. Similarly, the value class `date` might usefully have attributes such as `europeanFormat` and `americanFormat`. Each is an attribute with a value class (type) of `string`. In Figure 50, the value class `date` is shown with three attributes: `month`, `day`, and `year`.

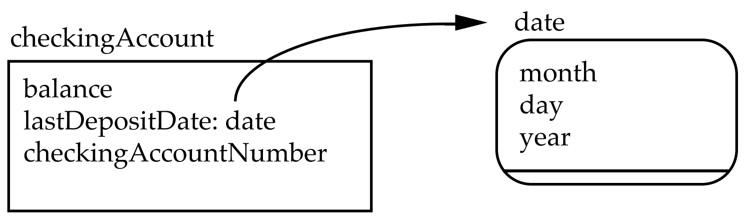


Figure 50—Attribute

An attribute is an interface specification not a realization specification. The declaration of an attribute is not a commitment to its form of realization; there is no implication that an attribute is realized as stored data.

6.4.1 Attribute semantics

The semantics that are common to properties in general are described in 6.3.1. Only the specializations of property semantics applicable to attributes are discussed and illustrated here. Statements that apply generally to attribute as a property are not repeated in the material that follows.

6.4.1.1 Naming

While it is common to use the value class name as the attribute name (as in Figure 55 for `temperature`), this use is not a requirement. An attribute name is a role name for the value class. An attribute *role name* is a name used to clarify the sense of the value class in the context of the class for which it is a property. Figure 50 illustrates the use of an attribute role name, `lastDepositDate`, that differs from the name of its value class, `date`. The name of the value class is not required to be part of the attribute name. For example, the attribute `comfortLevel` of a hotel room could map to the value class `temperature`.

6.4.1.2 Mapping

There are two kinds of attributes, instance-level and class-level. The more common kind is instance-level. An *instance-level attribute* is a mapping from the instances of a class to the instances of a value class. A *class-level attribute* is a mapping from the class itself to the instances of a value class. For either kind of attribute, the value class is also referred to as the *type* of the attribute.

In Figure 50, the attribute `lastDepositDate` is explicitly *typed*, i.e., specified as a mapping to the value class `date`. However, it is sometimes desirable to leave an attribute *untyped*, i.e., not explicitly specify a value class. This is common for the early models at the Integration level (see 8.2). In Figure 50, the attributes `balance`, `checkingAccountNumber`, `month`, `day`, and `year` are all untyped. Even in the case when an attribute is untyped, every attribute value is still an instance of <u>some</u> value class. An untyped attribute simply defers judgment. If, on the other hand, the intent is to specify that an instance of <u>any</u> class is acceptable, the attribute should be typed to the built-in class `any`.

Figure 51 provides a sample instance diagram supporting the view in Figure 50.

6.4.1.3 Mapping completeness

An attribute is assumed to be a total mapping unless it is specified `optional` (meaning that some attribute instances map to no instance of the value class). For example, in Figure 53, the attribute `lastDisputeDate` is declared `optional` because a `checkingAccount` may not (yet) have been involved in any dispute.

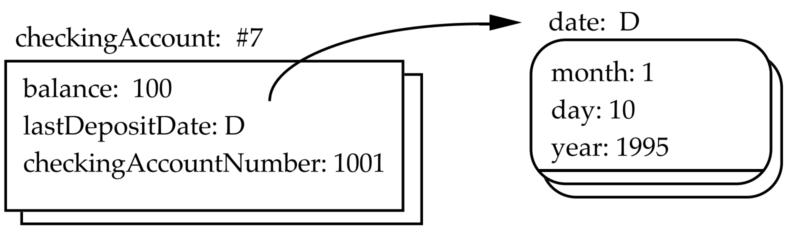


Figure 51—Attribute instance and attribute value

6.4.1.4 Single-valued/multi-valued

An attribute is assumed to be single-valued unless `multi-valued` is specified.

6.4.1.5 Scalar-valued/collection-valued

A value class instance to which an attribute maps need not be atomic; an attribute can map to a collection value class. Such an attribute is *collection-valued*. For example, a checking account may have a `fea-ture(s)` attribute that identifies a set of service options selected for the account (see Figure 53).

6.4.1.6 Collection cardinality

As introduced in 6.3.1.9, when a property maps to a collection class, the number of members in the collection can be constrained to a specified cardinality. Since the optional attribute `feature(s)` in `checkin-gAccount` is collection-valued, the specification as shown in Figure 53 further requires that the attribute, when mapped, prohibit a mapping to the empty collection.

Referential integrity for members that are state class oids is the obligation of the modeler. (This is in contrast to participant properties, where referential integrity is guaranteed by the semantics of relationship.)

6.4.1.7 Constant

Typically, a state class attribute can be updated, i.e., the mapping to the instances of a value class by a given state class instance can change over time.[49] However, in some cases it is necessary to prohibit change to an attribute value. An attribute is specified as *constant* to indicate that its value is unchanging once assigned. For example, the `openedDate` attribute in `checkingAccount` (Figure 53) has been specified `constant`. For a derived attribute, a designation of constant means the same as it does for operation (see 6.6).

An attribute of a value class cannot be updated. Therefore, all value class attributes are inherently constant.

6.4.1.8 Intrinsic

An attribute can be declared to be *intrinsic*, which implies single-valued, a total mapping, and constant. For example, a `checkingAccountNumber` might be considered an intrinsic property of the account; there is only and always a single checking account number for an account and it cannot meaningfully change.

[49]The mapping to the value class instance is updated, not the attribute value itself. For example, if an instance has an attribute with a value of `17` and the attribute is updated to `23`, the mapping of that instance to a value for the attribute is changed—`17` is not made into `23`.

6.4.1.9 Uniqueness constraint

An attribute should be declared to be part of a uniqueness constraint when there is a need to ensure that no two distinct instances of the class agree on the values of all the properties that are named in the uniqueness constraint. For example, in TcCo (see C.7) the attribute `partName` has been declared a uniqueness constraint; no two instances of a part may have the same name.

6.4.1.10 Derived

An attribute whose value is determined based on the values of other properties is a *derived attribute*. Figure 52 illustrates a derived attribute, `available`. The `available` attribute in `creditCardAccount` is derived based on the `limit` (in `creditCardAccount`) and `balance` (in `account`).

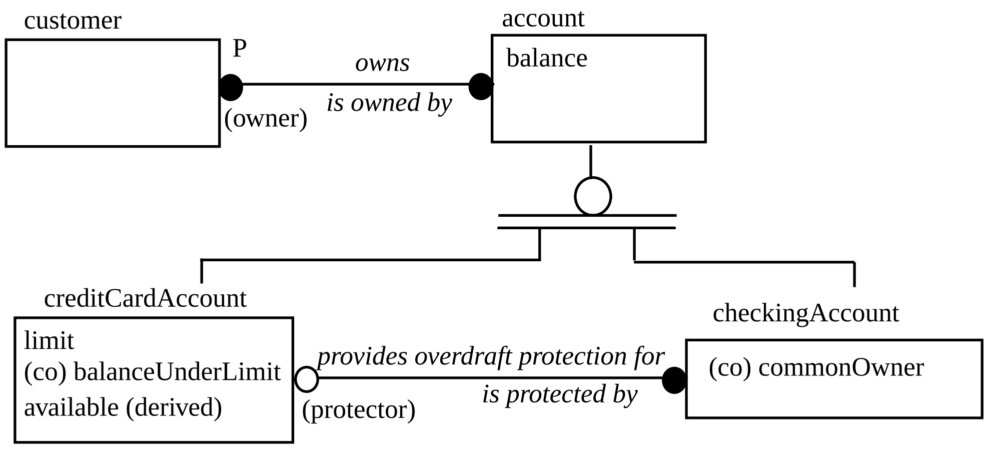
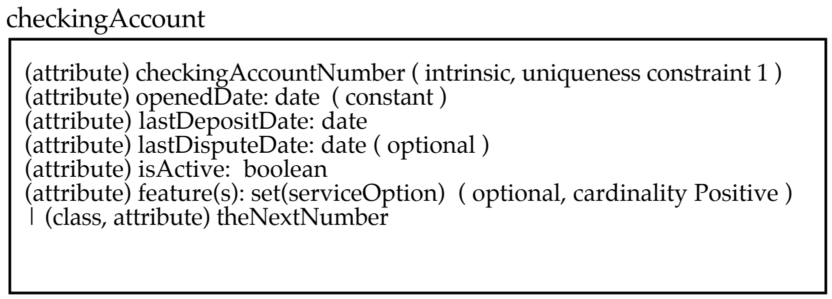


Figure 52—Derived attribute `available` in `creditCardAccount`

6.4.2 Attribute syntax

The syntax that is common to properties in general is described in 6.3.2. Only the specializations of property syntax applicable to attributes are discussed and illustrated here. Statements that apply generally to attributes as properties are not repeated in the material that follows. Figure 53 illustrates attribute syntax.



```
(attribute) checkingAccountNumber ( intrinsic, uniqueness constraint 1 )
(attribute) openedDate: date  ( constant )
(attribute) lastDepositDate: date
(attribute) lastDisputeDate: date ( optional )
(attribute) isActive:  boolean
(attribute) feature(s): set(serviceOption)  ( optional, cardinality Positive )
| (class, attribute) theNextNumber
```

Figure 53—Attribute syntax

6.4.2.1 Visibility annotation

a) The visibility of an attribute may be restricted as protected or private using this standard visibility annotation (see 6.3.2.2) at the beginning of the attribute signature. For example, in Figure 53, `theNextNumber` is a protected attribute.

6.4.2.2 PrefixCommaList clause

a) An attribute shall be designated using the keyword `attribute`. For example, in Figure 53, `checkingAccountNumber` is designated as an attribute.

b) "`class`" shall designate an attribute as a class-level attribute. For example, in Figure 53, `theNextNumber` is a class-level attribute.

c) If "`class`" is not specified, an attribute shall be an instance-level attribute. For example, in Figure 53, the attributes other than `theNextNumber` are instance-level attributes.

6.4.2.3 Property name clause

a) The name of a collection-valued attribute shall end with `(s)`. For example, in Figure 53, the role name `feature(s)` has been given to the attribute that identifies a set of service options for a checking account.

b) The name of a scalar attribute shall be formed in this standard, singular manner. For example, in Figure 53, the name `checkingAccountNumber` has been given to the attribute that uniquely identifies a checking account.

6.4.2.4 Arguments clause

a) An attribute signature shall include no more than one argument, the value class name (see Figure 54).

b) If the attribute maps to a scalar value class, then the `Arguments` clause shall contain the name of the related class. In Figure 53, the signatures for each of the attributes `openedDate`, `lastDepositDate`, and `lastDisputeDate` include an argument `date`, which is the name of the value class to which each attribute maps.

c) If the attribute maps to a collection value class, then the `Arguments` clause shall contain the name of the collection class.

d) If the attribute maps to a collection value class, then the `Arguments` clause may optionally provide the name of the collection's constituent value class as a parameter. For example, in Figure 53, the attribute `feature(s)` (in `checkingAccount`) maps to the collection `set` whose instances are sets of `serviceOption(s)`.

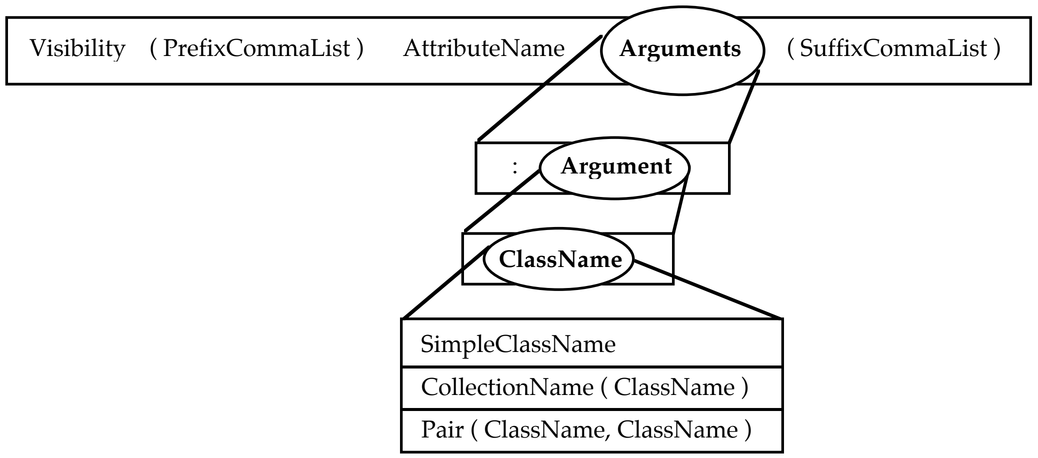


Figure 54—Attribute arguments

6.4.2.5 SuffixCommaList clause

a) The following keyword options shall be valid in the `SuffixCommaList` of an attribute:

1) <u>`mandatory`</u>`|optional`,
2) <u>`single-valued`</u>`|multi-valued`,
3) `cardinality X`,
4) `constant`,
5) `read-only`,
6) `intrinsic`,
7) `uniqueness constraint N`,
8) `subclass responsibility`,
9) `derived`,

where "`|`" denotes alternative keywords and <u>underlining</u> designates the default keyword if none is explicitly specified.

These options are illustrated in the following examples:

— In Figure 53, `lastDisputeDate` is an `optional` attribute.

— In Figure 53, `feature(s)` is an attribute restricted to map to a nonempty collection (when mapped) using the `cardinality P` keyword.

— In Figure 53, `openedDate` is a `constant` attribute.

— In Figure 53, `checkingAccountNumber` is an `intrinsic` attribute.

— In Figure 53, `checkingAccountNumber` is declared as `uniqueness constraint 1` for the state class, meaning that no two instances of a checking account may have the same account number value.

— In Figure 52, `available` is a `derived` attribute.

6.4.2.6 Value class mapping graphic

a) Depiction of the mapping from a class to the value class is commonly omitted from a view diagram.[50] However, when it is useful to depict this mapping, the mapping shall be represented as a line connecting the value class to the related class with an "open triangle" at the related class end, as shown in Figure 55. This figure illustrates

1) A mapping from a state class (`hotel`) to the value class `temperature`, and
2) A mapping from a value class (`temperature`) to the value class `real`.

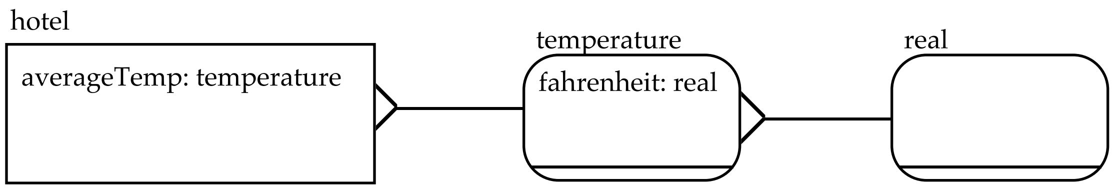


Figure 55—Value class mapping graphic

b) When the mapping is to a collection class, the mapping line may go to either

1) The collection class, as implied above, or

[50] In fact, the graphic representation of value classes is generally omitted from the view diagram.

2) The class of the elements in the collection, with the kind of the collection (e.g., list or set) written on the mapping line beside the class of the elements, as shown in Figure 56.

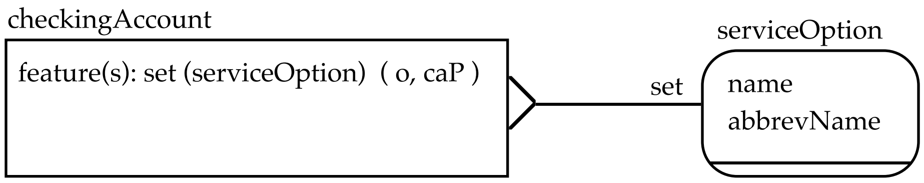


Figure 56—Collection-valued value class mapping graphic

6.4.3 Attribute rules

6.4.3.1 Mapping

a) An attribute may be explicitly associated with an underlying value class—i.e., typed.
b) A collection-valued attribute shall be list-valued unless explicitly typed to some other collection.

6.4.3.2 SuffixCommaList clause

a) An attribute of a value class shall be inherently `constant`.
b) A state class attribute declared `read-only`, for which no overrides are supplied, may never have any value.

6.4.4 Attribute requests

A request may be for the value of an attribute. A request for an attribute has no input arguments and only one output argument (the identity of the related value class instance). A request is issued by one instance to another instance and is the only way to access or alter an attribute value. The detailed discussion of request/instance success and failure combinations in 6.2.3 applies to attributes, and its specifics are not repeated here.

6.4.4.1 Request: get value

a) A “get value” attribute request shall have the form given in 6.2.2.1.

   For example, if `CA` is an instance of `checkingAccount` (illustrated in Figure 53), then its `lastDepositDate` attribute value may be read by issuing the following request:

   ```
   CA has lastDepositDate: TheLastDate
   ```

   or, alternatively, by the following request:

   ```
   TheLastDate is CA..lastDepositDate.
   ```

   1) When the variable `TheLastDate` has no value when the request is made:
      i) If the requested attribute is a total mapping (as is the attribute `lastDepositDate`), or it is an optional attribute (e.g., `lastDisputeDate`) that has a value, then the request variable shall have a value when the request is satisfied, and the request shall be true.
      ii) If the requested attribute is optional and has no value at the time of request, then the request shall fail.
   2) When the variable `TheLastDate` has a value when the request is made:
      i) If the variable’s value is the current value of the requested attribute (e.g., the attribute `lastDepositDate` has the same value as `TheLastDate`), then the request shall be true.
      ii) Otherwise, the request shall be false.

b) The same request form shall hold true for accessing the value of a value class attribute. For example, if `V` is the instance of `vector` (illustrated in Figure 44) that has an `x` attribute value of `100`, then the request (proposition)
```
    V has x: 100
```
is true. The request
```
    V has x: 0
```
is false. For a variable `X` with an unknown value, the request
```
    V has x: X
```
solves for `X`, finding that `X` is `100` (and is true).
For another example, if `T` is the instance of `temperature` that represents 32˚ Fahrenheit, then the request for its `celsius` property value may be stated as
```
    T has celsius: C
```
This request has a solution of `0` (zero); `C` is set to `0`, and the request is true. Furthermore, making the request (asserting the proposition) that
```
    T has celsius: 0
```
is true, but asserting that
```
    T has celsius: 7
```
is false.

c) A request for the value of an `optional` attribute shall be false if the attribute is unmapped.

d) If an `optional`, collection-valued attribute permits mapping to the empty collection and if the attribute is mapped, then a request for its value shall be true, and that collection may be empty.

e) If the `optional`, collection-valued attribute prohibits mapping to the empty collection, a request for its value shall either return a nonempty collection value or be false.
For example, assuming the variable `Features` has no value at the time of request, the request
```
        CA has feature(s): Features
```
sets `Features` to the set of service options for `CA`. Since this attribute is `optional` with the empty collection prohibited, `Features` will not be set to the empty set. Issuing a request with a value in `Features` tests this value against `CA`'s current `feature(s)` value and is true or false as appropriate.

6.4.4.2 Request: set value

a) A "set value" attribute request shall apply only to single-valued (scalar or collection) attributes.

b) A "set value" attribute request shall have the form given in 6.2.2.2.
For example, if `CA` is an instance of `checkingAccount`, then its `lastDepositDate` attribute value may be set to a value by issuing the request:
```
    CA has lastDepositDate:= NewDate
```
where `NewDate` is an instance of `date`.

6.4.4.3 Request: unset value

a) An "unset value" attribute request shall apply only to single-valued (scalar or collection) attributes.

b) An "unset value" attribute request shall have the form given in 6.2.2.3.
For example, if `CA` is an instance of `checkingAccount`, then its `lastDisputeDate` attribute value may be removed by the request

```
CA has lastDisputeDate:!= OldDate.
```

   1) If the variable `OldDate` has no value at the time of request, then
      — The variable `OldDate` is set to the value of `CA`'s `lastDisputeDate,`
      — The value of `CA`'s `lastDisputeDate` is cleared, and
      — The request is true.
   2) If the variable `OldDate` has a value at the time of request,
      i) If the value of `OldDate` is the current value of `lastDisputeDate` for instance `CA`, then
         — The request is true, and
         — The value of `CA`'s `lastDisputeDate` is cleared.
      ii) Otherwise,
         — The request is false, and
         — The value of `CA`'s `lastDisputeDate` is unchanged.
      iii) In either case, the value of the variable `OldDate` is unchanged.

c) The attempt to clear a `mandatory` attribute shall fail and shall be false.

6.4.4.4 Request: insert value

a) For a collection-valued (or multi-valued) attribute only, a single value shall be added to an existing collection using the "insert value" request form given in 6.2.2.4.[51]
For example, if `CA` is an instance of `checkingAccount`, then an addition to its `feature(s)` attribute collection may be made by the request

```
CA has feature(s):+= NewServiceOption
```

where `NewServiceOption` is a variable containing the oid of the one to be added.

6.4.4.5 Request: remove value

a) For a collection-valued (or multi-valued) attribute only, a single element shall be removed from an existing collection using the "remove value" request form given in 6.2.2.5.[52]
For example, if `CA` is an instance of `checkingAccount`, then the removal of a member from its `feature(s)` attribute collection may be made by the request

```
CA has feature(s):-= OldServiceOption
```

where `OldServiceOption` is a variable containing the oid of the one to be removed.

6.4.5 Attribute realization

6.4.5.1 Derivation/representation

a) The realization of a class shall specify whether an attribute is
   1) Part of the representation (i.e., stored), or
   2) Derived (i.e., has a derivation algorithm).

b) If no derivation is stated, the attribute shall be part of the representation.

[51]The value class is not actually "updated" by this request but rather the mapping is changed to the instance of the collection that has the resulting collection of values. The same is true for the "remove" request.

[52]The intended semantics of a multi-valued property are that if a value is added, then it should be there for a subsequent *get*, and that if it is removed, it should not be there for a subsequent *get*.

c) For each derived attribute, the realization shall specify the derivation using the specification language (see Clause 7) in the form given for a property realization with one argument (see 6.3.4).

Figure 57 shows the realization of the derived attribute, `available`, in `creditCardAccount` (Figure 52).

```
creditCardAccount: Self has available: A if_def
       A is Self..limit - Self..balance.
```

Figure 57—Realization of `creditCardAccount` attributes

For the value class `temperature` (in the variation shown in Figure 3), unbeknownst to the requester the representation is in kelvin, and the public `fahrenheit`, `celsius`, and `kelvin` attributes are derived in terms of that representation. The `fahrenheit`, `celsius`, and `kelvin` derivation algorithms are shown in Figure 58.

```
temperature: Self has fahrenheit: F if_def
       C is Self..celsius,
       F is 32 + C * 9/5.
temperature: Self has celsius: C if_def
       K is Self..kelvin,
       C is K - 273.16.
temperature: Self has kelvin: K if_def
       K is Self..rep.
```

Figure 58—Realization of `temperature` value class attributes

The meaning of the celsius derivation may be seen in a natural language paraphrase of the derivation:

the temperature instance Self has a celsius value C if it is the case that

K is Self's kelvin value, and

C is K – 273.16.

In the value class `vector` (shown in Figure 72), the chosen representation is the combination of the x, y, z coordinates. Figure 59 shows an example of the specification language for the derived attribute `magnitude` of the `vector` value class. The specifics of the syntax are explained in Clause 7.

```
vector: Self has magnitude: M if_def
       M is ( Self..x ^2 + Self..y ^2 ) ^0.5.
```

Figure 59—Realization of `magnitude` attributes

6.4.5.2 Interaction with constraints

a) A realization may specify an interaction between responsibilities. For example, the realization in Figure 57 illustrates the interaction between two responsibilities. In this case, the successful derivation of `available` relies on the `balanceUnderLimit` constraint to ensure that `balance` never exceeds `limit`, thereby avoiding a negative value for `available`.

6.5 Participant property

A *relationship participant property* (simply, *participant property*) is a property of a state class that reflects that class's knowledge of a relationship in which instances of the class participate. When a relationship exists between two state classes, each class contains participant properties for that relationship.

Participant properties arise from relationships and are based on instance identity. For every relationship, there are at least two participant properties, one in each of the related classes. A scalar participant property is a mapping from a state class instance to an instance of a related (not necessarily distinct) state class. A collection-valued participant property is a mapping from a state class instance to a collection of instances of a related (not necessarily distinct) state class. For every collection-valued participant property, there is a corresponding multi-valued scalar participant property, and vice versa.

If state class `c1` is related to state class `c3` then, by virtue of the relationship, there is a participant property `c3` in `c1` and a participant property `c1` in `c3` (see Figure 63). For example, if every transaction is incurred by one account, then the knowledge of the transaction's `account` is reflected by a participant property of `transaction`, and the knowledge of the account's transactions is reflected by participant properties of `account`. If an account can be owned by many customers, and a customer can own many accounts, then the knowledge of the account's customers is reflected by participant properties of `account`, and the knowledge of the customer's accounts is reflected by participant properties of `customer` (see Figure 61).

A participant property is an interface specification not a realization specification. The declaration of a participant property is not a commitment to its form of realization; there is no bias to implement the participant property by index, list, or other stored data.

6.5.1 Participant property semantics

The semantics that are common to properties in general are described in 6.3.1. Only the specializations of property semantics applicable to participant properties are discussed and illustrated here. Statements that apply generally to participant property as a property are not repeated in the material that follows.

6.5.1.1 Naming

The name of a participant property reflects the name of the class at the other end of the relationship. If the related class has been given a role name in the relationship, the role name, rather than the class name, is used as the basis for the participant property name. (See 6.5.3.1 for naming rules.)

6.5.1.2 Mapping

A participant property is a mapping from an instance of a state class to an instance of a state class.

6.5.1.3 Mapping completeness

The mapping completeness of a participant property is dictated by the cardinality of the relationship that it reflects (see 5.5). If the relationship cardinality prohibits the case where there is no related instance, then the scalar participant property's mapping is *total*. Such a participant property is declared to be `mandatory` as part of the relationship syntax. For example, in Figure 61, the participant property `owner` in `account` is `mandatory` because each account must be owned by at least one owner (customer).

If the relationship cardinality permits the case of no related instance, then the scalar participant property's mapping is *partial*. Such a participant property is specified as `optional` as part of the relationship syntax. For example, in Figure 61, the participant property `account` in `customer` is `optional` because not every customer owns an account.

6.5.1.4 Single-valued/multi-valued

If the relationship cardinality specification allows more than one related instance or is a cardinality specification of `Z`, then it specifies a multi-valued participant property that is scalar. The participant property `transaction` in `account` is *multi-valued* because each account may incur many transactions.

If the cardinality specification allows at most one related instance and is not a cardinality specification of `Z`, the participant property is *single-valued*.[53] For example, in Figure 61, the participant property `account` in `transaction` is single-valued because every transaction is incurred by at most one account.

6.5.1.5 Scalar-valued/collection-valued

If the relationship cardinality specification allows more than one related instance or is a cardinality specification of `Z`, then it specifies a collection-valued participant property that is single-valued. For example, in Figure 61, the participant property `transaction(s)` in `account` is *collection-valued* because each account may incur many transactions and is *single-valued* because there is only one collection.

If the cardinality specification allows at most one related instance and is not a cardinality specification of `Z`, the participant property is *scalar-valued* (simply, *scalar*).[54] For example, in Figure 61, the participant property `account` in `transaction` is scalar because every transaction is incurred by at most one account.

6.5.1.6 Collection cardinality

As introduced in 6.3.1.9, when a property maps to a collection class, the values of the property can be constrained to a specific cardinality by a declaration of its *collection cardinality.* A collection-valued participant property's cardinality is generally specified as a part of the relationship syntax.

For example, in Figure 61, the property `account(s)` in `customer` is `optional`. The basic sense of this relationship cardinality would permit mapping to the empty collection. If the intent is to prohibit mapping to the empty collection when mapped then the participant property syntax should specify `cardinality Positive` as well. If no collection cardinality is specified, a collection-valued property may map to a collection of any number of members, including zero (the empty collection).

6.5.1.7 Constant

A participant property is *constant* if it is unchanging once the relationship has been formed. For example, in TcCo (see C.7) the participant property `component` of `structureItem` is constant[55] since a `structureItem` cannot be related to a different `component (part)` and still be the same `structureItem`. By contrast, the participant property `standardVendor` of `boughtPart` is not constant since this standard vendor for a `boughtPart` may change over time.

6.5.1.8 Intrinsic

A participant property is an *intrinsic* participant property of the class when it reflects an intrinsic relationship, i.e., the relationship is single-valued, a total mapping, and constant (see 5.5). In Figure 61, the `account` reflected in the dependent state class `transaction` is an intrinsic participant property of `transaction`. This states that for each `transaction` there is at most one `account` (i.e., single-valued). Furthermore, for every `transaction`, there is always a related `account` (i.e., a total mapping).

[53]To support a modeling style that represents all relationships consistently as collection-valued, constrained to a specified cardinality, an alternative form of "at most one" is provided (see 5.5). In the discussions here, the scalar form will be assumed for cardinalities of "at most one" and "exactly one."

[54]See footnote 53.

[55]In Figure C.21, the participant property `component` of `structureItem` is specified as `intrinsic`, which subsumes `constant`.

Finally, it makes no sense to change a `transaction` to a different `account` because that would change the very nature of the `transaction` (i.e., constant).

6.5.1.9 Uniqueness constraint

A participant property should be declared to be part of a uniqueness constraint when there is a need to ensure that no two distinct instances of the class agree on the values of all the properties that are named in the uniqueness constraint. For example, in TcCo (see C.7) the participant properties `assembly` and `component` in `structureItem` have been declared a uniqueness constraint: no two instances of a `structureItem` may have the same combination of `assembly` and `component`. The uniqueness constraint in `boughtPart` illustrates a uniqueness constraint declaration that includes both an attribute and a participant property; no `boughtPart` may duplicate the combination of a `vendor`'s identity along with a `vendorPartId` value.

6.5.1.10 Derived

A participant property whose value is determined based on the values of other properties is said to be a derived participant property. Figure 60 is a view of product offerings, which can be either product items (e.g., a can of tomato sauce) or packaged items that contain product items or other packaged items (e.g., a box of pizza mix). It would be nice to be able to refer to all ingredients (and their properties) whether directly contained or included via a packing chain. The derived participant property `ingredient(s)` in `productOffering` provides this facility for reasoning about ingredients. These derivations presume the presence of a constraint ensuring that no `productOffering` includes itself.

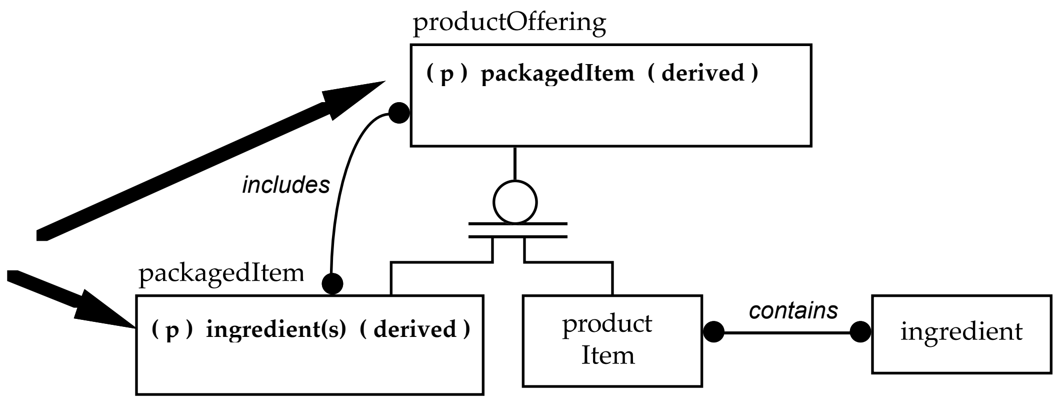


Figure 60—Derived participant properties

The realization of the derivation of this participant property can be expressed in the specification language as

```
packagedItem: Self has ingredients(s): Is if_def
   Is is [ I where
      (  Self has productOffering(s)..member: PO,
         PO has ingredient(s)..member: I)].
```

The realization of the multi-valued participant property `ingredient` is

```
packagedItem: Self has ingredient: I if_def
     Self has productOffering..ingredient: I.
```

6.5.2 Participant property syntax

The syntax that is common to properties in general is described in 6.3.2. Only the specializations of property syntax applicable to participant properties are discussed and illustrated here. Statements that apply generally to participant properties as properties are not repeated in the material that follows. Figure 61 illustrates participant property syntax.

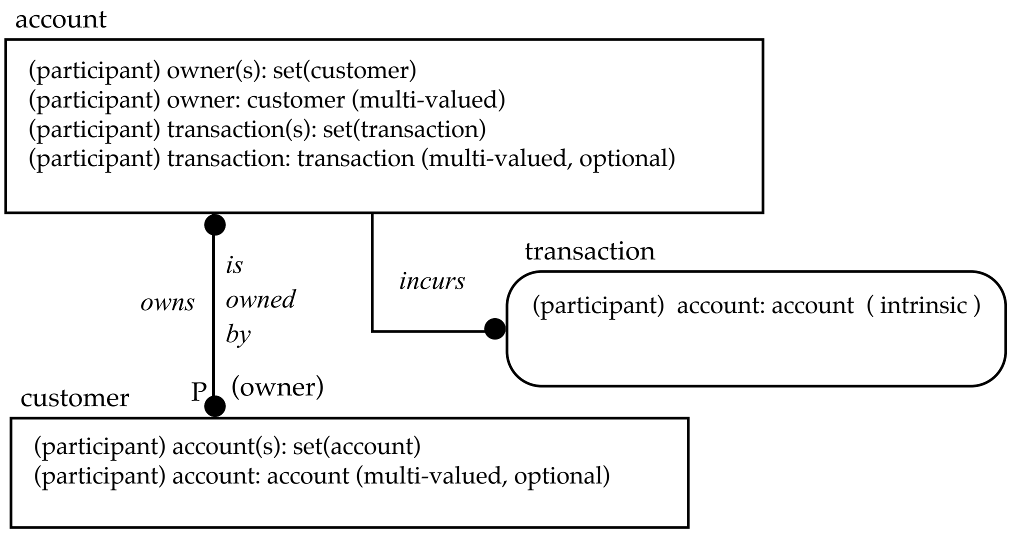

account
(participant) owner(s): set(customer)
(participant) owner: customer (multi-valued)
(participant) transaction(s): set(transaction)
(participant) transaction: transaction (multi-valued, optional)
owns
is owned by
incurs
transaction
(participant) account: account (intrinsic)
customer
P
(owner)
(participant) account(s): set(account)
(participant) account: account (multi-valued, optional)

Figure 61—Participant properties

6.5.2.1 PrefixCommaList clause

a) A participant property shall be designated using the keyword `participant`. For example, in Figure 61, `account` (in `transaction`) is designated as a participant property.

b) The keyword `class` shall not be applicable to participant properties.

6.5.2.2 Property name clause

a) A participant property name may optionally be placed inside the rectangle for the state class for which it is a property. When shown, it shall be displayed as illustrated in Figure 61.

   A participant property is present in a state class for each relationship in which the state class participates, but it need not be listed inside the class box (because doing so is graphically redundant with the information on the relationship arcs and the cardinality annotations). A participant property is normally displayed inside the class rectangle only when needed to state a constraint, like uniqueness, that could not otherwise be stated.[56]

b) If a role name has been designated for the related class, the participant property name shall reflect that role name. For example, in Figure 61,

   1) The role name `owner` has been given to `customer`'s participation in the `customer/ account` relationship.

   2) For the collection-valued form, the participant property name in `account` has been given the name `owner(s)`.

[56]An illustration of the display of a participant property with a declared uniqueness constraint can be found in Figure C.21.

3) For the multi-valued form, the participant property has been given the name `owner`.
4) No role name has been given to `account`'s participation in the `customer/account` relationship; accordingly, the collection-valued form of the participant property in `customer` has been given the name `account(s)`, corresponding to the class name suffixed with `(s)`.
5) Because it is scalar, and single-valued, there is no collection-valued form, and the participant property name `account` in `transaction` is formed without this suffix.

6.5.2.3 Arguments clause

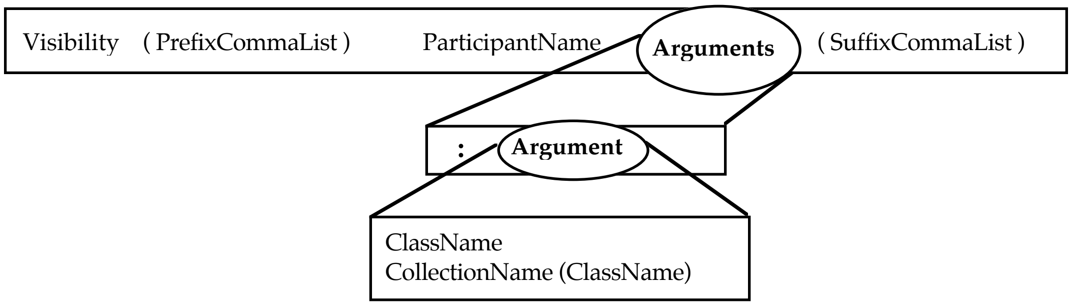


Figure 62—Participant arguments

a) A participant property signature shall have exactly one argument.
b) If the participant property is scalar, then the `Arguments` clause shall contain the name of the related state class. For example, in Figure 61, `account: account` (in `transaction`) illustrates a participant property that is scalar.
c) If the participant property is collection-valued, then the `Arguments` clause contains the name of a collection class, with the name of the related class as its parameter, as shown in Figure 62. For example, Figure 61 illustrates the syntax of the following collection-valued participant properties:

`account(s): set(account)`	(in `customer`)
`owner(s): set(customer)`	(in `account`)
`transaction(s): set(transaction)`	(in `account`)

6.5.2.4 SuffixCommaList clause

a) The following keyword options shall be valid in the `SuffixCommaList` of a participant property:
   1) <u>`mandatory`</u> `| optional`,
   2) <u>`single-valued`</u> `| multi-valued`,
   3) `cardinality X`,
   4) `constant`,
   5) `read-only`,
   6) `intrinsic`,
   7) `uniqueness constraint N`,
   8) `subclass responsibility`,
   9) `derived`,

   where "`|`" denotes alternative keywords and <u>underlining</u> designates the default keyword if none is explicitly specified.

   These are illustrated in the following examples:

   — In Figure 61, `account` (in `customer`) and `transaction` (in `account`) are `optional` participant properties.

   — In Figure 61, `account` (in `transaction`) is an `intrinsic` participant property.

6.5.3 Participant property rules

6.5.3.1 Naming

a) Assignment of participant property names is a built-in feature of the language (i.e., accomplished by the assignment of relationship role names); it shall not be overridden.

b) A scalar participant property name shall be the role name specified for the related class or, if there is no role name, the name of the related class.

c) A collection participant property name shall be the corresponding scalar name suffixed with `(s)`.

d) If the relationship mapping is multi-valued, there shall be both
   1) A scalar participant property, and
   2) A collection participant property—where the collection participant property name shall be the scalar participant property name suffixed with `(s)`.

   For example, in Figure 63,
   — Each `c1` is related to multiple `c3(s)`, where the role name by which `c1` knows each `c3` is `r1`, and
   — Each `c3` is related to at most one `c1`, where there is no role name, and
   — Each `c1` is related to multiple `c2(s)`, where there is no role name, and
   — Each `c2` is related to at most one `c1`, where `r2` is the role name by which `c2` knows `c1`.

   So
   — The mapping from `c1` to `c2` is multi-valued,
   — The mapping from `c2` to `c1` is single-valued,
   — The mapping from `c1` to `c3` is multi-valued,
   — The mapping from `c3` to `c1` is single-valued,

   and therefore,
   — `c1` has four participant properties named, respectively, `r1`, `r1(s)`, `c2`, and `c2(s)`:
   — `r1` is a multi-valued, scalar property
   — `r1(s)` is a single-valued, collection property
   — `c2` is a multi-valued, scalar property
   — `c2(s)` is a single-valued, collection property.
   — `c2` has one participant property named `r2`:
   — `r2` is a single-valued, scalar property.
   — `c3` has one participant property named `c1`:
   — `c1` is a single-valued, scalar property.

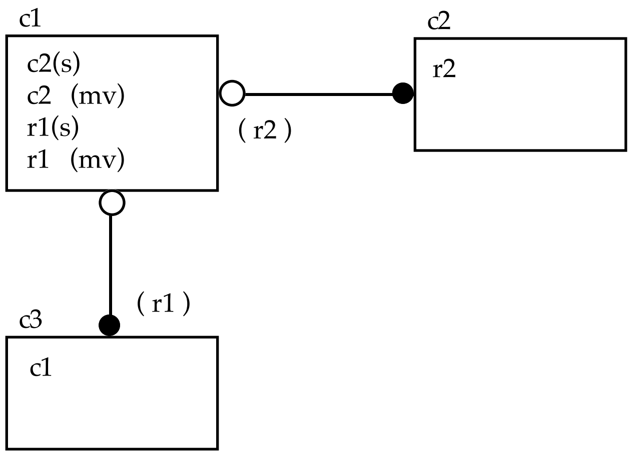


Figure 63—Participant property names

Furthermore,

- For each instance of `c1`:
  - The value of property `c2(s)` is the collection of the identity(s) of the related `c2` instance(s)
  - The value of property `r1(s)` is the collection of the identity(s) of the related `c3` instance(s)
  - Each value of property `c2` is the identity of a related `c2` instance
  - Each value of property `r1` is the identity of a related `c3` instance
- For each instance of `c2`:
  - The value of property `r2` is the identity of the related `c1` instance
- For each instance of `c3`:
  - The value of property `c1` is the identity of the related `c1` instance

6.5.3.2 Mapping

a) A collection-valued participant property shall be set-valued unless explicitly typed to some other collection.

6.5.3.3 Mapping completeness

a) The mapping completeness keyword syntax of a participant property shall always be a reflection of the relationship cardinality specification, and possibly a further refinement, as presented in Table 8.
b) The relationship graphic and participant property keyword combinations shown in Table 8 shall be the only allowed combinations.

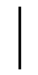
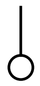
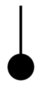
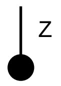
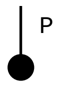
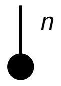
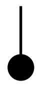
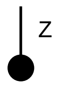
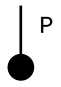
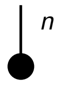
6.5.3.4 PrefixCommaList clause

a) All relationships, and therefore all participant properties, shall be instance-level. Therefore, the keyword `class` shall not be included in the `PrefixCommaList` for a participant property.

6.5.3.5 SuffixCommaList clause

a) A participant property declared `read-only`, for which no overrides are supplied, may never have any value.
b) If a participant property is declared `derived`, its corresponding participant property (i.e., representing the inverse relationship) shall also be `derived`.

Table 8—Summary of relationship graphic syntax and participant property keywords

Relationship graphic syntax	Signature keywords		Specification	
	for mapping completeness	for collection cardinality		
Single-valued, Scalar Participant Property:				
	m	(NA)	exactly 1	i.e., always mapped.
	o	(NA)	not more than 1	i.e., zero (unmapped) or 1.
Single-valued, Collection Participant Property:				
	m		always mapped	...with mapping to the empty collection allowed.
	o		may be unmapped	...and, when mapped, may map to the empty collection.
	o	ca P	may be unmapped	...and, when mapped, may not map to the empty collection (“positive” cardinality).
	m	ca Z	always mapped	...and may map only to either a collection of 1 or the empty collection.
	o	ca 1	may be unmapped	...and, when mapped, may map only to a collection of 1.
	o	ca Z	may be unmapped	...and, when mapped, may map only to either a collection of 1 or the empty collection.
	m	ca P	always mapped	...and may not map to the empty collection.
	m	ca N	always mapped	...and must map to a collection of exactly N (where N is a non-zero, unsigned integer).
	o	ca N	may be unmapped	...and, when mapped, must map to a collection of exactly N.
Multi-valued, Scalar Participant Property:				
	o		may be unmapped	...and, when mapped, may map to any number.
	o	ca Z	may be unmapped	...and, when mapped, may map to at most 1.
	m	ca P	always mapped	...to at least 1.
	m	ca N	always mapped	...to exactly N (where N is a non-zero, unsigned integer).

6.5.4 Participant property requests

A request may be for the value of a participant property. A request for a participant property has no input arguments and only one output argument (the identity of the related state class instance). A request is issued by one instance to another instance and is the only way to access or alter a participant property value. The detailed discussion of request/instance success and failure combinations in 6.2.3 applies to participant properties, and its specifics are not repeated here. In all cases the semantics are such that any corresponding updates to the participant property in the related instance are done as a part of the request.

a) For a participant property, the semantics of get, set, unset, add, and remove requests shall be the same as for attributes (see 6.4.4).
For example, if `CA` is an instance of `account` (illustrated in Figure 61) with two accounts (`A11` and `A22`), then after the following requests:
```
        CA has account(s):+= A33
        CA has account(s):+= A44
```
`CA` owns four accounts: `A11`, `A22`, `A33`, and `A44`.
Continuing the example, if `CA` is an instance of `checkingAccount` now owning four accounts (`A11`, `A22`, `A33`, and `A44`), then after the following requests:
```
        CA has account(s):-= A22
        CA has account(s):-= A44
```
`CA` owns two accounts: `A11` and `A33`.

6.5.5 Participant property realization

a) The realization of nonderived participant properties is built-in to the semantics of the specification language and need not be specified by the modeler.
b) If desired, the built-in realization may be overridden or a derived participant property derivation rule may be stated using the form given for property realizations with one argument (see 6.3.4).

6.6 Operation

The *operations* of a class specify the behavior of its instances.[57] People abstract the operations of a class from what individual instances of the class are able to do or have done to them. From the facts that individual insurance policies accept claims against them, that savings accounts have withdrawals, and that restaurants take reservations, we abstract the operations of the classes: the `insurancePolicy` class has an `acceptClaim` operation, the `savingsAccount` class has a `makeWithdrawal` operation, and the `restaurant` class has a `takeReservation` operation.

An *operation* is an abstraction of what an instance does. An attribute or participant property is an abstraction of what an instance knows. The two are intimately related. The insurance policy that can accept a claim knows what the policy covers and in what amounts. It uses that knowledge to accept the claim. The savings account knows its balance and knows the identity of the owner of the account. It uses that knowledge to do the withdrawal.

Operations can perform input and output, and can change attribute and participant property values. Operations can be stated using any property operator syntax, i.e., read, set, unset, insert, and remove syntax. Within the model, operations are the only way to use values and effect change; there are no free-floating processes, activities, functions, or procedures. Every operation is associated with one class and is thought of as a responsibility of that class. No operations are the joint responsibility of multiple classes.

[57]Because of this dynamic aspect, an operation may also be called an *active property*. In the literature, there has been a distinction in this terminology. *Operation* came from ODMG-93 [B11] as meaning something with multiple arguments; it included both mutable and immutable classes so it could be read-only. *Active property* was originally intended to mean something that does something and included things that did not have arguments. However, in this document the terms are used interchangeably.

A request to an operation causes a method to be run, i.e., executing or "evaluating" it. An operation may require input arguments and may set output arguments. The value of the output argument is referred to as the "value of the operation" or the "solution."

6.6.1 Operation semantics

The semantics that are common to properties in general are described in 6.3.1. Only the specializations of property semantics applicable to operations are discussed and illustrated here. Statements that apply generally to operation as a property are not repeated in the material that follows.

6.6.1.1 Naming

An operation is given a name that is typically a verb or verb phrase. The name should be chosen to reflect the sense of the activity that is represented by the operation. For example, `acceptClaim`, `makeWithdrawal`, and `takeReservation` would be representative operation names for an `insurancePolicy`, a `savingsAccount`, and a `restaurant`, respectively.

6.6.1.2 Mapping

There are two kinds of operations, instance-level and class-level. The more common kind is instance-level. An *instance-level operation* is a mapping from the (cross product of the) instances of the class and the instances of the input argument types to the (cross product of the) instances of the other (output) argument types. A *class-level operation* is a mapping from the (cross product of the) class itself and the instances of the input argument types to the (cross product of the) instances of the other (output) argument types.

An intuitive example is the operation `plus`, which adds two integers, and can be visualized as the mapping table shown partially in Figure 64 to illustrate the instance `1` and its output argument response for each possible input argument. In this case, the mapping is from the cross product of the instance (`1`) and the `Addend` into the `Sum`.

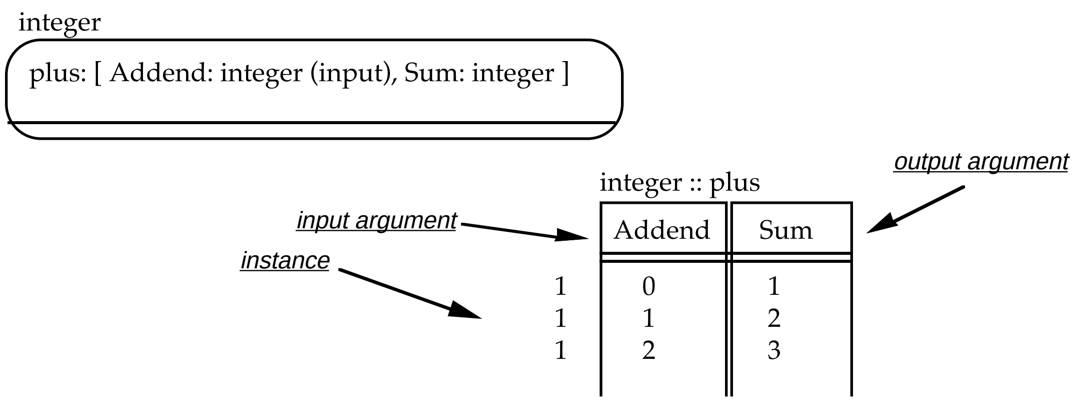


Figure 64—Operation mapping table

In another example, the operation `plus` in the class `vector` (Figure 65) adds two vectors, yielding a new result vector. If

```
VH is vector with ( x: 100, y: 0 ),
VV is vector with ( x: 0, y: 100 )
```

then

```
            VH has plus: [ VV, V ],
            V == vector with ( x:100, y:100 ).
```

Any class can have operations, including value classes.

6.6.1.3 Mapping completeness

An operation is *total* when it gives a solution or produces a response for all instances and valid input argument values. The `plus` operation in `vector` is total because any two vectors can be added to yield a unique result `vector`. The keyword `mandatory` designates a total mapping (see 6.6.3.2).

An operation is *partial* when it may have no meaning for some instances, i.e., it may not give a solution or produce a response. This concept can be thought of like a mathematical *partial function*, e.g., `divide` is a partial function because divide by zero is not specified. `slope` (of a `vector`) is partial because a vertical vector has no slope. In a business example, a `refinance` operation would be partial because a mortgage might be already paid off and thus the notion of refinance would have no meaning. In a second business example, the `makeWithdrawal` operation in `savingsAccount` is not total. A withdrawal can occur only if there is enough in the account to cover the withdrawal. The keyword `optional` designates a partial mapping (see 6.6.3.2). For example, in Figure 44 the operations `post` and `protectionTransfer` (in `checkingAccount`) have been specified as `optional` operations.

6.6.1.4 Single-valued/multi-valued

An operation is single-valued unless declared `multi-valued`. For the class `real`, a multi-valued, scalar `squareRoot` operation and/or a single-valued, collection `squareRoot(s)` operation could be defined. With such definitions, the following would all be true in the specification language:

```
        4 has squareRoot: 2
        4 has squareRoot: -2
        4 has squareRoot(s): { 2, -2 }
```

6.6.1.5 Scalar-valued/collection-valued

An operation that returns a single value (such as `add`) or returns a single value at a time (such as `squareRoot`) is *scalar.* For collection-valued operations where several values are correct, such as `squareRoot(s)` above, a successful request returns all truthful solutions in a single collection.

6.6.1.6 Constant

An operation is a *constant* if the same set of input values always yields the same set of output values.

6.6.1.7 Read-only

An operation is *read-only* if it does not change any attribute or participant property. This includes private as well as public and protected properties.[58] Thus, an operation declared `read-only` may request only `read-only` responsibilities.

6.6.1.8 Intrinsic

An operation can be declared to be *intrinsic*, which implies single-valued, constant, and total. The declaration of intrinsic for an operation means that it is a constant and always returns a response (total) that has a single value (scalar).

[58]This precludes caching the result of a constant read-only derivation.

6.6.2 Operation syntax

The syntax that is common to properties in general is described in 6.3.2. Only the specializations of property syntax applicable to operations are discussed and illustrated here. Statements that apply generally to operations as properties are not repeated in the material that follows. Figure 65 illustrates operation syntax.

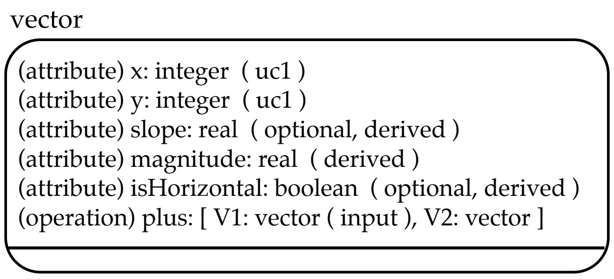

vector
(attribute) x: integer (uc1)
(attribute) y: integer (uc1)
(attribute) slope: real (optional, derived)
(attribute) magnitude: real (derived)
(attribute) isHorizontal: boolean (optional, derived)
(operation) plus: [V1: vector (input), V2: vector]

Figure 65—Operations

6.6.2.1 Visibility annotation

a) The visibility of an operation may be restricted as protected or private using this standard visibility annotation at the beginning of the operation signature (see 6.3.2.2). For example, in Figure 65 the operation `plus` (in `vector`) is public. In Figure 44, the operation `post` (in `checkingAccount`) is public while the operation `protectionTransfer` has been specified as a protected operation.

6.6.2.2 PrefixCommaList clause

a) An operation shall be designated using the keyword `operation`. For example, in Figure 65, `plus` is designated as an operation.
b) “`class`” shall designate an operation as a class-level operation.
c) If “`class`” is not specified, an operation shall be an instance-level operation. For example, in Figure 44, `create` is a class-level operation property. The remaining operations in Figure 44 are instance-level operations.

6.6.2.3 Property name clause

a) The signature shall include the operation name. For example, in Figure 65 the name `plus` has been given to the operation that adds two vectors.

6.6.2.4 Arguments clause

a) An operation may have any number of arguments and possibly none. For example, the operation `close` may be requested of an instance of a class `file`. The operation `delete` may be requested of an instance of an `account` that is not active.
b) A class (either state or value) may be specified for each argument.
c) An argument value shall be an instance of the argument’s declared class; that class is called the *type* of the argument.
d) If no type is declared for an argument, then that argument shall accept any instance.[59]

[59]Of course, only the instances of a few classes will give the results expected. Typing the arguments helps one to reason about the property. On the other hand, insisting on typing too soon during model development is counter-productive. The conclusion is that both typed and untyped arguments need to be supported. See Clause 7 for a discussion of typing.

6.6.2.5 ArgSuffixCommaList clause

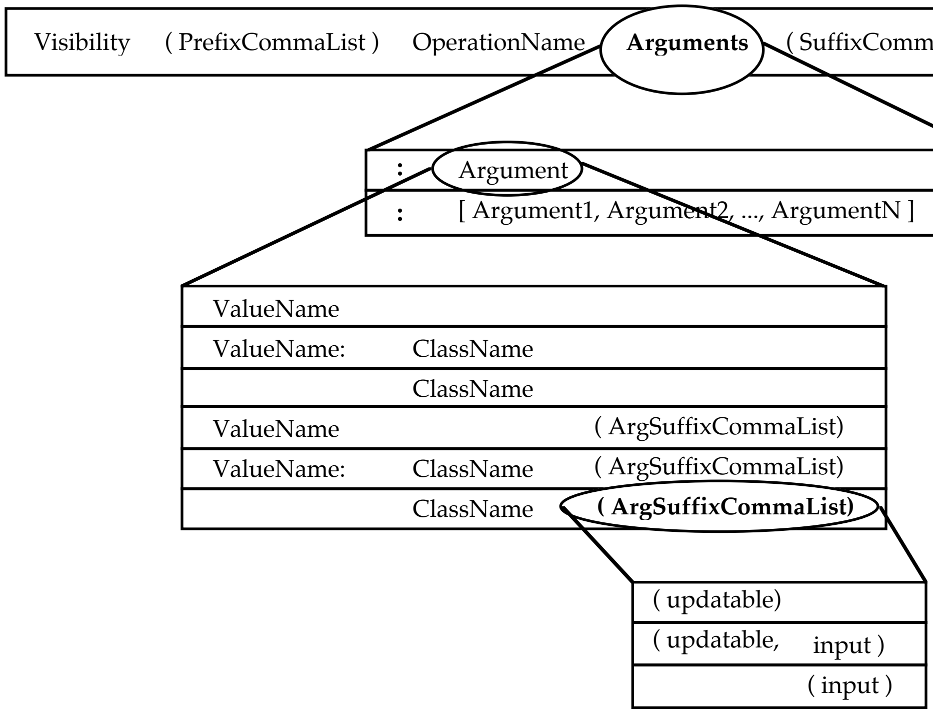


Figure 66—Operation arguments

a) The following keyword options shall be valid in the `ArgSuffixCommaList` of an operation argument:
   1) `updatable,`
   2) `input`
b) An operation argument shall be designated as `updatable` if the state class instance whose oid is the argument value may be changed by the operation. Designating an argument `updatable` means that a request may be made to change the state of the instance identified by the argument.
c) An argument <u>not</u> designated as `updatable` means that there shall be no requests made to change the state of the instance identified by the argument.
d) An operation argument shall be designated as `input` if the argument must have a value when the operation is requested.
e) If an argument is <u>not</u> designated `input`, then it need not have a value when the operation is requested.
f) Multiple `ArgSuffixCommaLists` shall be equivalent to a single `ArgSuffixCommaList` with the keywords separated with commas. For example, `(updatable) (input)` is equivalent to `(updatable, input).`

6.6.2.6 SuffixCommaList clause

a) The following keyword options shall be valid in the `SuffixCommaList` of an operation:
   1) <u>`mandatory`</u>|`optional`,
   2) `constant`,
   3) `read-only`,
   4) `intrinsic`,
   5) `subclass responsibility`,

where "|" denotes alternative keywords and underlining designates the default keyword if none is explicitly specified.

Figure 44 illustrates the operations `post` and `protectionTransfer` (in `checkingAccount`), which have been specified as `optional` operations.

6.6.3 Operation rules

6.6.3.1 Arguments clause

Table 9—Argument specification and values before/after invocation

Specification in argument suffix	Value at invocation	If the argument had a value at invocation, did the instance it identifies change?
`updatable`	maybe	maybe
`updatable, input`	yes	maybe
`none`	maybe	no
`input`	yes	no

a) If any argument has a value at invocation, then it shall have the same value at completion.

   This rule refers to the argument value itself. For example, if the argument is `X` and `X` has a value when the operation is requested (e.g., `X` is `10`), then `X` is still `10` when the operation completes.

b) An input argument shall have a value at invocation of an operation.

c) If an operation fails, then all argument values shall be unchanged.

d) If an operation succeeds, then all arguments shall have values.

e) If no value was supplied for an argument on invocation, a successful operation shall set it to a value.

f) If a value for a `noninput` argument is supplied on invocation, the operation shall succeed if the value determined by the operation matches the value supplied; the operation shall fail if the values do not match.

g) If at invocation an `updatable` argument's value is a state class instance, properties of that instance may be changed by the operation, but the argument value itself shall not change.

h) If at invocation a `nonupdatable` argument's value is a state class instance, that instance shall not be changed by the operation. Specifically, at the conclusion of the operation, all of the nonderived participant properties and nonderived attributes of the instance shall be unchanged.

i) Table 9 summarizes the rules for argument specification and values before and after invocation, if the mapping succeeds:

j) Only an argument that is a state class instance may be designated as `updatable`.

k) A collection-valued operation shall be list-valued unless explicitly typed to some other collection.

6.6.3.2 Mapping completeness

a) An operation shall be declared `optional` when it may have no meaning for some instances, i.e., it may not give a solution or produce a response. For example, in Figure 44, the `post` operation has been declared `optional` because a debit posting may only occur if there is enough in the account to cover the amount of the debit.

b) An operation not declared `optional` shall be `mandatory`.

6.6.3.3 Constant

a) For a set of input values, an operation declared `constant` shall always yield the same set of output values.

6.6.3.4 Read-only

a) No operation declared `read-only` shall have any of its arguments declared `updatable`.
b) If any operation argument is declared `updatable,` then the operation may not be declared `read-only`.

6.6.4 Operation requests

A request may be for the application of an operation. A request is issued by one instance to another instance and is the only way to invoke an operation. The detailed discussion of request/instance success and failure combinations in 6.2.3 applies to operations, and its specifics are not repeated here.

6.6.4.1 No-argument request

a) The form of a request for an operation without arguments is given in 6.2.2.6.
For example, assuming that `creditCardAccount` has an operation `closeAccount` that terminates an active account, then the request
```
        CCA has closeAccount
```
terminates the account and is true, if `CCA` is a currently active account. Otherwise, the request is false.

6.6.4.2 Multiargument request

a) The form of a multiargument request shall be:
```
        I has P: [ V1, V2, ... Vn ]
```
where `I` is a specified class instance, `P` is a named operation of the class, and the `V`s are argument values.
b) The argument values of a multiargument request shall appear in the same order as the corresponding arguments in `P`'s signature.
c) The request shall be true if instance `I` may perform the operation. It shall be false if the instance may not perform the operation—i.e.,
1) If the operation has no meaning for the instance to which the request was sent, or
2) If performing the operation would yield an invalid solution.
For example, (referring to Figure 44) if `VH` is an instance of the value class `vector` (having an `x` value of `100` and a `y` value of `0`) and it is sent the request
```
        VH has plus: [ VK, VV ]
```
where
— `VK` is another instance of the value class `vector` (having an `x` value of `0` and a `y` value of `50`), and
— `VV` (the output argument in this `plus` request) has no value when the request is sent,
then
— the request is true, and
— `VV` has the value (identity) of the vector that has an `x` value of `100` and a `y` value of `50`.
On the other hand, if `VV` (the output argument) has a value at the time of the request and that value does not match the value determined by the operation, then the request fails.
d) An operation that is declared `optional` shall be false when a request is sent to an instance for which the requested operation is not applicable or the result would be invalid. For example, (referring to Figure 44) if `CA` is an instance of `checkingAccount` with a balance of `$100` (and no overdraft protection), then posting check 101 for `$75` using the request:

```
CA has post: [ 101, 75 ]
```

is true, with the side-effect of decreasing the balance to `$25`. On the other hand, the request to post check 105 for `$250` using the following request

```
CA has post: [ 105, 250 ]
```

is false (i.e., it fails).

6.6.4.3 Single-argument request

a) The form of a single-argument operation request shall be:

```
I has P: V
```

b) In addition to this "get value" form (`:`), a single-argument request may also be written using other property operators—for example, the "set value" (`:=`) and "unset value" (`:!=`) property operators. These property operators shall be supported for an operation to provide representation-independence.

For example, imagine that a property is designed originally as an attribute with its clients sending "get" and "set" messages. Subsequently, a decision is made to change the property to an operation. It should not be necessary to require all the client requests to change.

6.6.4.4 Update request

a) If an operation involves multiple updates and the operation fails (i.e., is false), then the state of the view shall "roll back" to the state prior to the operation request.

b) If an operation is of the form:

```
forall F: G
```

only `G` shall be allowed to perform updates.

6.6.5 Operation realization

6.6.5.1 Operation specification

a) The realization of an operation shall be stated using the specification language (see Clause 7) in the form given for property realizations with the appropriate number of arguments (see 6.3.4).

Figure 67 shows the realization of the operation `post` in `checkingAccount` (Figure 44). The `post` operation reduces the checking account's balance by the amount of the check being posted, providing the balance remains greater than or equal to zero.

```
checkingAccount: Self has post: [ CheckNbr, CheckAmt ] if_def
       X is Self..balance - CheckAmt,
       X >= 0,
       Self has balance:= X.
```

Figure 67—Realization of `checkingAccount` operation

Figure 68 shows the operation realizations for the `vector` value class that is shown in Figure 72). The `plus` operation, for example, adds two vectors by adding their x, y, z coordinates.

```
vector: Self has plus: [ V1, V2 ] ifdef
         X is Self..x + V1..x,
         Y is Self..y + V1..y,
         Z is Self..z + V1..z,
         V2 is vector with ( x: X, y: Y, z: Z ).
vector: Self has dot: [ V1, P ] ifdef
         P is Self..x * V1..x
         + Self..y * V1..y
         + Self..z * V1..z.
vector: Self has timeScalar: [ S, V2 ] ifdef
         X is Self..x * S,
         Y is Self..y * S,
         Z is Self..z * S,
         V2 is vector with ( x: X, y: Y, z: Z ).
```

Figure 68—Realization of `checkingAccount` operation

6.7 Constraint

In the real world only certain patterns make sense. These patterns are represented by constraints. A *constraint* is a statement of facts that are required to be true in order that the model conform to the real world.[60] A constraint is specified by a logical sentence over property values. If the sentence is true, the constraint is met. If the sentence is false when the constraint is requested, an exception is raised. In other words, the constraint is disallowing something that makes no sense in the real world, screening out things that are not validated by the real world.

In IDEF1X, a constraint is a type of responsibility.[61] One class has the responsibility for knowing if the constraint is met. That constraint may be an instance responsibility or a class responsibility. A constraint can be inherited like any other responsibility. Some constraints, e.g., uniqueness constraints, are specified simply by marking annotations on the constrained property(s); others are explicitly named and stated in the specification language. (See Clause 7 for a full discussion of constraints.)

6.7.1 Constraint semantics

6.7.1.1 Instance-level/class-level

A constraint can be an instance-level constraint or a class-level constraint. A constraint is an instance-level constraint if it is true or false for each instance individually. A constraint is a class-level constraint if it is true or false for the class. An example of an instance-level constraint is that the balance of a credit card account must be below the limit for that account. An example of a class-level constraint is the requirement that the total balance of all accounts not exceed a limit established for the entire set of accounts.

[60]The conditions expressed in the constraint must be true at the completion of a change of state. There may be points during the state change where these conditions are violated, but these are not considered a violation of the constraint.

[61]There are many transaction models, and this version of IDEF1X has chosen not to select one but rather provide only the most basic notions of stating a constraint and providing a way to check it. It is up to the modeler to specify when to check the constraint and what to do when the constraint fails, in whatever way is appropriate to that model.

6.7.1.2 Named constraint

Some constraints are inherent in the modeling constructs, such as value class constraints, uniqueness constraints, and cardinality constraints. Other constraints, referred to as *named constraints*, are named and specified by the user.[62] A *named constraint* is explicitly named, its meaning is stated in natural language, and its realization is written in the specification language.

Figure 69 introduces two named constraints that are explained below. The specification language statement of these constraints is explained later in the discussion of constraint realization (see 6.7.5).

name:	`balanceUnderLimit`
description:	For each instance of a credit card account, the balance must be under the limit.

name:	`commonOwner`
description:	If a credit card account is providing overdraft protection for a checking account, then an owner of the checking account must be an owner of the protecting credit card account.

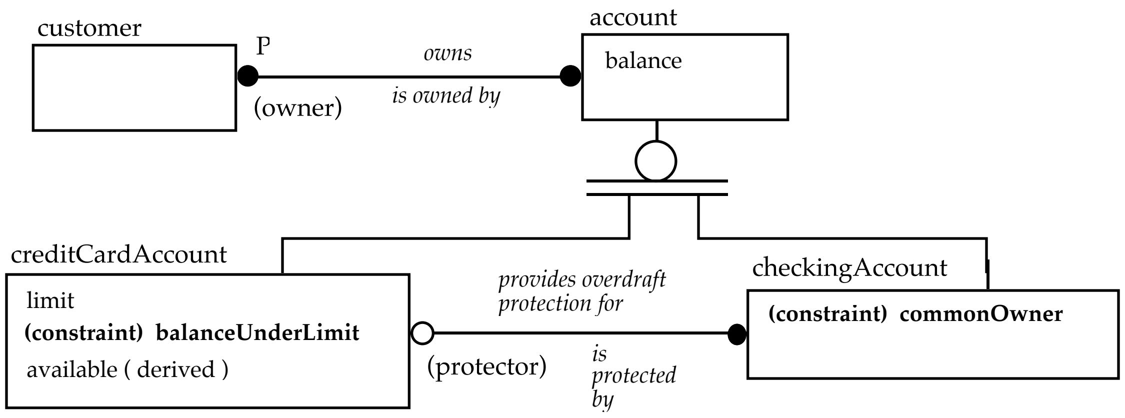


Figure 69—Named constraints

6.7.1.3 Common ancestor constraint

A type of named constraint frequently encountered in modeling occurs when there are two or more paths to an instance from one of its ancestors. Each path is a relationship or generalization or a series of such constructs in which the child or subclass in one is the parent or superclass in the next. For example, if a `hotel` has two related classes, `room` and `tv`, and they each have a common, related class `tvInARoom`, then there are two paths between `hotel` and `tvInARoom`—one through `room` and one through `tv`, as shown in Figure 70.[63]

A *common ancestor constraint* states a restriction on the instances of the ancestor to which an instance of the descendent may relate. Such a constraint typically involves two or more relationship paths to the same ancestor class and states either that a descendent instance must be related to the same ancestor instance through each path, or that it must be related to a different ancestor instance through each path. The `hotel/room/tv` example illustrates the former; the `hotel` that contains the `room` must be the `hotel` that owns the `tv`.

[62]For example, the IDEF1X$_{93}$ metamodel stated 24 constraints that were specific to that model. See [B13], pp. 133–134.
[63]The model in Figure 70 corresponds to the key-style model in Figure 97, which uses foreign keys to state the business rule.

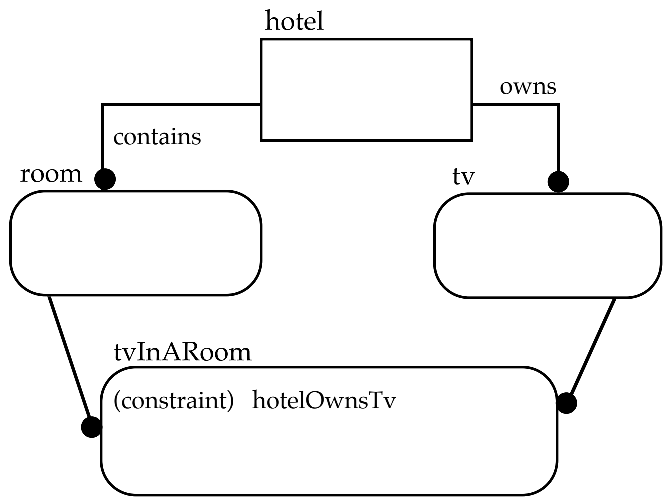


Figure 70—Common ancestor constraint

The common ancestor constraint shown in Figure 70 is stated below:

name:	`hotelOwnsTv`
description:	For each instance of a TV in a room, the hotel that contains the room must be the hotel that owns the TV.

6.7.1.4 Uniqueness constraint

A uniqueness constraint is one of the "unnamed" constraints built into the IDEF1X modeling semantics and syntax. In a diagram, a uniqueness constraint for a class is specified by using the suffix `uniqueness constraint N` (`ucN`), where "`N`" is a positive integer. This annotation appears in the `SuffixCommaList` of each property that is subject to the "Nth" uniqueness constraint declared for the class. This constraint is illustrated in Figure 71.[64]

In this example, the dependent classes and uniqueness constraints preserve the business rules that are stated using primary keys in a similar key-style model (see also 9.7 and 9.9). Some of these rules are:

a) The business intends each `hotel` to have a `hotelId` and does not want two hotels to have the same `hotelId`. This rule is specified by the `( uc1 )` suffix on `hotelId` in `hotel`.
b) A `room` means a room-numbered room in a specific `hotel`. This excludes hallway linen closets or a bedroom within a suite. Each `hotel` assigns its own room numbers. No two rooms in a given hotel may have the same number. This is specified by the `( uc1 )` suffix on `hotel` and `roomNumber` in `room`.
c) What is relevant to the business is the fact that television sets are owned by a hotel, not the physical television sets *per se*. If one hotel sells off a television set and another hotel happens to buy it, no one cares. Furthermore, the second hotel will assign its own `tvNumber`. This is specified by the `( uc1 )` suffix on `hotel` and `tvNumber` in `tv`.
d) A TV may be used in many rooms of the hotel (over time), and a room may use many TVs. A record of the usage hours for each TV is to be kept by room.

[64]See Footnote 63.

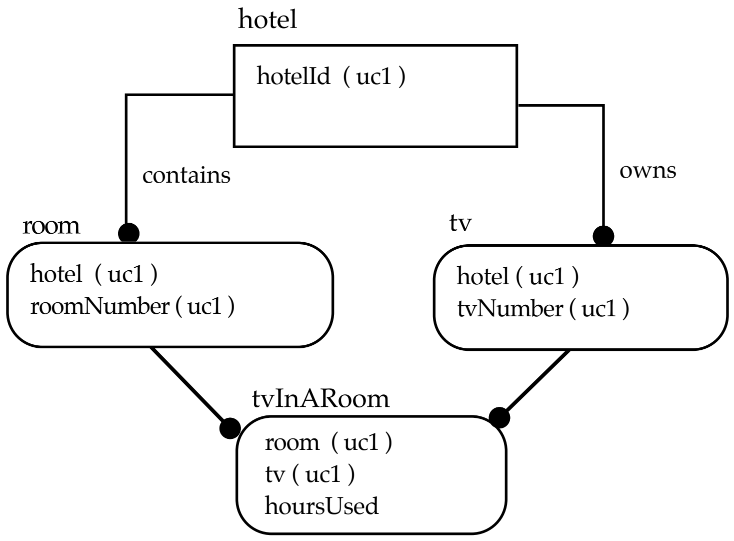


Figure 71—Uniqueness constraints

A uniqueness constraint may include multiple properties in its declaration, and these properties may include a mixture of property types, i.e., attributes and participant properties. For example, the constraint expressed in `room` includes an attribute (`roomNumber`) and a participant property (`hotel`). The constraint expressed in `tvInARoom` is formed entirely over participant properties.

6.7.1.5 Value class uniqueness constraint

Uniqueness constraints are used with value classes to specify an instance (see Figure 72). For example, the specification language statement

```
T is temperature with fahrenheit: 32
```

says that `T` is the instance of `temperature` that has a `fahrenheit` value of `32`. Likewise, the specification language statement

```
V is vector with ( x: 1, y: 2, z: 3 )
```

says that `V` is the instance of `vector` that has the specified coordinate values. The properties used in an associative literal can be the properties of any uniqueness constraint on the value class, regardless of representation. The representation of the specified instance is established according to the specification language for the realization for the uniqueness constraint.

For a value class, each uniqueness constraint must have a realization specified, where the name of the property is, for example, `uc1`, `uc2`, etc. Arguments are positional, as in any realization. All uniqueness constraint realizations for value classes are private. Thus, the position of the arguments in the signature may be changed without disturbing message senders outside the class. (See also Clause 7.)

6.7.1.6 Instance value constraint

A value class can have instance value constraints. In Figure 72, `isRectilinear` is an instance value constraint on `rectilinearVector`. An instance value constraint is true if the instance adheres to some constraint on the instance. In this case, the constraint is that the vector be rectilinear. Instance value constraint

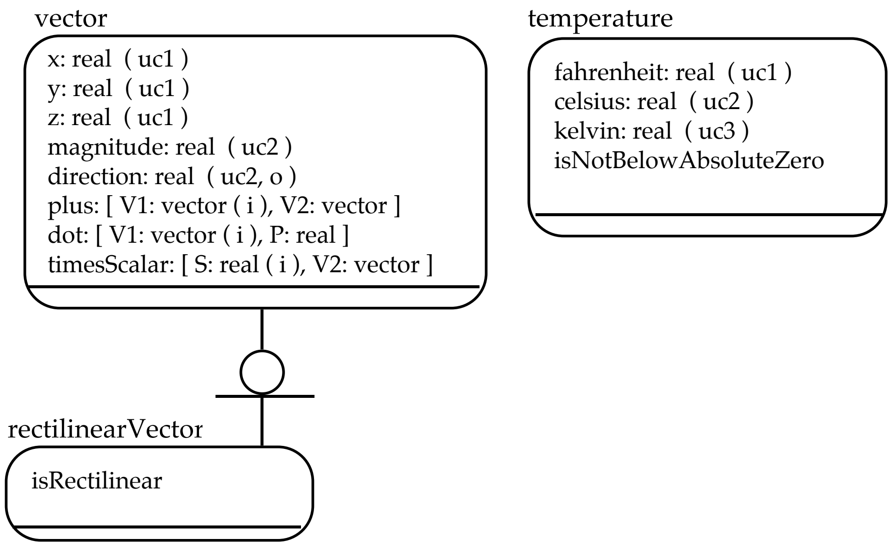


Figure 72—Value Class uniqueness constraints

checking is part of the semantics of specification of a value class instance by a uniqueness constraint. While an instance of a class is being established, the instance value constraints for the class are checked. If any are not true, the attempt to establish the instance fails. This applies recursively up the generalization hierarchy.

6.7.2 Constraint syntax

6.7.2.1 Signature

a) The *signature* of a constraint shall consist of the constraint name.

b) In a diagram, the constraint signature may be shown as an *annotated constraint signature* (the signature with additional keyword annotations), inside the class rectangle.

c) The general form of an annotated constraint signature shall be

```
Visibility ( PrefixCommaList ) ConstraintName ( SuffixCommaList )
```

d) With the exception of the `ConstraintName`, each of the elements of the annotated constraint signature shall be optional.

e) Each of the keywords in the constraint signature may be abbreviated as explained in 6.3.2.9, so long as no ambiguity results. Specifically,

```
( class, constraint )
```

may be abbreviated as

```
( cl, co ).
```

6.7.2.2 Visibility annotation

a) The visibility of a constraint may be restricted as protected or private using this standard visibility annotation at the beginning of the constraint signature (see 6.3.2.2).

6.7.2.3 PrefixCommaList clause

a) A class-level constraint shall be marked by the keyword `class` in parentheses preceding the constraint name as part of the `PrefixCommaList`, as shown in Figure 73.

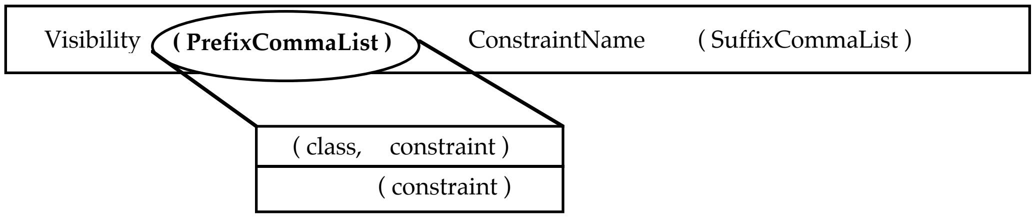


Figure 73—Constraint `PrefixCommaList`

b) If “`class`” is not specified, the constraint shall be an instance-level constraint.
c) A constraint shall be marked by the keyword `constraint` in parentheses preceding the constraint name as part of the `PrefixCommaList` (Figure 73). This is illustrated in Figures 69 and 70.
d) Multiple `PrefixCommaList`s shall be equivalent to a single `PrefixCommaList` with the keywords separated with commas. For example, `(class)  (constraint)` is equivalent to `(class, constraint).`

6.7.2.4 SuffixCommaList clause

a) The following shall be the only keyword option that is valid in the `SuffixCommaList` of a constraint:
   `subclass responsibility`
b) By definition, a constraint shall always be `total`, `constant`, `read-only`, and `single-valued`.

6.7.3 Constraint rules

6.7.3.1 Responsible class

a) A named constraint shall be specified as a responsibility of one of the classes that is referred to in its description text. (That class is considered responsible for knowing if the constraint is satisfied.)

6.7.3.2 Naming/signature

a) A constraint of a given signature may appear in more than one class in a view.
b) No two constraints with the same signature may appear in the same class.
c) A constraint and a property with the same signature may not appear in the same class.

A more complete explanation of the signature uniqueness requirements is given in 7.5.3.

6.7.4 Constraint requests

6.7.4.1 Constraint checking

a) A request may be made for the truth of a constraint.[65] A request is issued by one instance to another instance and shall be the only way to test a constraint.
b) The form of a request for a constraint shall be that for a request with no arguments (see 6.2.2.6).

[65]For example, a constraint check might be made in a post-condition. See also the discussion of constraint checking in 7.10.

For example, (referring to Figure 69) if `CA` is an instance of `checkingAccount`, then the common owner constraint for `CA` may be checked by the request

```
CA has commonOwner.
```

c) The request shall be true if the `commonOwner` constraint is satisfied for `CA`; the request shall be false if it is not satisfied.

d) A constraint is a predicate—i.e., something that may be true or false—with the requirement that the predicate be true for every instance of the class. For example, the truth of the `commonOwner` constraint may be checked for every instance of `checkingAccount` by the sentence:

```
forall ( #checkingAccount has instance(s)..member: CA ):
    CA has commonOwner ).
```

This sentence can be read "for every `checkingAccount` instance, `CA`, the `commonOwner` constraint is true for `CA`" or, in a more natural fashion, "every checking account satisfies the common owner constraint." The sentence is false if for any instance `CA` the `commonOwner` constraint is not true.

6.7.5 Constraint realization

6.7.5.1 Named constraint

a) The realization of a named constraint shall be stated using the specification language (see Clause 7) in the form given for property realization with no arguments (see 6.3.4).

For example, the `balanceUnderLimit` constraint in Figure 69 has the realization shown in Figure 74.

```
creditCardAccount: TheCreditCardAccount has balanceUnderLimit ifdef
        TheCreditCardAccount..balance < TheCreditCardAccount..limit.
```

Figure 74—Realization of `balanceUnderLimit` constraint

The natural language reading of this constraint is: "The credit card account has a balance under its limit if the credit card account's balance is less than the credit card account's limit."

In this example, `TheCreditCardAccount` was used instead of `Self` to illustrate another style. Either is valid. If the `Self` form had been used, the natural language reading would change to read: "I have a balance under my limit if my balance is less than my limit."

The `commonOwner` constraint in Figure 69 has the realization shown in Figure 75.

```
checkingAccount: TheCheckingAccount has commonOwner ifdef
        (if TheCreditCardAccount has protector: TheCheckingAccount
        then
           TheCheckingAccount..owner == TheCreditCardAccount..owner
        endif).
```

Figure 75—Realization of `commonOwner` constraint

The natural language reading of this constraint is: "A checking account has a common owner if a credit card account is the checking account's protector and the checking account's owner is the credit card account's owner."

The common ancestor constraint shown in Figure 70 is stated in Figure 76.

```
tvInARoom: Self has hotelOwnsTv ifdef
        Self..room..hotel == Self..tv..hotel.
```

Figure 76—Realization of `hotelOwnsTv` constraint

The natural language reading of this constraint is: “A tv in a room has valid hotel ownership if the hotel that contains the room of this tv and the hotel that owns the tv are precisely the same hotel.”

6.7.5.2 State class uniqueness constraint

a) A uniqueness constraint may have no user-defined realization for a state class—i.e., the realization of a state class uniqueness constraint is built-in to the language.

6.7.5.3 Value class uniqueness constraint

a) A realization shall be written for a value class uniqueness constraint.
b) Value class uniqueness constraints shall be realized using the specification language. Figure 77 shows the constraint realizations for the uniqueness constraints in the `temperature` value class that is shown in Figure 72. The pattern is the same in each case:
   1) Map the input argument to a kelvin value (the chosen representation), and
   2) Say that the representation `kelvin` value agrees.

```
temperature: Self has uc1: [ F ] ifdef
         K is 273.16 + ( F - 32 ) * 5/9,
         Self has kelvin: K.
temperature: Self has uc2: [ C ] ifdef
         K is 273.16 + C,
         Self has kelvin: K.
temperature: Self has uc3: [ K ] ifdef
         Self has kelvin: K.
```

Figure 77—Realization of `temperature` value class uniqueness constraints

6.7.5.4 Instance value constraint

a) Uniqueness constraints are used to establish an instance of a value class.

   Instance value constraints ensure that an instance is valid. In the `temperature` class, it is required that the `kelvin` value be nonnegative. The realization of this instance value constraint is:

```
temperature: Self has isNotBelowAbsoluteZero ifdef
    Self..kelvin >= 0.
```

   It is part of the semantics of instance specification to check the instance rules, so the specification language statement

```
    T is temperature with celsius: -300
```

   is false and `T` has no value.
b) Instance value constraints shall be realized using the specification language.

Figure 78 shows the constraint realizations for the `vector` value class that is shown in Figure 72.

```
vector: Self has uc1: [ X, Y, Z ] if_def
         Self has x: X,
         Self has y: Y,
         Self has z: Z.
vector: Self has uc2: [ M, D ] if_def
         D..cosineVector has timeScalar: [ M, V ],
         X is V..x,
         Y is V..y,
         Z is V..z,
         Self has x: X,
         Self has y: Y,
         Self has z: Z.
```

Figure 78—Realization of `vector` value class constraints

6.8 Note

A *note* is a body of free text that describes some general comment or specific constraint. A note may

— Be used in an early, high-level view prior to capturing constraints in the specification language;
— Further clarify a rule by providing explanations and examples;
— Be used for “general interest” comments not involving rules.

These notes may accompany the view graphics.

6.8.1 Note semantics

Notes can be used in a variety of ways, for example,

a) To make a general statement about something during the early stages of analysis that would become more formalized as a constraint in the specification language. For example, a common ancestor constraint that could be stated using RCL might initially be stated informally in a note. Similarly, an “exclusive OR” constraint might state for an instance of a given parent class, that, if an instance of one child class exists, then an instance of a second child class will not exist.
b) To record a preliminary understanding of some constraint that may be refined using the graphical syntax, e.g., annotating a generalized “many” cardinality constraint.
c) To describe circumstances in which an attribute with a value assertion specified in RCL can have no value.

A note is associated with the impacted view component, i.e., class, responsibility, relationship, or view. It may apply to a single component or to several.

6.8.2 Note syntax

6.8.2.1 Note body

a) A note shall contain a note body that consists of a block of free text.
b) When a note body is presented on a diagram or other display medium, *whitespace* (spaces, tabs, etc.) shall be used to separate the note text from the note identifier.

6.8.2.2 Note Identifier

a) A note shall contain a note identifier that is a nonzero, unsigned integer.
b) A note identifier shall be presented enclosed within parentheses.
c) When a note is attached to a relationship cardinality dot, the note identifier shall be placed either
   1) Following the cardinality annotation symbol (the `P`, `Z`, etc.), if there is one, or
   2) Directly following the dot, if there is no cardinality annotation symbol.
d) When multiple notes apply to the same component, one of two display forms shall be used:
   1) Each note identifier shall be enclosed within parentheses, or
   2) All note identifiers, separated by commas, shall be enclosed within a single set of parentheses.
e) When attaching a note to a relationship cardinality annotation (the dot or the symbol placed with the dot), the note identifier shall be placed either
   1) Following the cardinality annotation symbol, if there is one, or
   2) In place of the symbol, if there is none.
f) A note identifier that applies to one of the elements of a relationship label (verb phrase or role name) shall follow the element to which it applies.[66]

   Figure 79 shows a two-direction relationship label with participant role names, with notes for each of the elements.

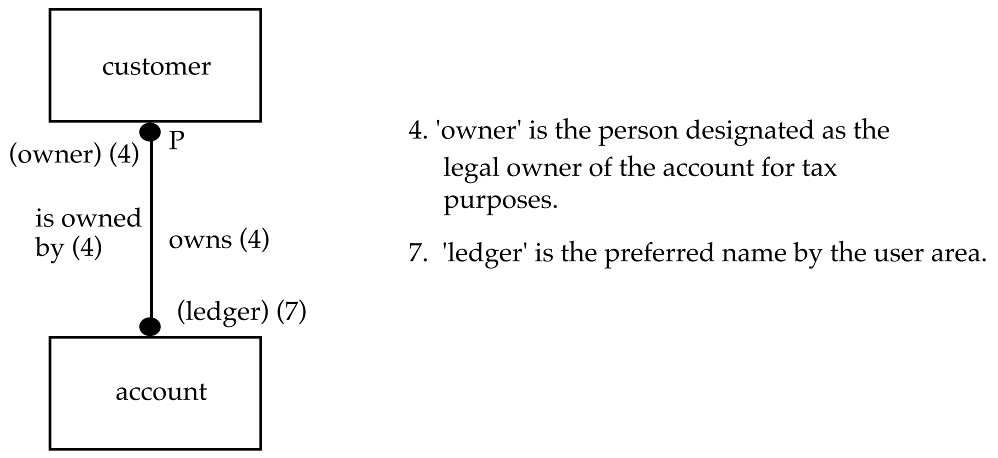

customer
P
(owner) (4)
is owned by (4)
owns (4)
(ledger) (7)
account
4. 'owner' is the person designated as the legal owner of the account for tax purposes.
7. 'ledger' is the preferred name by the user area.

Figure 79—Relationship label with notes

6.8.3 Note rules

6.8.3.1 Note body

a) A note shall be either
   1) General in nature, or
   2) Documenting a specific constraint.
b) The note body may be displayable on a view diagram.
c) Note text may include any character symbol.
d) All note body characters, including text spacing and formatting, shall be significant and shall be preserved.
e) The same text shall apply to the same note number when that note number is used multiple times in a view.

[66]There is no confusion between the note annotations and the role names since role names may not begin with an integer.

6.8.3.2 Note identifier

a) A note identifier shall be unique within a view.
b) A note may be attached to the following:
    1) The label of any model component—e.g., class label, view label, responsibility label, relationship verb phrase
    2) The cardinality annotation symbol
    3) The cardinality annotation
c) A note may not be attached to a note identifier or to a note body.

7. Rule and constraint language

7.1 Introduction

The Rule and Constraint Language (RCL) complements the graphic constructs of IDEF1X$_{97}$. RCL is used to specify the realizations of responsibilities and to express queries and updates against models. The combination of graphics and RCL allows the modeler to represent and reason about the subject under study to whatever level of specificity the modeler decides is appropriate.

The overall goal of RCL is to combine the clarity of logical specifications—that is, specifications based on logic—with the abstractions of the object model and to do so in a way that is tightly integrated with the graphic constructs and directly executable. Logic and objects are combined by treating an object message as a logical proposition. Object interfaces are stated using the graphic constructs. The responsibility realizations are stated using RCL. The names used in the RCL are the names appearing in the graphics.

Realizations are logical sentences formed by connecting message propositions with logical connectives such as *and*, *or*, *not*, and *if then*. Read declaratively, a sentence states what must be true about a solution, without regard to how the solution is found. Read procedurally, a sentence states what must be done to obtain a solution.

RCL includes

a) Direct support for the modeling constructs of IDEF1X$_{97}$
b) A single language for pre-conditions, post-conditions, assertions, queries, updates, and realizations for attributes, participants, operations, and constraints
c) A logical, declarative reading as well as a procedural one
d) Two-valued logic—no nulls
e) A distinction between mutable and immutable objects
f) Flexible typing ranging from untyped to statically typed
g) Property overriding for substitution or specialization
h) Dynamic binding based on argument dynamic types
i) Direct execution

7.1.1 Objects and classes

An object is a discrete thing, distinct from all other objects. Each object has an intrinsic, immutable *identity*, independent of its property values and classification.

Every object is classified into one or more classes. An object is an *instance* of each class into which it is classified. The set of objects classified into a class is the *extent* of the class.

Classes are defined within views.

There are two kinds of classes: state classes and value classes. A *state class* has a time-varying extent and the objects in the extent have time-varying property values. A *value class* has a fixed extent, and the objects in the extent have fixed property values.

7.1.2 Generalization

Generalization is concerned with the definition of objects.

There is a single top class, called `object`. Every other class has one or more direct superclasses. The meaning is that an object that is an instance of a class is also an instance of each superclass. A superclass of a class is a direct superclass or a superclass of a direct superclass.

A subclass is said to be lower that its superclass. If an object is an instance of a class `C` and not an instance of any subclass of `C`, then `C` is a *lowclass* of the object.

A subclass *inherits* the responsibilities of its superclasses. A subclass may have additional responsibilities beyond those of its superclasses or may *override* one or more of the responsibilities of the superclasses.

A property `P′` of a class `C′` that overrides a property `P` of a superclass `C` may do so in one of two ways: as a substitution for `P` or as a specialization of `P`. If `P′` *substitutes* for `P`, then `P′` is used for every message to instances of `C′` that would use `P` if the message were to an instance of `C`. If `P′` *specializes* `P`, then `P′` is used for some messages to instances of `C′` and `P` is used for other messages to instance of `C′`, depending on the (dynamic types of the) argument values in the message.

Whether `P′` is a substitute or specialization is a matter of intent. It is up to the modeler to choose whichever best models the “real world” under study. Once the choice is made, the rules regarding the typing of arguments are used to carry out that intent.

7.1.2.1 Multiple clusters

State and value classes may have multiple clusters of subclasses. The classes within a cluster are pairwise mutually exclusive, meaning that no object is an instance of two classes in the cluster. Two classes in different clusters are not necessarily mutually exclusive. Two classes are *mutually exclusive* if they are in the same cluster or a superclass of either is mutually exclusive with a superclass of the other. No class may have two superclasses that are mutually exclusive with one another.

Two classes are *parallel* if neither is a superclass of the other and they are not mutually exclusive. Parallel classes may occur only with multiple clusters.

A cluster is a *total cluster* if every instance of the superclass is an instance of one of the subclasses in the cluster, otherwise it is a *partial cluster.*

A class is *abstract with respect to a cluster* if the cluster is total. A class is *abstract* if it is abstract with respect to at least one cluster.

Parallel value classes must be abstract and every pair of parallel value classes must have a common subclass.

The result of these rules is that a value class instance always has exactly one lowclass, but a state class instance may have multiple lowclasses.

7.1.3 Collection and pair classes

Where `T` is a type, the value classes `set(T)`, `list(T)`, `bag(T)`, and `pair(T`$_1$`,T`$_2$`)` provide *parametric polymorphism*. They are polymorphic on the type of the elements, `T`, but `T` is used only to specify the types of property arguments. The computations within the realizations do not depend on the specific value of `T`.

7.1.3.1 Collection classes

Where `T` is a class, a `set(T)`, `list(T)`, or `bag(T)` is a *collection class*. Examples include the classes `set(integer)` and `list(bag(real))`. Each instance of a collection class is a collection. For example, the set with elements `1, 2, 3` is a valid instance of `set(integer)`, but the set with elements `4, "abc", 1.7` is not. Both are valid instances of the collection class `set(object)`.

7.1.3.2 Pair class

Where `T`$_1$ and `T`$_2$ are classes, a `pair( T`$_1$`, T`$_2$` )` is a *pair class*. Examples include the classes `pair(integer, integer)` and `pair( list(bag(real)), integer )`. Each instance of a pair class is a pair. For example, the pair with left side `'big'` and right side `8` is a valid instance of `pair(identifier, integer)`. It is also a valid instance of the class `pair(object, object)`.

7.1.4 Responsibility

An object has a set of responsibilities, and all objects in a class have the same kind of responsibilities. A *responsibility* is a property or a constraint. A *property* is an attribute, a participant property arising from a relationship, or an operation. An *attribute* is a mapping from a class to a value class. A *participant property* is a mapping from a state class to a state class. Both state classes and value classes may have operations and constraints. A responsibility has a name and zero or more arguments. An attribute has one argument—a value of the attribute. A participant property has one argument—the identity of a related object. An operation has zero or more arguments. A constraint has no arguments.

A responsibility is a *relational mapping* from the cross product of the extents of the classes of the receiver and the input arguments to the cross product of the extents of the classes of the output arguments. The mapping may be total or partial, single-valued or multi-valued, and with or without side effects. Alternatively, a responsibility is a *relation*—a subset of the cross product of the receiver's extent and the argument's extents. The responsibility maps a particular receiver object and input argument values to particular output argument values if those particular values are a member (tuple) of the relation. For a value class, the relation is fixed. For a state class, the relation is time varying.

7.2 Realization

a) Each responsibility is *realized* by
   1) Stored values for the tuple, or
   2) A computation. The computation states the necessary and sufficient conditions that a tuple be in the relation.
b) The RCL relevant to realizations is reproduced below. See 7.15 for the complete RCL syntax and an explanation of the notation used to present the syntax.
   1) `RealizationRCL` → `Head` **`if`**$_{\mathbf{def}}$ `Body.`
   2) `Head` → `class_qname : Variable` **`has`** `ResponsibilityName` *{* `PropertyOperator Arguments` *}*
   3) `Body` → *{* **`pre`** `Sentence,` *}* `*` `Sentence` *{* `,` **`post`** `Sentence` *}**
   4) `ResponsibilityName` → `Identifier` or `Identifier(` **`s`** `)`
   5) `PropertyOperator` → `:` *or* `:=` *or* `:!=` *or* `:+=` *or* `:-=`
   6) `Arguments` → `Argument` *or* `[`**`Argument`** *{* `, Argument` *}** `]`

7) `Argument → Variable { : TypeLiteral }`

8) `TypeLiteral →`
```
any
or bot
or Variable
or class_qname
or parametricVClass_qname(TypeLiteral {, TypeLiteral }* )
```

c) The built-in parametric value classes are `set(T)`, `list(T)`, `bag(T)`, `pair(T1,T2)`, and `accumulator(T)`.

d) The `Variable`s in a multiargument property are aligned by position with the interface specification for the property.

e) If the interface specifies a `ValueName` for an argument, then the `Variable` shall be the same as the `ValueName`. For example, from Figure 44, the `vector` class operation

```
(operation) plus: [ V1: vector (input), V2: vector ]
```
would have a realization
```
vector: Self has plus: [ V1, V2 ] if_def …
```

f) The `pre` and `post` clauses are explained in 7.10.2.

g) The `Sentence` is the computation that derives the output variable values from the identity of the receiver and the input values.
   1) The `Sentence` is a logical sentence. It evaluates to true or false.
   2) The `Sentence` specifies how the receiver and inputs are related to the outputs.
   3) If what the sentence says is true for some particular receiver and inputs, then they do map to some particular outputs. If it is false, they do not.

h) If a `Sentence` evaluates to true, then it shall determine a value for each output variable. If it does not, an exception shall be raised.

i) All variables are local to a realization (a query); there are no global variables.

j) Attributes and participant properties have default realizations for

1)	`propertyName : Variable`	
2)	`propertyName := Variable`	
3)	`propertyName :!= Variable`	if optional
4)	`propertyName :-= Variable`	if multi-valued or collection-valued
5)	`propertyName :+= Variable`	if multi-valued or collection-valued

k) Attributes and participant properties are not instance variables; they are methods that operate on completely hidden instance variables. The defaults may be overridden by supplying a realization in RCL.

l) A property name may be an operator such as "+," which enables
```
X is "let's" + "do" + "it"
```
to be written.

7.2.1 Value class uniqueness constraints

a) For each uniqueness constraint, a value class has a responsibility named `uc1`, `uc2`, or (in general) `ucN`, where `N` is the uniqueness constraint number.

b) The arguments are positional, in the order the uniqueness properties are specified for the class.

c) In the realization, the `Sentence` shall determine the value `V` for each representation property `P` and send the message
```
Self has P: V.
```

d) If the uniqueness properties are multi-valued, one of the equivalent values shall be consistently used as the representation property value in order for equality to function properly. For example, a rational number value class may use `numerator` and `denominator` as the representation properties and the uniqueness constraint properties. The rational number `1/2` is the same as `2/4` or `3/6` or `4/8`, etc. The obvious value to use as representation is the reduced fraction.

7.2.2 Overriding built-ins

a) Any of the built-in class responsibilities may be overridden, including `new`, `init`, `create`, `delete`, etc.

b) Any of the default property operators may be overridden.
   1) Typically, a message to invoke the default (a message to `super`) is done in the overriding method.
   2) The message for the default for each property operator is given in Table 11.

Table 10—Messages for default properties

PropertyHead	Message for default property
`Cn: Self has P : V`	`Self super has P : V`
`Cn: Self has P := V`	`Self super has P := V`
`Cn: Self has P :!= V`	`Self super has P :!= V`
`Cn: Self has P :+= V`	`Self super has P :+= V`
`Cn: Self has P :-= V`	`Self super has P :-= V`

7.3 Message

a) A request to an object for one of its responsibilities is called a *message*. A message consists of
   1) The identity of the receiver,
   2) The name of a responsibility, and
   3) Optionally
      i) A property operator,
      ii) The values of the input arguments, and
      iii) the (typically unknown) values (typically as variables) of the output arguments for the responsibility.

   If a responsibility is viewed as a mapping, a message applies the mapping to the receiver and input values to yield the output values. Viewing a responsibility as a relation, a message is a proposition that is true if the tuple consisting of the receiver and argument values occur as a row in the relation, and false otherwise.

b) The syntax of a message is

   `Object` *{* **`super`** *}* `Having` *{* `PathExpr` *}* `ResponsibilityValue`

   where

   1) `PathExpr` → *{* `PropertyExpr` *{* `: SimpleObject` *}* `..` *}+*
   2) `PropertyExpr` →
      `ResponsibilityName_var`
      or `PropertyNameSingular(Objects)`
   3) `Objects` → `Object` *{* `, Object` *}**
   4) `ResponsibilityValue` →
      `ResponsibilityName_var` *{* `PropertyOperator SimpleObject` *}*
      or `ResponsibilityOid` *{* `: SimpleObject` *}*
      or `PropertyNameSingular ( Objects` *)*
   5) `ResponsibilityName` → `Identifier` *or* `Identifier(` **`s`** `)`
   6) `PropertyNameSingular` → `Identifier`

```
7)  ResponsibilityOid → Variable
8)  Object →
    SimpleObject
        or Literal
        or Object.. { PathExpr } PropertyExpr
        or UnaryOp Object
        or Object BinaryOp Object
        or Object where Sentence
9)  SimpleObject →
    Variable
        or String
        or Identifier
        or Number
        or #Constant
        or true
        or false
        or SimpleObject : SimpleObject
        or SimpleObjectList
        or {}
10) SimpleObjectList →
    [{SimpleObject { , SimpleObject }* } ]
        or [ SimpleObject | SimpleObjectList_var ]
```

c) A *path expression* is a series of properties, `Property`$_1$.. `Property`$_2$`..`, …, `..` `Property`$_n$ for which `n >= 0`.

The message

`Object has Property`$_1$`.. Property`$_2$`.., …, .. Property`$_n$`..Responsibility`

is defined to be equivalent to the conjunction

```
Object has Property₁: V₁,
V₁ has Property₂: V₂,
…
Vₙ₋₁ has Propertyₙ:,
Vₙ has Responsibility
```

where the variables `V`$_1$, `V`$_2$, … `V`$_n$ do not otherwise occur within the query or realization.

1) If any property has input arguments, they may be specified by `Property`$_i$`(Args)`.
2) If there are `N` arguments, the first `N-1` should be specified; the last is assumed to be the output, `V`$_i$.

d) An `Object` is a state class instance or value class instance. Syntactically, an `Object` may be a variable. If the receiver or an input argument is a variable, it shall have a value when the message is sent.

e) The `ResponsibilityOid` shall be a responsibility instance. This form directly invokes the realization for the responsibility. Inheritance is bypassed. The pre-condition, post-condition, total, function, and read-only constraints are checked.

7.3.1 Message to a class

a) A message may be sent to a class or to an instance of a class. Syntactically, a message is to a class when the receiver is of the form `#Cn` where `Cn` is the name of a class.

b) In a class-level realization, `Self` is bound to the receiver class, so
1) A message to `Self` is a message to the receiver class, and
2) A message to `Self super` is a message to a superclass of the class in which the realization is defined.

7.3.2 Creating a new state class instance

a) A new state class instance is created by the message

```
#Cn has new: I
```

where `Cn` is a state class name and `I` is the identity of the new instance.

b) An instance may be initialized by

```
I has init: [ P1: V1, P2: V2, … , Pn: Vn ]
```

where each `Pi` is a direct or inherited property of `I`. The result is that a

```
I has Pi := Vi
```

is done for `i = 1 to n`.

c) The functions of `new` and `init` are combined in the `create` method:

```
#Cn has create: [ P1: V1, P2: V2, … , Pn: Vn ]
```

This message is equivalent to

```
#Cn has new: I,
I has init: [ P1: V1, P2: V2, … , Pn: Vn ].
```

d) The identity of the created instance may be specified by a last argument,

```
#Cn has create: [ P1: V1, P2: V2, … , Pn: Vn, I ]
```

which equates the oid of the new instance to `I`.

e) If `I = #Constant` when the `create` is issued, then `I` shall be the oid of the created instance so long as it does not duplicate any prior oid.

7.3.3 Deleting a state class instance

a) A state class instance is deleted by the message

```
I has delete
```

where `I` is the identity of the instance to be deleted.

7.3.4 Displaying an instance

a) An instance is displayed by the message

```
I has display
```

where `I` is the identity of the instance to be displayed.

b) For a state class instance, the display includes
    1) An external identity (`#Constant`),
    2) The name of the lowclass(es), and
    3) Property name value pairs for all nonderived attributes and participant properties, including inherited properties.

c) For a value class instance, the display includes
    1) The name of the lowclass, and
    2) Property name value pairs for all the nonderived attributes.

d) If the display is graphical, the format shall be that of the instance diagrams or instance tables.

e) The format of nongraphical displays is implementation-dependent.

7.3.5 Boolean attribute

a) A boolean attribute `p` of an `Object` is set true by a message

```
Object has p:= true.
```

b) A boolean attribute `p` of an `Object` is set false by a message

```
Object has p:= false.
```

7.3.6 Changing the class of a state class instance

a) An instance of a state class may be removed from the extent of a class or added to the extent of a class, so long as doing so is consistent with the generalization hierarchy (see Figure 18).

b) When an instance is removed from the extent of a class, any relationships in which the instance participants are updated as though the instance had been deleted—but only for that class.

7.3.6.1 Specialize

a) An existing instance `I` may be added to a subclass `Cn` by the `add` method:
```
        #Cn has add: [ P1: V1, P2: V2, … , Pn: Vn, I ]
```
where the `Pi: Vi` are used to initialize the `Cn` properties.

7.3.6.2 Remove

a) An existing instance `I` may be removed from the extent of a class `Cn` by the unspecialize method `remove:`
```
        #Cn has remove: I
```

7.3.7 Specifying an existing instance

7.3.7.1 State class

a) The equivalent propositions
```
        I is Class( PropertyValue { , PropertyValue }* )
        I is Class with ( PropertyValue { , PropertyValue }* )
```
set `I` to the identity of an instance of the state class `Class` that has the specified property values.

b) The proposition is equivalent to the conjunction
```
        Class has instance: I,
        I has PropertyValue
        { , I has PropertyValue }*
```

7.3.7.2 Value class

a) The equivalent propositions
```
        I is Class( PropertyValue { , PropertyValue }* )
        I is Class with ( PropertyValue { , PropertyValue }* )
```
set `I` to the identity of the instance of the value class Class that has the specified property values.

b) The properties named shall constitute a uniqueness constraint for the value class.

7.4 Typing

Typing is concerned with the use of objects—not their definition. More specifically, typing is concerned with when it is safe to send a message to an object or to pass an object as an input argument in a message, where safe means without chance of a run-time error such as “property not found.” More specifically yet, the key question is when is it safe to use an object of one class when an object of another class is expected. The notions of type and subtype are used to answer that question.

Objects have classes and variables have types. Every object has a lowclass. A variable may have a type declared, which is the name of a class or any. The type of a variable limits the objects that may be assigned to the variable to just those objects that are instances of the class specified as the type. If the type is any, then any object may be assigned. When a variable is bound to an object, a lowclass of the object is sometimes called a dynamic type of the variable. The declared type of the variable is called the static type.

Type checking means checking that variables are assigned only to objects that conform to the type declarations. Type checking may be static (done on the source text of the RCL using the static types) or dynamic (done at run time using the lowclass of the objects as well as the static types). Either or both may be done with RCL. Untyped, partially typed, or fully typed models may be executed.

7.4.1 Type and subtype

A class implements a type if it has all the responsibilities of the type. An object has type `T` if the object is an instance of a class that implements type `T`. Every class implements a type of the same name. Class `#Cn` implements type `Cn`. A type `T` is a subtype of type `T′` if `T` includes all the responsibilities of `T′`. Unlike a class, a type does not have instances. Subtype is not the same as subclass. Subclass implies subtype, but not the other way round.

a) Every object has type `any`, the *universal type*. The universal type is used as an escape from type checking for a fully typed model.
b) The key idea of subtyping is that of *subsumption*: if `X` has type `T` and `T` is a subtype of `T′`, then `X` has type `T′`.
c) The notation
```
T <: T′
```
means that `T` is a subtype of `T′`. Subtype is reflexive and transitive.
d) For all types `T, T′`
```
1) bot <: T
2) T <: any
```
3) `T <: T′` ← `#T` is a subclass of `#T′`
```
4) Cn(T1,T2,…Tn) <: Cn(T1, T2′,…,Tn′) ← T1 <: T1′ ∧ T2 <: T2′…∧ Tn
   <: Tn′
```
where `Cn` is a parametric value class, such as `set`.
e) With the definitions given, subsumption holds:
f) `X` has type `T′` if `X` has type `T` and `T <: T′`.
g) Because <: is reflexive, every class is both a subtype and supertype of itself.
h) Because <: is transitive, every direct and indirect subclass of a class is a subtype of the class, and every direct and indirect superclass of a class is a supertype of the class.
i) Class C_1 may be used when class C_2 is expected if C_1 <: C_2.
j) The type `bot` is implemented by no class, so no instance ever has type `bot`. The lowclass of an empty list, set, or bag is `list(bot)`, `set(bot)`, or `bag(bot)`, respectively. The result is that, for example, an empty set is type-acceptable to `set(T)` for any type `T`.

7.4.2 Dynamic type

a) Every object has a lowclass.
b) A state class instance may have multiple lowclasses.
c) A value class instance always has exactly one lowclass.
d) A lowclass of an object is also called a *dynamic type* of a variable bound to the object or an expression that evaluates to the object.
e) The lowclass of a collection class is `Cn(T)` where `T` is the least upper bound class of the types of the members of the collection. For an empty collection, `T = bot`.

7.4.3 Static type

7.4.3.1 Variables

a) Variables (e.g., local variables and arguments) may optionally have a *declared type*, also called a *static type*.
b) If no type is declared, the variable is *untyped*, meaning it has no static type.
c) For an argument, the static type `T`, called the *argument type*, means that only values that have type `T` are acceptable as values for that argument.

d) The type of an argument is declared as part of the property signature.
e) For a local variable, the static type `T`, called the *variable type*, means that only values that have type `T` are acceptable as values for that variable.
f) The type of a variable is declared by

```
        Variable { : TypeLiteral } is Object
        or variable Variable { : TypeLiteral } { is Object }
```

g) The static type of `Self` is the class for which the realization is defined.

7.4.3.2 Literals

a) The static type for an integer literal, such as `7`, is `integer`.
b) The static type for a real literal, such as `3.14`, is `real`.
c) The static type for a string literal, such as `"hi"`, is `string`.
d) The static type for an identifier literal, such as `ho`, is `identifier`.
e) The static type of a pair literal `X: Y` is `pair(Tx, Ty)` if `Tx` is the static type of `X` and `Ty` is the static type of `Y`; otherwise, `pair(any, any)`.
f) The static type of a list literal `[ X` where `Proposition ]` is `list(T)` where `T` is the type of `X`. The static type of the list literal `[ ]` is `list(bot)`. The static type of the list literal `[ X1, X2, …, Xn ]` is `list(T)` where `T` is the least common supertype of the static types of `X1, X2, …, Xn` if they all have a static type; otherwise, `list(any)`.
g) The static types of the `set` and `bag` literals are exactly analogous to those for the `list` literals.

7.4.3.3 With

a) The static type of

```
        class_name with ( PropertyValue { , PropertyValue }* )
```

is class_name.

7.4.3.4 Casting

a) For any `Object`, the cast

```
        Object as T
```

has static type
   1) `T` if `Object` has no static type
   2) `T` if `Object` has static type T_s and T_s `<: T` or `T <:` T_s
   3) None, and the static type check fails otherwise.
b) Casts affect only the static type, not the dynamic type.

7.4.4 Typing rules for overrides

a) The typing rules adopted for overrides have these objectives:
   1) Allow type checking to avoid run-time errors
   2) Allow overriding for substitution if substitution is the intent
   3) Allow overriding for specialization if specialization is the intent
   4) Ensure overriding adheres to the substitution principle

7.4.4.1 Overriding

a) A property `P′` of a class `C′` that overrides a property `P` of a superclass `C` shall meet certain conditions:
   1) `P′` shall have the same name as `P`.
   2) `P′` shall have the same number of arguments as `P`.
   3) `P′` shall be read-only if `P` is read-only.
   4) `P′` shall be constant if `P` is constant.

5) `P′` shall be at least as visible as `P`.
6) `P′` shall be single-valued if `P` is single-valued.
7) For every argument `A: T` in `P` and corresponding argument `A′: T′` in `P′`,
   `T′ <: T` or `T <: T′`, and
   i) If A is an output argument, then `T′` <: `T`.
   ii) If A is not updatable, then `A′` shall not be updatable.

7.4.4.2 Overriding for substitution

a) If `P′` *substitutes* for `P`, then `P′` is used for every message to an instance of `C′` that would use `P` if the same message were sent to an instance of `C`.
b) Two additional conditions shall be met:
   1) For every input argument `A: T` in `P` and corresponding argument `A′`: `T′` in `P′`,
      i) `A′` is an input argument, and
      ii) `T <: T′`.
   2) For every output argument `A` in `P` and corresponding argument `A′` in `P′`,
      i) `A′` is an output argument.

   The first condition is called the *contravariance* rule.

7.4.4.3 Overriding for specialization

If the rules for overriding are met, but not those for specialization, then the result is overriding for specialization.

a) For some input argument `A: T` in `P` and corresponding argument `A′: T′` in `P′`,
   1) `A′` is an input argument, and
   2) `T′ <: T`, and
   3) `T′ != T`.

   This condition is known as *covariant* specialization.

7.4.5 Determining the class of an object

a) For an object `X`,
   ```
   X has lowClass: LC
   ```
   is true for any lowclass `LC` of `X`.
b) For an object `X`,
   ```
   X has class: C
   ```
   is true for any class `C` of which `X` is an instance.

7.5 Dynamic binding

A message is dynamically bound to one class responsibility (i.e., one specific responsibility of one specific class), binding the argument values in the message to the responsibility's arguments and evaluating the body of the responsibility's realization.

A message consists of the identity of a receiver, a responsibility name, and, optionally, a property operator; values for the input arguments; and variables (typically, although values may be provided) for the output arguments. The static text of the message may in general determine the static type of the receiver and arguments; but, because of overriding and subtyping, all that can be known about their dynamic types is that the dynamic type is a subtype of the static type. As a result, the static text of the message in general determines only a set of possible responsibilities to be used to resolve the message. The specific responsibility within the set cannot be known until run-time, when the identity (and therefore dynamic type) of the receiver and input arguments are known.

Message resolution determines the one, specific realization to use. There are three aspects to message resolution:

a) *Signature matching*—determining whether or not a given responsibility matches the message,
b) *Search order*—the order in which responsibilities are considered for matching,
c) *Uniqueness*—rules on responsibility definitions to guarantee that every message can be resolved to at most one responsibility.

The signature matching choices revolve around the use of static or dynamic types for the message receiver and arguments. Smalltalk uses the dynamic type of the receiver and does not use the types of the arguments at all. C++ and Java use the dynamic type of the receiver and the static type of the arguments. CLOS uses the dynamic types of the receiver (treated as just another argument) and the arguments.

The search order choices are concerned with, first, whether the receiver plays a dominant or equal role relative to the arguments; second, the relative order of search of instance-level responsibilities versus class-level responsibilities versus metaclass responsibilities; and, third, the order in which multiple superclasses are searched.

The choices on matching and search order largely determine the uniqueness rules needed.

7.5.1 Signature matching

The signature of a responsibility consists of its name, property operator, and the number and type of its arguments.

a) Each argument is either an input argument or not.
    1) The arguments are in a fixed order, and the message argument values are assumed to be in the proper order.
    2) An input argument shall have a value in order to invoke the responsibility.
    3) An output argument need not have a value, but is permitted to have one.
b) A responsibility may be used for a message
    1) If the responsibility names are the same, and,
    2) If there are arguments,
        i) The property operators are the same, and
        ii) For each input argument in the signature, the corresponding argument in the message is a value (not an uninstantiated variable), and
        iii) The argument in the message has a type that is a subtype of the signature type.
c) The argument in the message has two types: static T_s and dynamic T_d. The argument in the signature has just a static type `T`. Message resolution for RCL uses the dynamic type of the input arguments, i.e., a match requires T_d `<: T`.

    This rule supports overriding for both substitution and specialization. The receiver still has the dominant role because it is the dynamic type of the receiver that determines which classes to search. It is only among the properties of a class that the dynamic type of the input arguments are used. This is called *encapsulated multi-methods*.
d) If more than one property's signature matches the message, the best match is used. If `P` and `P′` both match a message, then `P` is the best match if `P` is less than `P′` according to
    1) Explicit properties < implicit properties
    2) Instance-level < class-level
    3) `C` < `C′` if `C` is a distinct subclass of `C′`
    4) `T` < `T′` if `T <: T′` and `not(T = T′)`

7.5.2 Search order

Inheritance occurs from instance to instance, from class to instance, from class to class, and from metaclass instance to class.

a) This is done by alternately using the steps 1 and 2 below, where step 1 searches instance-level responsibilities, and step 2 searches class-level responsibilities.
   1) If the message is to an instance of a class, the search begins with the instance-level responsibilities of the lowclass(es) of the instance. If no match is found, the search for instance-level responsibilities proceeds up toward the root (`object`). Any subclasses that need to be searched are searched before the superclass. If no match is found after searching `object`, the message is delegated to the lowclass, which requires step 2.
   2) If the message is to a class, the search begins with the class-level responsibilities of the class. If no match is found, the search for class-level responsibilities proceeds up toward the root (`object`). Any subclasses that need to be searched are searched before the superclass. If no match is found after searching `object`, the message is delegated to the class as an instance. (It is an instance of a metaclass.) In this message, the receiver is an instance, which requires step 1.

b) The receiver of a message may be an instance of a class or a class itself. A message is sent to an instance for instance-level responsibilities and to a class for class-level responsibilities.

c) If the receiver is an instance, the search begins with step 1. If the receiver is a class, the search begins with step 2.

d) The search ends when a match is found or when the next class to search for instance responsibilities has already been searched, in which case a “responsibility not found” exception is raised.

7.5.3 Uniqueness

The uniqueness conditions guarantee that a message can be resolved to at most one class responsibility.

a) Two signatures that agree on all but possibly type *overlap* if it is possible for a message to match both. Signatures `P` and `P′` overlap if
   1) The names are the same, and
   2) The number of arguments are the same, and
   3) For every argument `A` with type `T` of `P` and corresponding argument `A′` with type `T′` of `P′`,
      if `A` is an input argument or `A′` is an input argument,
      then `T` and `T′` are not mutually exclusive.

b) Signature `P` is *less than* `P′` if
   1) `P` and `P′` overlap, and
   2) For every argument `A` with type `T` of `P` and corresponding argument `A′` with type `T′` of `P′`,
      if `A` is an input argument or `A′` is an input argument,
      then `T <: T′`.

c) A set of signatures is *unambiguous* if
   1) For every pair of distinct signatures `P` and `P′` in the set of signatures,
      if `P` overlaps `P′`
      then
         `P` less than `P′`, or
         `P′` less than `P`.

d) The uniqueness conditions are as follows:
   1) For every class, the signatures of the instance-level responsibilities shall be unambiguous, and the signatures of the class-level responsibilities shall be unambiguous.
   2) For every pair of parallel classes, the union of the signatures of the instance-level responsibilities shall be unambiguous, and the union of the signatures of the class-level responsibilities shall be unambiguous.

7.5.4 Static type checking

Static type checking uses the same signature matching and search order, but with static types instead of dynamic.

a) Static type checking requires that identifiers, not variables, be used for all class names and property names.
b) The search starts at the static type of the receiver. Each signature is tested for a match using the static types of the message argument values and the static types of the property signature arguments.
c) If no static type was declared, no type check is done.

7.5.5 Message to super

a) A message to `super`,
```
        Self super has Responsibility
```
modifies the search order for message resolution.
b) A message to `super` may be used only within a realization.
c) The search starts as though the search had failed at the class for which the realization is defined.

7.5.6 Visibility

a) Private responsibilities are accessible only by messages to `Self` from realizations of the class.
b) Protected responsibilities are accessible only by messages to `Self` or `Self super` within a subclass from realizations of the class or a subclass.
c) For private, the static type of `Self` shall be the same as the class of the responsibility.
d) For protected, the static type of `Self` shall be a subtype of the class of the responsibility.

7.5.7 Read-only

a) Within a read-only responsibility, no value of any instance may be changed.

7.5.8 Constant

a) A constant responsibility gives the same result for the same input arguments, regardless of the values of the instances.

7.6 Assignment

a) An assignment such as
```
        V2 is V1
```
is a proposition that proposes that `V2` is the same as `V1`. In the common case, `V2` is an uninstantiated variable and `V1` is a value. The solution to the proposition is to make `V2` the same as `V1`, if the typing rules permit.
b) If `V1` has type `T1` and `V2` has type `T2`, the proposition
```
        V2 is V1
```
shall satisfy `T1 <: T2`.
c) If `V1` and `V2` are both values, then `V2 is V1` is true if `V1 == V2`.
d) If `V1` is uninstantiated, then `V1` and `V2` are equated. For dynamic (run time) type checking, any assignment that causes a variable with type `T` to be assigned a value that does not have type `T` raises an exception.
e) Assignment to variables (local variables, arguments, and `Self`) is nondestructive. If `V2` already has a value, that value is not changed. In contrast, a message such as

```
        I has p:= V
```
does change the value of instance `I`'s property `p`.

7.7 Propositions

A proposition is true or false. If true, it may (and typically does) instantiate variables or, in other words, find a solution for their unknown values. A solution is a set of values for the variables in the proposition that makes the proposition true.

a) If the proposition is false, there is no solution and no variables are instantiated.
b) If a proposition is evaluated when all its variables are already instantiated, then it shall be true if those values are a solution; otherwise, false.
c) The syntax of a proposition is as follows:
   1) `Proposition` →
```
true
    or false
    or Object { super } Having { PathExpr } ResponsibilityValue
    or Object.. { PathExpr } ResponsibilityValue
    or Object RelOp Object
    or SimpleObject = SimpleObject
    or variable Variable { : TypeLiteral } { Being Object }
    or Variable { : TypeLiteral } Being Object
```
   2) `Having` → **`has`** *or* **`had`**
   3) `Being` → **`is`** *or* **`was`**
d) A `had` or `was` proposition may be used only within a post-condition.

7.7.1 Assert

a) The proposition
```
        assert Proposition
```
   is true if `Proposition` is true.
b) It is false if `Proposition` is false, in which case an exception is raised.
c) An `assert` does not make the `Proposition` true. It simply tests whether it is true.
d) An `assert` is read-only; any updates done by the `Proposition` are backed out, whether the `assert` succeeds or fails.

7.7.2 Negation

a) A proposition such as
```
        not Self has p: [ X, Y ]
```
   is true if
```
        Self has p: [ X, Y ]
```
   is false.

This is called *negation as failure*. It is based on the *closed world assumption*: whatever is true is known to be true by the model, so anything not known to be true by the model shall be false.

This topic is covered more thoroughly in the formalization (see Clause 10).

7.7.3 Equality

The definitions of equality are based on the idea that for two things to be equal, they must be indistinguishably substitutable one for the other in any context.

a) For two variables `I1` and `I2`, each bound to a state class instance,
```
        I1 == I2
```
if `I1` and `I2` are bound to the same instance.

b) For two variables `V1` and `V2`, each bound to a value class instance,
```
        V1 == V2
```
if they:
1) Have a common superclass, and
2) Have the same value for all the properties of a uniqueness constraint.

c) `X1 != X2`
is defined as equivalent to
```
        not ( X1 == X2 ).
```

7.7.4 Ordering comparisons

a) The proposition
```
        Object RelOp Object
```
is defined based on the total ordering described in 7.12.

b) The `RelOp` properties are properties of `object` and value in the metamodel and may be overridden.

7.8 Sentences

a) The syntax for `Sentence` is as follows:
1) `Sentence` →
```
Proposition
    or not Sentence
    or Sentence, Sentence
    or Sentence or Sentence
    or if Sentence then Sentence endif
    or if Sentence then Sentence else Sentence endif
    or forall Sentence : Sentence
    or for Accumulator all Sentence : Sentence
    or exists Sentence
    or assert Sentence
```

7.8.1 Conjunction

a) A conjunction
```
        P, Q
```
is true if `P` is true and `Q` is true.

b) If `P` and `Q` have no side effects and raise no exceptions, then `P, Q` is true if and only if `Q, P` is true.

c) If `P` or `Q` is false, then no side effects due to either shall occur.

d) If `Q` or `P` have side effects or raise exceptions, `Q, P` may give a different result.

7.8.2 Disjunction

a) A disjunction
```
        P or Q
```
is true if `P` is true or `Q` is true.

b) If `P` and `Q` have no side effects and raise no exceptions, then `P or Q` is true if and only if `Q or P` is true.

c) If `Q` or `P` have side effects or raise exceptions, `Q or P` may give a different result.

7.8.3 Implication

a) An implication
```
        if S1 then S2 endif
```
is true if `S1` is false or `S2` is true.

b) At most one solution is obtained for `S1`.

c) If there is a solution, then `S2` is evaluated with common variables bound to the values determined by the solution.

d) If `S1` in fact has multiple solutions, some one of them is used and the others are ignored.

e) An RCL interpreter shall offer an option of raising an exception if the condition sentence has more than one solution.

7.8.4 Conditional

a) A conditional
```
        if S1 then S2 else S3 endif
```
is equivalent to
```
        if S1 then S2 endif, if not S1 then S3 endif
```

7.8.5 Bounded quantification

a) A bounded universal quantification
```
        forall ( S1 ) : ( S2 )
```
is true if, whenever `S1` is true, `S2` is true.

b) For each solution obtained for `S1`, `S2` is evaluated with common variables bound (temporarily) to the values determined by the solution.
   1) If `S2` is false, then the `forall` is false.
   2) If `S2` is true, then
      i) The next solution for `S1` is obtained, and
      ii) `S2` is evaluated with that solution.

   This procedure continues until some solution of `S1` causes `S2` to be false, or there are no more solutions to `S1`.

c) If `S1` has no solutions, the `forall` is true.

d) A `forall` never leaves any variables it temporarily instantiates bound to a value.
   1) At the conclusion of the `forall`, variables in S1 or S2 have exactly the same value (no value) as they had before the `forall` was evaluated.
   2) At the conclusion of a successful `forall`, all side effects due to S2 remain.
   3) S1 is not permitted to have side effects; and, if any occur, an exception is raised.

e) A bounded existential quantification
```
        exists ( S1 )
```
is true if `S1` is true.

f) If `S1` has no solutions, the `exists` is false.

g) An `exists` never leaves any variables it temporarily instantiates bound to a value.
   1) At the conclusion of the `exists`, variables in `S1` have exactly the same value (no value) as they had before the `exists` was evaluated.
   2) At the conclusion of a successful `exists`, all side effects due to `S1` remain.

7.8.6 Bounded accumulation

a) A bounded accumulation
```
        for Accumulator all ( S1 ) : ( S2 )
```
is a `forall` that accumulates results in the `Accumulator`.

For example, to compute the `Sum` of the members of a `List` of integers,

```
Acc is accumulator(initial: 0, final: Sum),
for Acc all (List has member: X):
  ( Acc..current is Acc..previous + X).
```

7.9 Type checking

Static type checking is done on the source text of the RCL using the declared, static types. Dynamic type checking is done during model execution using dynamic types and possibly static types.

a) An implementation shall offer, but not require, static type checking for
   1) Message resolution,
   2) Assignment, and
   3) Argument passing.
b) It shall be possible to execute a model even if static type checking is not done or it fails, in which case dynamic type checking shall be done on assignment and argument passing.
c) The combination of static and dynamic type checking used to check for conformance to the declarations of visibility, updatable, and constant is implementation-dependent.

7.10 Constraint checking

7.10.1 Constraints

a) For a constraint responsibility, the effective constraint is the conjunction of the constraint with all overridden constraints of the same name.
b) A constraint may be checked
   1) Explicitly by sending a message for it, or
   2) Automatically using the options described under 7.10.3.

7.10.2 Pre-conditions and post-conditions

Pre-conditions and post-conditions are, in concept, part of the interface, not the realization, but syntactically they are stated with the realization because they need access to argument values and the property values of the receiver.

a) The syntax for pre-conditions and post-conditions is as follows:

   `Body → { pre Sentence, } * Sentence { , post Sentence }*`
b) The pre-condition is
   1) The disjunction of the `pre Sentences`, or
   2) **`true`** if there are no `pre Sentences`.
c) The post-condition is
   1) The conjunction of the `post Sentences`, or
   2) **`true`** if there are no `post Sentences`.
d) In a post-condition, a message such as

   `I had P: V`

   or

   `V was I..P`
   1) Gets the old property values as they were before the body was evaluated.
   2) May be used only within post-condition sentences.
e) The same results should be obtained when evaluating with pre-condition and post-condition checking turned off or on (as long as no exceptions are raised). To this end, the following rules apply:
   1) `Pre` and `post` are read-only.
   2) No variable used in the `Sentence` may be used in the `pre` or `post Propositions` except `Self` and the arguments.

f) The pre-conditions and post-conditions stated for a property are combined with those of the properties it overrides to form the *effective pre-conditions and post-conditions*.
   1) A model may be evaluated with pre-condition and post-condition checking turned on or off.
      i) If turned on and an effective pre-condition or effective post-condition is not met, an exception is raised.
      ii) If the `Sentence` evaluates to `false,` the post-condition is not checked.
   2) A total (mandatory) property shall succeed if its pre-condition is met.

7.10.3 Constraint checking options

a) An RCL interpreter shall offer the option of checking the following and raising an exception if the constraint is not met:
   1) A single-valued property has at most one value when the value is requested.
   2) A mandatory property has a value when the value is requested.
   3) A collection valued property with a cardinality constraint shall satisfy the constraint when the property value is requested.
   4) A value class instance specified by a literal (including the `with` form) meets all constraints.
b) These options are in addition to the type checking and pre-condition and post-condition checking options.

7.11 Query

During the evaluation of a model, query RCL is used by the modeler testing the model.

a) The syntax for query RCL is:
   `QueryRCL` → `Sentence.`
b) In query RCL, the `Sentence` may be true or false.
   1) If true, it solves for the values of the variables in the message.
   2) There may be more than one solution.
c) In query RCL, the following are implementation-dependent:
   1) The manner of entering the query RCL,
   2) The manner of displaying whether a `Sentence` is true or false, and
   3) The display of the solutions.
d) Side effects due to the `Sentence` are discarded if the `Sentence` is false.
e) Side effects due to a true `Sentence` may be optionally retained in an implementation-dependent manner.

7.12 Total ordering

a) Variables and objects are totally ordered from low to high in the following way:
   1) Variables in an implementation-dependent order.
   2) Real numbers from minus infinity to plus infinity.
   3) Integer numbers from minus infinity to plus infinity.
   4) Identifiers and strings in an implementation-dependent order, subject to the following rules:
      i) The empty identifier or string precedes all nonempty.
      ii) Uppercase letters A through Z are in ascending order.
      iii) Lowercase letters a through z are in ascending order.
      iv) Digits from 0 through 9 are in ascending order.
      v) An identifier or string `S1` precedes an identifier or string `S2` if the first character of `S1` is less than the first character of `S2`, or the first characters are equal and the rest of `S1` precedes the rest of `S2`.
   5) Lists and pairs in lexographic order.
      i) A pair `A:B` precedes a pair `C:D` if `A` precedes `C`, or `A` equals `C` and `B` precedes `D`.

ii) A list `A` is less than a list `B` if the first element of `A` precedes the first element of `B` or the first elements are equal and the rest of `A` precedes the rest of `B`.

6) All other objects in an implementation-dependent order.

b) This order determines the result of comparing two objects (unless the `RelOp`s are overridden).

7.13 Implementation-dependent

a) Anything that is specified as implementation-dependent may be implemented in any way the implementer sees fit.

7.13.1 Error conditions

a) The action taken by an RCL interpreter to “raise an exception” is implementation-dependent.

7.13.2 Numeric characteristics

a) Minimum and maximum values, overflow, and so on are all implementation-dependent.

7.14 Lexical characteristics

7.14.1 Character set

a) The following defines the RCL character set:

1) `Digit` → `0 through 9`
2) `Lowercase` → `a through z`
3) `Uppercase_` → `A through Z` *or* `_`
4) `AlphaNumeric` → `Uppercase_` *or* `Lowercase` *or* `Digit`
5) `Variable` → `Uppercase_ { AlphaNumeric }*`
6) `Integer` → `{ - }{ Digit }+`
7) `Real` → `Integer.{ Digit }+ { e Integer }`
8) `Whitespace` → `any of space tab newline return formfeed backspace`
9) `SpecialCharacterExceptQuote` →
   any of `! @ # $ % ^ & * ( ) + | - = \ { } [ ] : ; < > ,. ? /`
10) `CharacterExceptQuote` →
   `AlphaNumeric`
   or `Whitespace`
   or `SpecialCharacterExceptQuote`
11) `Character` → `CharacterExceptQuote` *or* ‘ *or* “
12) `CharacterExceptDoubleQuote` → `CharacterExceptQuote` *or* ‘
13) `CharacterExceptSingleQuote` → `CharacterExceptQuote` *or* “
14) `SingleQuoteCode` → ‘‘
15) `UnquotedIdentifier` → `Lowercase` *{* `AlphaNumeric` *}**
16) `Identifier` →
   `UnquotedIdentifier`
   or ‘*{* `CharacterExceptSingleQuote` *or* `SingleQuoteCode` *}+* ‘
17) `QualifiedName` → `Identifier` *{* `:` `Identifier` *}**
18) `Constant` → `Integer` *or* `QualifiedName`
19) `DoubleQuoteCode` → “”
20) `String` → “*{* `CharacterExceptDoubleQuote` *or* `DoubleQuoteCode` *}** “
21) `PropertyOperator` → `:` *or* `:=` *or* `:!=` *or* `:+=` *or* `:-=`
22) `UnaryOp` → `+` *or* `-`
23) `BinaryOp` → `RelOp` *or* `+` *or* `-` *or* `*` *or* `/` *or* `mod` *or* `**` *or* `^`
24) `RelOp` → `<` *or* `<=` *or* `==` *or* `>=` *or* `>` *or* `!=`

b) If `xxx` is an unquoted identifier, then `xxx` and `'xxx'` are interchangeable.
c) Whitespace outside quotes serves only to separate tokens.

7.14.2 Comments

a) Comments are initiated by the characters
`--`
outside single or double quotes.
b) The comment extends to the end of the line.
c) Comments are treated as whitespace.

7.15 RCL syntax

7.15.1 RCL syntax conventions

The syntax for RCL is given using a version of Backus-Naur form (BNF) in which a series of production rules say how a nonterminal symbol produces other nonterminal or terminal symbols. All the legal terminal symbol sequences in the language can be produced by starting with the top symbol (`RCL`) and applying a series of production rules.

a) An extended notation is used to simplify the grammar:

1)	→	is the *produces* symbol
2)	*or*	means alternative
3)	*{ … }*	means the contents are optional
4)	*{ … }**	means zero or more repetitions of the contents
5)	*{ … }+*	means one or more repetitions of the contents
6)	**`keyword`**	is a terminal of the language
7)	xxx_name	is syntactically an identifier (denoting an xxx)
8)	xxx_qname	is syntactically a qualified name (denoting an xxx)
9)	xxx_var	is syntactically an xxx or a variable (bound to an xxx)
10)	*italics*	is an informal comment, such as *through* or *any of*

b) Whitespace outside single or double quotes serves only to separate tokens.
c) Comments are treated as whitespace.

7.15.2 Operator priority and associativity

The priority and associativity of the operators in an expression determine the nonterminal symbol that produces the expression.

a) All operators are
   1) Prefix, for example, negation as in `-7`
   2) Infix, for example, plus as in `3+4`
b) All operators are
   1) Left associative, for example, plus as in `a+b+c`, which is equivalent to `(a+b)+c`
   2) Right associative, for example, pair as in `a:b:c`, which is equivalent to `a:(b:c)`
   3) Nonassociative, for example, `'!='` as in `X != 10 != 17`, which is invalid.
c) All operators have a priority, from loose to tight. For example, `'+'` is looser than `'*'`, so `1+3*4` is equivalent to `1+(3*4)`.
   Table 12 shows the operators from loose at the top to tight at the bottom. Operators on the same line have the same priority.
d) Parentheses override the priority and associativity of the operators. For example, the grouping of `a..b:2..c` as `(a..b):(2..c)` shall be overridden by `a..(b:2)..c`.

Table 11—Operator priority

prefix non	prefix right	infix left	infix non	infix right
				`or`
				`,`
	`not assert`			
			`= < <= == >= > !=`	
		`as`		
		`with`		
			`:= :!= :+= :-=`	`:`
		`+ -`		
		`*/ mod`		
				`** ^`
	`+ -`			
		`..`		
`#`				

7.15.3 BNF

a) The BNF for RCL follows:

1) `RCL` →
`DeclarationRCL`
or `QueryRCL`
or `RealizationRCL`

2) `DeclarationRCL` →
`stateClass_qname : OID` **`has`** `ResponsibilityName` *{* `: SimpleObject` *}*.

3) `OID` →
`# OIDTerm`

4) `OIDTerm` →
`Constant`
or `Type`
or *{* `+ or -` *}* `OIDTerm`
or `OIDTerm PropertyOperator OIDTerm`
or `[OIDTerm` *{* `, OIDTerm` *}** `]`

5) `Type` →
`Variable`
or `#class_qname`
or `#(class_qname : [ Type` *{* `, Type` *}** `] )`

6) `QueryRCL` → `Sentence.`

7) `RealizationRCL` → `Head` $\textbf{if}_{\textbf{def}}$ `Body.`

8) `Head` →
`class_qname : Variable` **`has`** `ResponsibilityName` *{*
`PropertyOperator Arguments` *}*

9) `Body` → *{* **`pre`** `Sentence,` *}* * `Sentence` *{* `,` **`post`** `Sentence` *}**

10) `Sentence` →

```
    Proposition
       or not Sentence
       or Sentence, Sentence
       or Sentence or Sentence
       or if Sentence then Sentence endif
       or if Sentence then Sentence else Sentence endif
       or forall Sentence : Sentence
       or for Accumulator all Sentence : Sentence
       or exists Sentence
       or assert Sentence
11) Proposition →
    true
       or false
       or Object { super } Having { PathExpr } ResponsibilityValue
       or Object.. { PathExpr } ResponsibilityValue
       or Object RelOp Object
       or SimpleObject = SimpleObject
       or variable Variable { : TypeLiteral } { Being Object }
       or Variable { : TypeLiteral } Being Object
12) Having → has or had
13) Being → is or was
14) Object →
    SimpleObject
       or Literal
       or Object.. { PathExpr } PropertyExpr
       or UnaryOp Object
       or Object BinaryOp Object
       or Object where Sentence
       or Object as TypeLiteral
15) SimpleObject →
    Variable
       or String
       or Identifier
       or Number
       or #Constant
       or true
       or false
       or SimpleObject : SimpleObject
       or SimpleObjectList
       or {}
16) SimpleObjectList →
    [{SimpleObject { , SimpleObject }* } ]
       or [ SimpleObject | SimpleObjectList_var ]
17) Literal →
    class_qname_var with ( ResponsibilityValue { ,
    ResponsibilityValue }* )
       or class_qname( ResponsibilityValue { ,
          ResponsibilityValue }* )
       or CollectionName(Objects)
       or [ Objects ]
       or [ Object | List ]
       or { { Objects } }
       or { Object | Set }
18) List →  Object           that is a list
```

```
19) Set → Object              that is a set
20) PathExpr → { PropertyExpr { : SimpleObject } .. }+
21) PropertyExpr →
    ResponsibilityName_var
       or PropertyNameSingular(Objects)
22) Objects → Object { , Object }*
23) ResponsibilityValue →
    ResponsibilityName_var { PropertyOperator SimpleObject }
       or ResponsibilityOid { : SimpleObject }
       or PropertyNameSingular ( Objects )
24) ResponsibilityName → Identifier or Identifier( s )
25) PropertyNameSingular → Identifier
26) ResponsibilityOid → Variable
27) Arguments → Argument or [ Argument { , Argument }* ]
28) Argument → Variable { : TypeLiteral }
29) Accumulator → Object    that is an accumulator
30) TypeLiteral →
    any
       or bot
       or class_qname
       or Variable
       or class_qname
       or parametricVClass_qname(TypeLiteral {, TypeLiteral }* )
```

8. Model infrastructure constructs

The constructs described in other clauses can be utilized in different ways for different reasons. Among the common reasons are to

a) Understand and document the current scope and rules of some subject of interest,
b) Analyze and propose a potential scope and rules of an area of interest, and
c) Serve as a blueprint for an information system that supports or will support an area or subject of interest.

Experience with key-style IDEF1X (Clause 9) and similar languages shows that certain documentation and organization concepts span almost all of the uses of these constructs, much as, independent of the content of the book, almost all books have a title, an author name, and a date published. Additionally, almost all books are organized into chapters and have their pages numbered. In the same way that a common organizational paradigm has emerged for books, so has one emerged for the constructs described in this standard.

This clause gives all practitioners a common baseline from which to organize and document their work. It is limited to four key organizational concepts: view, environment, glossary, and model. This standard does not prohibit the use of additional concepts to organize the constructs described in this standard. The following topics are discussed in this clause to provide an infrastructure for modeling:

a) **View:** A view is composed of the language constructs documented in the earlier clauses. A *view* is a collection of classes, relationships, responsibilities, properties, constraints, and notes (and possibly other views), assembled or created for a certain purpose and covering a certain scope. A view may cover the entire area being modeled or only a part of that area.
A view specifies the structuring and declarations of the classes it contains. The allocation of properties and constraints to classes, the taxonomy of classes, and the sentences for properties and constraints are specified only within views.

Each view shall be of one style: identity style or key style. Identity-style model constructs are discussed in Clauses 5 and 6, and the identity-style view is described here (see 8.1). The key-style view is discussed in Clause 9.

b) **View Level**: A view may have an identified level. Valid levels depend on the view style. View level for identity style is discussed in 8.2.[67]
c) **Environment**: In IDEF1X, an *environment* is a concept space—an area in which a concept has an agreed-to meaning and one or more names. Every view is developed for a specific environment. The environment controls the scope of the view as well as the names and properties given to its elements. The constituent elements of a view can only be understood, used, and referred to within a frame of reference. Environment provides that frame of reference (see 8.3).
d) **Glossary**: An environment is supported by an *environment glossary*, which is the collection of the names and descriptions of all defined concepts (views, classes, relationships, responsibilities, properties and constraints) within that environment. The glossary describes the concepts that were specified in views. In other words, the glossary reflects the <u>descriptions</u>, or meanings, of the concepts but it does not "contain" the concepts themselves. For example, it is the meaning of the `customer` class that is in the glossary, but not the structuring and declarations of the `customer` class itself. The principle motivation for the glossary is to ensure consistent semantics across all views in an environment.

   A *model glossary* is the collection of the names and descriptions of all defined concepts that appear within the views of a model. Since a model may span environments, the scope and content of its glossary are determined by the views contained in the model (see 8.4).
e) **Model**: An IDEF1X *model* consists of one or more views along with textual descriptions of the views and view components (classes, properties, etc.) called out in the views (see 8.5).

8.1 View

A *view* is a collection of classes, relationships, responsibilities, properties, constraints, and notes (and possibly other views), assembled or created for a certain purpose and covering a certain scope. A view may cover the entire area being modeled or only a part of that area. Views are typically presented as graphic diagrams. The notion of view is needed because a class is too small and a model or environment can be too large to reason about effectively.

A view exists together with any number of other potentially overlapping views within a single environment (see 8.3). An environment contains defined concepts shared across one or more views. A view may use an existing description of a concept (class, property, etc.); it may also separately describe concepts as they apply within the view (see 8.4).

Views emerge and change over time. A view may include concepts described in other views. One view may be a subset of another view or it may be a composite formed from other views.

8.1.1 View semantics

8.1.1.1 Shadow class

A view provides the specification for the structuring of the constituent parts (classes) that make up the view. The allocation of properties, relationships, and constraints to a class, the taxonomy of classes, the composition of contained views, and the sentences for properties and constraints are specified only within a view.

However, it is often desirable to depict a class in views other than the one in which it is specified. A class presented in a view that is specified in some other view is referred to as a *shadow class* in that view.

[67]View level for key style is discussed in 9.10.

8.1.1.2 Subject domain

A view taken as a whole is referred to as a *subject domain.* A subject domain is an area of interest or expertise. The responsibilities of a subject domain are an aggregation of the responsibilities of a set of current or potential named classes. A subject domain may also contain other subject domains. A subject domain encapsulates the detail of a view; there is a one-to-one correspondence between a subject domain and a view.

The use of subject domains allows a hierarchy of views to be formed, with increasing detail at each level. When a subject domain appears in a view, only the "outside" can be seen. When a subject domain appears as a view, the internal detail can be seen. This concept is illustrated in Figure 80.

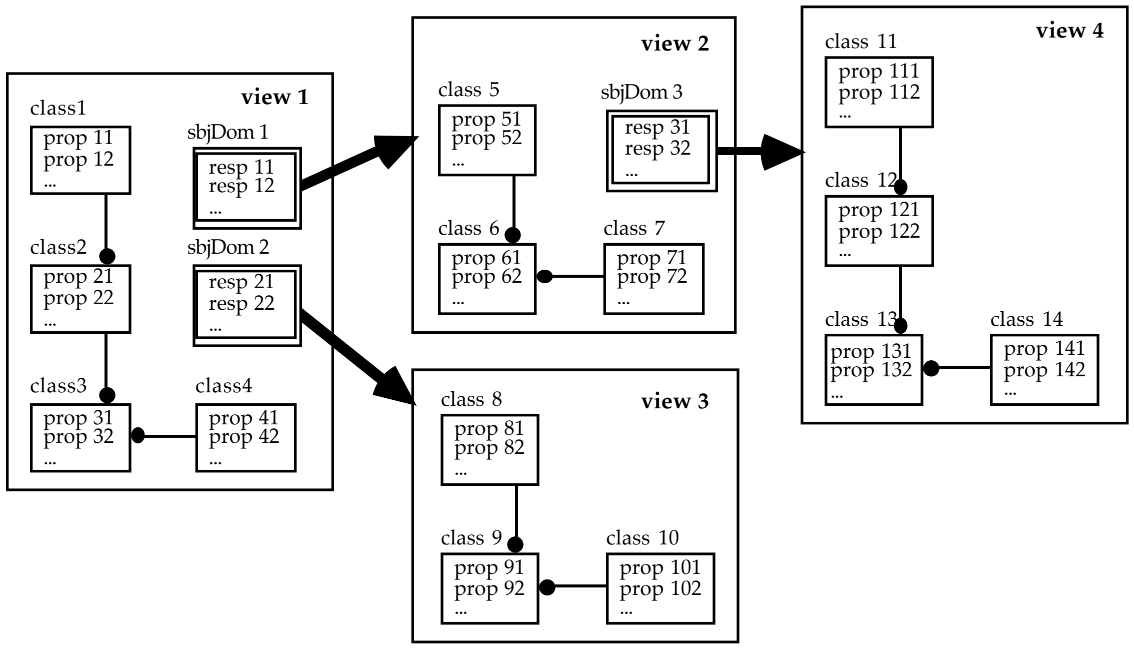


Figure 80—Subject domains and views

A preliminary step in understanding the nature of a subject domain is to describe the subject's overall purpose and scope and to identify its most abstract responsibilities. A *subject domain responsibility* is a generalized concept that the analyst discovers by asking "in general, what do instances within this subject domain need to be able to do or to know?" The classes and contained views in a view together supply the knowledge, behavior, and rules that make up the subject. These are collectively referred to as the subject domain's responsibilities. Subject domain responsibilities are not distinguished as views or classes during the early stages of analysis.

Subject domains are typically used in initial model development (e.g., Survey level) to allow reasoning about broad concepts, although they are not restricted to this level and use. For example, the subject domain tends to become a "natural allocation" when partitioning the model for development since there is the notion of tighter collaboration between the classes of a subject domain.

8.1.1.3 Types of view

Views may be formed for a variety of reasons. To show the classes and responsibilities covered by a topic, or general subject area, a *subject-based view* can be formed. A subject-based view includes all and only the constructs that describe the subject area.

To show the classes and properties needed to support a specific business function, a *function-based view* can be formed. A function-based view includes all and only the constructs needed to support that function.

To show classes and public responsibilities, a *consumer view* can be formed. To show classes, public responsibilities, and protected or private properties, a *producer view* can be formed.

This standard specifies no explicit view types. These types are only illustrative of typical usage.

8.1.1.4 Internal consistency

A nonempty set of instances for which all constraints of a view are satisfied is called a *consistent set of instances* for that view. A view is *internally consistent* if and only if a consistent set of instances exists for that view. Figure 81 depicts a view that is internally inconsistent because the constraint `same` requires the parents of `d` to be the same instance of `c`, while the generalization semantics require the classes `c1` and `c2` to be mutually exclusive.

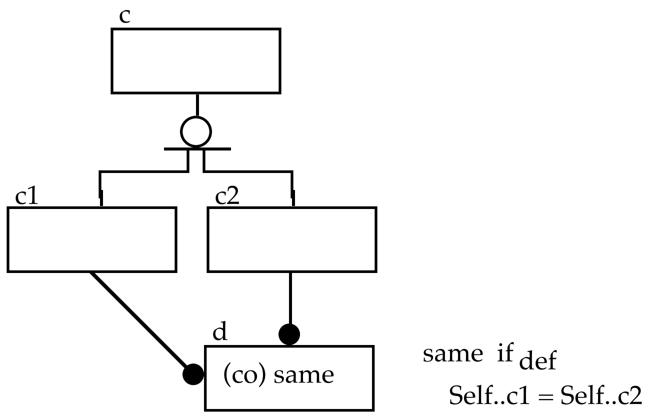


Figure 81—Inconsistency within a single view

8.1.2 View syntax

A view may be shown graphically either as a single subject domain or in a view diagram of the contained classes, responsibilities, relationships, properties, constraints, and notes.

8.1.2.1 View diagram graphic

a) The standard graphic presentation for a view diagram of the classes, responsibilities, relationships, properties, constraints, notes, and subject domains that compose a view is illustrated along with the description of these constructs in Clauses 5 and 6.

8.1.2.2 Alternative presentation mode

a) Constructs may be "hidden" (omitted from graphic presentation) in a view.

Examples of the constructs that may be "hidden" include attributes, relationship verb phrases, participant properties, operations, constraints, cardinality specifications, discriminator properties, note identifiers, and the prefix comma list and suffix comma list for properties.

b) Suppressing selected elements of the syntax allows a view to be presented in alternate modes for different purposes and audiences. This standard specifies no explicit alternate presentation modes. The examples below are only illustrative of possible usage.
Examples of alternate presentation modes include views displayed with all of their protected and private properties hidden and views displayed omitting the "`(derived)`" designation from the property signatures. These presentations might be appropriate for users of the classes in the view.

c) In any form of abbreviated presentation, all applicable syntactical and semantic rules shall still be enforced; some information is merely not displayed.

8.1.2.3 Subject domain graphic

a) A subject domain shall be represented as a double-bordered rectangle, as shown in Figure 82.

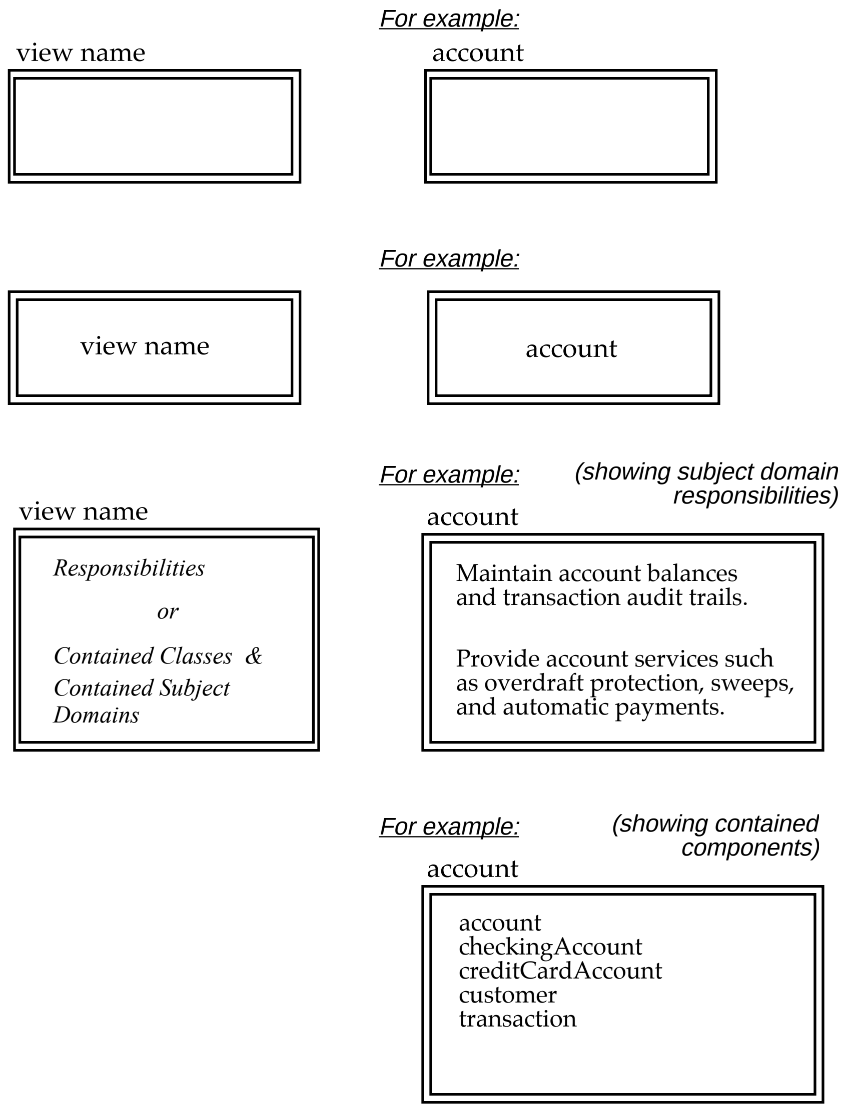

For example:
view name
account
For example:
view name
account
For example:
(showing subject domain responsibilities)
view name
Responsibilities
or
Contained Classes &
Contained Subject Domains
account
Maintain account balances and transaction audit trails.
Provide account services such as overdraft protection, sweeps, and automatic payments.
For example:
(showing contained components)
account
account
checkingAccount
creditCardAccount
customer
transaction

Figure 82—Alternative representations of subject domain

b) The subject domain name shall be placed
   1) Inside or above the rectangle when no responsibilities, contained classes, or contained subject domain names are shown, or
   2) Above the rectangle when text (responsibilities, or contained classes and subject domain names) is shown inside the rectangle.

      When the latter style is used, the presentation can be thought of as a “toggle.” Either subject domain responsibilities are displayed, or the subject domain’s contained classes and contained subject domains are presented.

8.1.2.4 Shadow class graphic

a) The graphic for a shadow class shall be a *reference line* (i.e., repetitions of one long dash followed by two short dashes) for the shape appropriate to its class,[68] as shown in Figure 83.

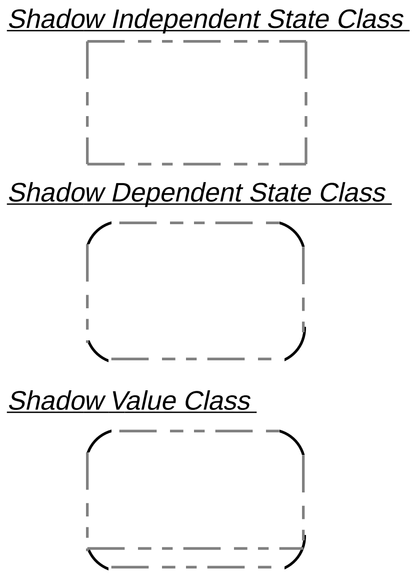


Figure 83—Shadow class graphic syntax

8.1.3 View rules

8.1.3.1 Naming

a) A view shall have a name.
b) The view name shall appear on any presentation of the view, e.g., the view diagram.
c) A view name may be a *simple name* or a *fully qualified name*.
d) A view shall have both a simple name and a fully qualified name.
   1) A view not included within another view shall have a simple name that is the same as its qualified name.
   2) A view included within another view shall have a qualified name as follows:

      A view `V′` included within another view with the fully qualified name `Vn` shall have the fully qualified name `Vn:Vsn′`, where `Vsn′` is the simple name of the included view.

[68] “Shapes appropriate to class” are described in 5.2.2 and 5.3.2.

8.1.3.2 Description narrative

a) A narrative describing the view shall be recorded.
b) The purpose of the view shall be recorded.
c) The scope of the view shall be recorded.

8.1.3.3 Style

a) A view shall have an identified style, designated as either identity style or key style (Clause 9).

8.1.3.4 Level

a) A view may have an identified level appropriate to its style, conforming to the specification of levels stated in either 8.2 (for identity style) or 9.10 (for key style).
b) If a level is specified for a view, that view may not include constructs not applicable to its level, as specified in either 8.2 (for identity style) or 9.10 (for key style).

8.1.3.5 Composition

a) A view may contain a mixture of classes (state and value classes) and subject domains.
b) A view may be contained as a subject domain within other views.
c) No view may contain or be contained within itself, either directly or indirectly (i.e., no cycles).
d) A view may be composed of a mix of view responsibilities, classes, and other views, i.e., it is not restricted to having a homogeneous composition.
e) A defined concept may appear in any number of views.
f) A view may import a class from another view by naming the class by its fully qualified name.

8.1.3.6 View-component label

a) Every defined concept in a view shall be labeled with its name or (where applicable) one of its alias names.
b) If a defined concept has more than one name in the environment glossary, within a given view it shall be referred to consistently by only one of its names, regardless of how often it appears in the view diagram.[69]
c) The label for a defined construct within a view shall conform to the lexical rules for naming stated in 4.2.3.
d) Within a view, no two classes or contained views (subject domains) shall have the same label.
e) Within a view, no two responsibilities of the same class shall have the same signature.
f) Within a view, no two relationships that relate the same two classes shall have the same set of names and role names.
g) When a class or responsibility is referred to in RCL, the name label assigned to the defined concept in the view shall be used.
h) When an imported class appears in a view, the fully qualified name for the class shall be shown in the view.

8.1.3.7 Internal consistency

a) If the objective of the view is that it be internally consistent, it should be possible to demonstrate that a consistent set of instances exists.

[69]The appearance of a defined concept multiple times on a view diagram is permitted to make the display more convenient.

8.2 Identity-style view level

There are distinct levels of view in identity-style modeling.[70] Each level has to balance the admittedly conflicting goals of any view: be understandable to users and be useful to developers. Each level is intended to be distinct, specified in terms of the modeling constructs used.

Any view may be restricted to one level. This has two advantages. First, limiting each level to the appropriate set of modeling constructs promotes modeling what is appropriate to the level and only what is appropriate to the level. Second, having distinct levels provides clear work product definition for management. However, there is no prohibition against forming a view with no level specified and using constructs from many levels within that view when this approach is useful.

The views in adjacent levels relate to each other by a mapping or transformation. The mapping or transformation is enabled by employing a consistent set of modeling concepts. Levels do not imply a particular pattern of development, e.g., waterfall, iterative, or fountain. The methodological development pattern determines the scope of the views and the order in which they are produced, but not their content. The content of a level of view is independent of the methodological development pattern. Table 13 summarizes the levels of view in identity-style modeling.

Table 12—Summary of view levels (identity style)

Level of view	Characteristic modeling constructs	Primary intent
1	Subject domains, responsibilities of subject domains **(Survey level)**	Specify and manage major areas of reusable assets and the applications and projects that use them.
2	Survey level plus frameworks and patterns of (super) classes, responsibilities of classes **(Integration level)**	Architect and integrate features, prototypes, and releases within a project as well as across projects and applications.
3	All classes, relationships, properties, constraints **(Fully specified level)**	Complete specification of all semantics for a project or project release, independent of the implementation platform and language.
Technology-dependent levels	Database specifications –or– Classes (programming language, database, properties, constraints) **(Implementation level)**	Complete specification in terms of implementation platform and language constructs.

8.2.1 Identity-style view level semantics

8.2.1.1 Level 1 (survey level)

The survey level deals with subject domains, rather than classes. Classes are too fine-grained for early reasoning. While it may be that a subject domain is named for a principal class, the responsibilities of the subject domain will be seen in subsequent views to be distributed over multiple classes within the subject domain. Therefore, the subject domain is named in the survey view rather than the class. To allow modeling

[70]See 9.10 for a discussion of the view levels for key-style modeling and for a comparison of view-level concepts and constructs.

of complex systems, subject domains are recursively specified as containing other subdomains or, ultimately, classes.

The name “survey” is used instead of “business” or “enterprise” because it is useful to develop this level of view for technical or support areas as well as business areas. For example, a survey-level view would be very useful (even essential) for a distributed system of repositories and development tools.

8.2.1.2 Level 2 (integration level)

The integration level supports representation and reasoning about the most important concepts in the subject domain. The classes in this level are superclasses or other important, discovered classes—at least initially. A class is “discovered” in the sense that it represents a concept already present in the minds of the people who understand the subject domain. The integration level view also includes classes that have been “invented” (typically by abstracting from the discovered classes) to promote system resiliency in the face of change.

Initially, aggregate responsibilities of classes are specified, rather than individual properties. When properties are depicted, they are not distinguished as attributes or operations.

The integration level must be specific enough to support technical integration decisions. Just as a consistent key structure was a prerequisite for integrated databases for the original IDEF1X models, the *frameworks* of this level provide the specificity needed for sharing and integration. Some frameworks, such as those for the presentation layer, may be supplied by vendors. Others are built by the enterprise.

This level is in many ways the most important and the most difficult. It requires deep insights into the needs of the enterprise and the technical ability to be both abstract and precise.

When fully specified views are available over the scope of the integration level views, the integration level views can be updated to include all the classes, responsibilities, and properties important to integration and reuse. For example, a common technique is to specify an abstract superclass with properties possessed by all its subclasses. The integration level includes abstract superclasses, but might not include their subclasses.

8.2.1.3 Level 3 (fully specified level)

The fully specified level completely specifies all classes. The fully specified view begins as a subset view of an integration level view. Responsibilities are refined into properties (attributes, participant properties, and operations) and constraints. The interfaces of individual properties are specified, in terms of both semantics (the meaning of the property) and syntax (the signature).

A view is fully specified if an implementer can choose <u>any</u> implementation for whatever is not specified (so long as it is consistent with what <u>is</u> specified) and the result will be acceptable. This criterion is the decision rule for the boundary on specificity on a fully specified view.

8.2.1.4 Technology-dependent levels (implementation level)

The implementation level(s) include all classes needed for implementing a fully specified view on a chosen platform. An initial implementation level-view typically begins with a default transformation of the fully specified level classes.

The specification language property realizations (methods) in the fully specified view act as operational specifications of the semantics for the implementation level. The classes in an implementation-level view are specified using the constructs of the implementation platform. For example, if the platform is relational, then the view is in terms of tables, columns, datatypes, referential integrity constraints, and so on. If the platform is C++, then the view is in terms of C++ names, base classes, derived classes, virtual member functions, static members, operator overloading, and so on. If the platform is Smalltalk, then the view is in terms of Smalltalk names, superclasses and subclasses, instance and class variables and methods, and so on. If the platform is Java, then the view is in terms of Java names, superclasses and subclasses, fields, methods, and so on.

In some cases, the default transformation (class to class, attribute to variable, operation to method, etc.) may not produce the best implementation; changes may be made for performance, availability, maintainability, or operational reasons. Classes may be added for database, communications, graphical user interface (GUI), or other support. Classes may also be partitioned across a series of distributed environments.

8.2.2 Identity-style view level syntax

8.2.2.1 Level 1 (survey level)

There is no special syntax for a survey-level view, beyond that of its constituent elements.

8.2.2.2 Level 2 (integration level)

There is no special syntax for an integration-level view, beyond that of its constituent elements.

8.2.2.3 Level 3 (fully specified level)

There is no special syntax for a fully specified level view, beyond that of its constituent elements.

8.2.2.4 Level 4 (implementation level) and beyond

The implementation level is part of a future version of this standard. Additional levels may also be identified, as needed.

8.2.3 Identity-style view-level rules

8.2.3.1 Rules for survey level, integration level, and fully specified level views

a) Table 13 summarizes the constructs appropriate to the various levels.
b) In an identity-style view, a many-to-many relationship may be used at any level.
c) An associative class should be introduced into an identity-style view if and only if the associative class instance has the responsibility to do something or to know something more than simply the identity of the participating instances.

8.3 Environment

In natural language, an environment is the surrounding things, conditions, circumstances, and influences that affect the development, decisions, and perspective of an organism or organization. Another name for environment might be “frame of reference.” In IDEF1X, an *environment* is a concept space—an area in which a concept has an agreed-to meaning and name(s). Every view is developed for a specific environment, and an environment may have any number of views. The environment controls the scope of its views as well as the names, descriptions, properties, and constraints given to its elements. The constituent elements of a view can only be understood, used, and referred to within a frame of reference. Environment provides that frame of reference.

In this way, an environment describes a scope of integration. The names and descriptions of IDEF1X defined concepts (i.e., views, classes, relationships, responsibilities, properties, constraints) can apply throughout an environment.[71] This enables environments to provide a migration path toward standard names and meanings for concepts.

[71]Classes, responsibilities, properties, and constraints may also be defined within a view (see 8.4).

Table 13—View level constructs (identity style)

	Level		
Construct	**Survey**	**Integration**	**Fully specified**
Subject domains	Yes	Yes	Yes
Subject domain responsibilities	Yes	Yes	Yes
State classes	Yes, as constituents of subject domains	Yes, typically abstract	Yes
Value classes	No	Some	Yes
Relationships	No	Yes	Yes
Generalizations	No	Yes	Yes
Responsibilities of classes	No	Yes	No
Attributes	No	Some, but not distinguished as attributes	Yes
Participant properties	No	Yes, as a reflection of relationships	Yes
Operations	No	Some, but not distinguished as operations	Yes
Constraints	Typically, no	Some	Yes
Property realizations	No	Some	Yes
Pre- and post-conditions	No	Some	Yes
Attribute mapping	No	Some	Yes
Detailed cardinalities	No	Some	Yes
Notes	Yes	Yes	Yes

8.3.1 Environment semantics

8.3.1.1 Structuring

Multiple environments may exist and may be structured in hierarchies with each environment having at most one parent environment. In a given environment, a concept has name(s) and a meaning stated for it in that environment or in an ancestor environment. If a concept is not described in an environment, its meaning in the closest ancestor environment is used. This allows local concepts to be described and named in lower, local environments. As standard names and descriptions are agreed upon, concepts may “move up” to an environment shared more broadly.

For example, an environment `E` may be within the scope of a parent environment `F`, meaning that every concept named in `F` is available in (although not necessarily relevant) in `E`. If a concept is described and named in `E`, it overrides any meaning of the same concept available in `F`. If a concept is not described in `E`, either it is not (perhaps yet) relevant in `E` or its meaning in `E` comes from `F`.

Figure 84 depicts four environments: Corporate, Engineering, Manufacturing, and Sales. Corporate has been established as the parent environment of Engineering, Manufacturing, and Sales. Therefore, classes described in the Corporate environment are available to the others. The situation shown indicates the following:

a) Since the class `customer` is described only in the Corporate environment, a common meaning is available to Engineering, Manufacturing, and Sales. There is no requirement that `customer` appear in a view in Engineering, Manufacturing, or Sales. In other words, the meaning of `customer` may not be relevant in any environment other than Corporate. This standard does not specify how concepts are added to parent environments.

b) Either Engineering and Manufacturing do not use the concept `product`, or they use the meaning of `product` available in Corporate. Sales has its own sense of `product`. This local meaning may enhance, refine, or contradict the Corporate meaning. This standard does not specify if or how differences between the concept meaning in Corporate and Sales would be resolved.

c) Engineering and Manufacturing have separate, not necessarily consistent, meanings for `part`. `Part` is not currently relevant to Sales. This standard does not specify if or how Engineering and Manufacturing come to a common meaning for `part`.

d) The meanings for `drawing`, `tool`, and `contact` are local to Engineering, Manufacturing, and Sales, respectively. This standard does not specify if or how these meanings are promoted to Corporate.

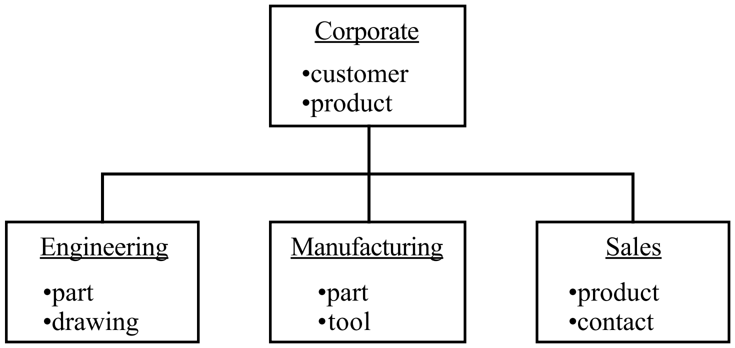


Figure 84—Environment hierarchy example

This standard does not specify the size or scope of an environment. The environments illustrated in Figure 84 are broad. While not typical, an environment can be as small as a single project or user.

8.3.2 Environment syntax

There is no graphical syntax for an environment. Figure 84 is included only to augment the explanation of environment hierarchies.

8.3.3 Environment rules

8.3.3.1 Naming

a) Each environment shall be assigned a name.

8.3.3.2 Description narrative

a) A narrative description of the environment shall be recorded.

8.3.3.3 Structuring

a) Environments may be structured as a hierarchy.
b) Multiple separate hierarchies may be established, i.e., there is no requirement that environments “converge” into a single environment at the top.
c) A defined concept in environment `E` shall override any meaning of the same concept in an ancestor environment of `E`.

8.3.3.4 Composition

a) An environment may contain views of different levels.
b) An environment shall be supported by one and only one glossary (see 8.4).

8.4 Glossary

Environments and views are supported by glossaries. A *glossary* is the collection of the names and descriptions of all terms that may be used for defined concepts (views, classes, relationships, responsibilities, properties, and constraints) within an environment or view. The principal motivation for the glossary is to ensure consistent semantics across all views in an environment[72] (see 8.3). In particular, an environment glossary enables meaningful merging and subsetting of views, which in turn enables systems integration.

The glossary describes the concepts; views make statements using those concepts. The glossary reflects the descriptions, or meanings, of the concepts; it does not contain the concepts themselves. For example, it is the meaning of `customer` that is in the glossary, not the structuring and declarations of the `customer` class itself. The allocation of properties and constraints to classes, the taxonomy of classes, and the sentences for properties and constraints are specified only within views. It is the *description* (meaning) that can be common across views.

8.4.1 Glossary semantics

8.4.1.1 Name/alias

Each concept within an environment is given a name. Some concepts may be given more than one name. When a concept has more than one name, one of its names may be designated as the “primary” name, in which case each of the other names is an *alias* name for the concept. However, if none of the multiple names is designated as “primary,” then the names are not distinguished; they are all simply “names.” Figure 85 illustrates concept names, primary names, and aliases.

• concept-1 (meaning)	**name-a**
• concept-2 (meaning)	**name-b**, name-c, name-d
• concept-3 (meaning)	name-e, name-a
• concept-4 (meaning)	name-g, name-b, **name-h**

where **name** denotes “primary”

Figure 85—Concept names and aliases

[72]In its explanation of Glossary [B13], p. 24 says “Definitions are held in a glossary common to all models within the context of the stated purpose and scope.” If the purpose is integration, the glossary needs to be common over the intended scope of integration.

As Figure 85 illustrates, the same name may be given to more than one concept within an environment. However, no name may be the primary name for more than one concept in an environment. The primary name of one concept may be a nonprimary name of another.

When a view presents a concept (e.g., in a diagram), it uses one (and only one) of the names specified for the concept in the environment glossary. Referring to Figure 85, if `concept-3` appears in a view diagram, either `name-e` or `name-a` must be used exclusively in that view, or a new name may be created and added to the environment glossary.

Additionally, there are restrictions prohibiting giving certain concepts the same name as other concepts in views (see 8.1.3). These restrictions are as follows:

a) No two classes or subject domains may use the same name, and
b) Two properties or constraints of the same class may use the same name only if they have different signatures (see also 6.3.3.1, 6.7.3.2, 7.5.3, and 8.1.3.6).

8.4.1.2 Description narrative

Each defined concept within an environment should be given a statement of its meaning. This meaning is written in narrative text.

8.4.1.3 Environment glossary

An environment glossary allows defined concepts presented in a view to have a common meaning across the environment, independent of view. These concepts are then available to be used in views. In this way a state class such as `employee` may appear in multiple views, with a somewhat different set of properties and relationships in each, and yet still reflect the same concept. The intent is that `employee` be the class of all employees, i.e., individual persons who are classified as belonging to the class `employee` on the basis of some common criteria. It is that sense of what it *means* to be an employee that is described in the glossary. Similarly, a concept such as `birthDate` may be described once within the environment and used as a value class in appropriate views. It is that sense of what a birth date *means* globally that is described in the glossary.

Some defined concepts within an environment express the meaning of a combination of other defined concepts. For example, the concept of `employee` might be stated as one glossary entry and the concept of `birthDate` might be stated as another defined concept. A third defined concept might explain the sense of what an “`employee birthDate`” is if that concept has a more specific meaning than the terms individually. When `birthDate` is used as an attribute of `employee`, with or without the assignment of a role name, the sense of what it is to be an “employee birthdate” is provided by this compound defined concept. In other words, this “in context” meaning includes the sense of how a `birthDate` describes an `employee`.

8.4.1.4 Model glossary

Since a model may span environments, the scope and content of its glossary is determined by the views contained in the model. A *model glossary* is the collection of the names and descriptions of all defined concepts that appear within the views of that model.

8.4.2 Glossary syntax

There is no graphical syntax for a glossary.

8.4.3 Glossary rules

8.4.3.1 Composition

a) An environment glossary shall contain the descriptions and names entered locally as well as those in all environments higher in the hierarchy of environments.
b) A concept described and named in a local environment shall override, for that environment, any description and name(s) for the concept from an environment higher in the hierarchy.
c) A model glossary shall collect all concepts described in each view the model contains.

8.4.3.2 Name

a) Every defined concept shall be named.
b) A name, whether primary, alias, or undesignated, shall conform to the lexical rules for naming stated in 4.2.
c) No name may be the primary name for more than one concept in an environment.
d) The environment glossary may name (and define) concepts not used in any view within the environment.

8.4.3.3 Description narrative

a) The environment glossary shall contain a narrative description for each class within the environment.
b) The description of a class should allow a user of the class to determine whether a thing qualifies as an instance of the class.
c) The environment glossary shall directly contain a narrative description of the environment itself.
d) The environment glossary may contain a narrative description for each view, relationship, responsibility, property, and constraint within the environment.
e) The same meaning may not apply to two distinct defined concepts within an environment.
f) The description of a defined concept may reference the description of (one or more) other defined concepts, e.g., to avoid repeating sections of text already stated in the referenced description.
g) For each defined concept, the glossary shall permit the specification of additional optional information such as author name, creation date, and last modification date.
h) A view should have a narrative description. This description may contain statements about the relationships in the view, brief descriptions of classes and properties, discussions of rules or constraints that are specified, and any other information useful to the user of the view.
i) For each view, the environment glossary shall permit the specification of additional optional information such as completion or review status and view type (e.g., function-based, subject-based).
j) All the elements of a view description shall be displayable on the view diagram.
k) The model glossary shall directly contain a narrative description of the model itself.

8.5 Model

A *model* is a packaging of one or more views along with narrative descriptions of the views and view components (e.g., classes, properties) called out in the views. The components of an IDEF1X model are the constructs (e.g., classes, properties) described in Clauses 5 and 6. These constructs are first organized into views. Views may then be organized into models (see 8.1).

IDEF1X models are a representation of something. Like many other kinds of models (e.g., model cars, models of the solar system), IDEF1X models suppress certain aspects of the modeled subject. This suppression is done in order to make the model easier to deal with, to make it more economical to manipulate, and to focus attention on aspects of the modeled subject that are important for the intended purpose of the model. For instance, an accurate model of the solar system could be used to predict when planetary conjunctions will take place and the phases of the moon at a particular time. Such a model would generally not attempt to represent the internal workings of the sun or the surface composition of each planet.

The success of a model very much depends upon a common understanding of just which aspects of the modeled subject are suppressed and which ones are attended to in the given model. Accordingly, each IDEF1X model should be accompanied by a statement of purpose (describing why the model was produced) and a statement of scope (describing the general area covered by the model).

8.5.1 Model semantics

8.5.1.1 Types

Models may be formed for a variety of reasons. There are no explicit model types. Any collection of views gathered together for a stated purpose may be called a model.

8.5.2 Model syntax

There is no graphical syntax for a model.

8.5.3 Model rules

All the following rules apply only to a <u>completed</u> model.

8.5.3.1 Naming

a) A model shall be assigned a name.

8.5.3.2 Description narrative

a) A narrative description of the model shall be recorded.
b) The purpose of the model shall be recorded.
c) The scope of the model shall be recorded.

8.5.3.3 Composition

a) A model shall contain one or more views.
b) A model shall contain a glossary.
c) A model may contain views of different levels.
d) A model may contain views from different environments.

9. Key-style modeling

The identity style of IDEF1X modeling introduced in this standard supports, but is not limited to, the construction of object models. Object techniques are gaining an ever-increasing community of users who must understand not only the data (information) aspect of a system but also its behavior. The features of the identity-style language support this audience and will be extended in future versions of this standard to provide additional capabilities.

Data modeling, however, continues to provide an effective technique for developing integrated databases and will continue to be used for what is expected to be a considerable period of time. Many organizations have invested heavily in constructing models based on earlier versions of IDEF1X and are expected to continue to do so. Both data modelers and object modelers must be supported.

As described in 1.3, the needs of IDEF1X users continue to evolve. While many users will adopt the object techniques early, others will, for various reasons, continue to develop data models. With minor exceptions, a model conformant with earlier versions of IDEF1X will remain conformant under this version.[73]

This clause describes how to apply the concepts presented in the previous clauses to produce a key-style view. The *key style* of IDEF1X modeling is backward-compatible with the US government's federal standard for IDEF1X, FIPS PUB 184 [B13]. Key-style models may continue to be used to represent the structure and semantics of data within an enterprise, i.e., for data (information) models.[74] With the extension of domain into value class, key-style models can also be used to support the development of extended-relational implementations.

A key-style model is a highly restricted identity-style model. The "new concepts" described in Figure 2 are not used in a key-style model, and the features indicated as "unnecessary for the object model" (primary keys, foreign keys, and identifying relationships) are retained. The result is a language that is almost identical to that described in the current federal standard, FIPS PUB 184 [B13]. To assist in mapping the federal standard to this IEEE standard, the topics in this clause are presented in a structure similar to that of Section 3 (Syntax And Semantics) of FIPS PUB 184 [B13]. The following topics are discussed in this clause in support of key-style modeling. For a condensed comparison of the identity style and key-style constructs, see also Annex B.

a) **Entity:** An entity roughly corresponds to a severely limited state class.
b) **Domain/Value Class:** The domain in FIPS PUB 184 [B13] is replaced by value class.
c) **View:** A view roughly corresponds to an identity-style view but is limited to key-style constructs.
d) **Attribute:** An attribute roughly corresponds to a limited identity-style attribute.
e) **Relationship:** A "one-to-many relationship" is represented by a migrated foreign key rather than a participant property. A "many-to-many relationship" (nonspecific relationship) is a severely limited version of its identity-style counterpart.
f) **Generalization:** Generalization has the same meaning as in identity-style modeling, but is represented by foreign keys.
g) **Primary and Alternate Key:** A key represents a uniqueness constraint. One key (the primary) represents the identity of each entity instance. Keys are not specified in identity-style modeling, which employ intrinsic instance identity instead.
h) **Foreign Key:** A foreign key is used to represent a relationship; foreign keys are not specified in identity-style modeling.

[73]The exceptions are as follows: Domain is replaced by value class. The integer number that could follow the entity name in [B13] has been dropped from this standard. The allowance for "Author Conventions" and the "For Exposition Only" view annotation in [B13] has been dropped from this standard. The "range" cardinality annotation in [B13] has been dropped from this standard (a note can be used to record the range). The second method of naming the relationship from the child perspective, i.e., using the direct object of the phrase in place of the entire verb phrase as in [B13], has been dropped from this standard.

[74]Note that none of the syntax or rules in this clause applies to the model constructs of an identity-style view, unless specifically stated.

i) **View Level:** The view levels roughly correspond to the identity-style view levels.

j) **Glossary:** Glossary objectives are similar in key-style and identity-style modeling.

k) **Note:** Notes are used extensively in key-style modeling as there is no other way to specify many relevant constraints.

Also discussed in this clause are key-style lexical rules. Finally, Annex B provides some comments on migration considerations for those users moving from data modeling to object modeling.

The formalization for IDEF1X in Clause 10 of this standard covers only the identity style (the full version of the language). The key style is not formalized in this standard. Doing so would yield a formalization similar to that in Annex B of FIPS PUB 184 [B13].

9.1 Entity

Entities in key-style modeling represent the things of interest in data modeling, that is, things about which some information is relevant. A key-style entity is equivalent to a state class that has only attribute properties, that has no class-level properties, and whose instances are uniquely identified by one or more attribute values (i.e., the instances have none of the participant properties, operations, or constraints available in identity-style views).

The notion of classification in key style and identity-style modeling differs in one significant way. In the key style, there is a restriction on classification that forbids any two instances of a class from agreeing on all attribute values. The restriction is due to the fact that, so far as key-style modeling is concerned, there is nothing else to know about an entity.[75]

9.1.1 Entity semantics

9.1.1.1 Class/instance

A key-style *entity class* (simply, *entity*) is a representation of a set of real or abstract things (people, objects, places, events, ideas, combinations of things, etc.) that have common attributes or characteristics. An *entity instance* (simply, *instance*) is an individual member of the set. A real world object or thing may be represented by more than one entity within a view. For example, John Doe may be an instance of both the entities `employee` and `buyer`. Furthermore, an entity instance may represent a concept involving a combination of real world objects. For example, `John` and `Mary` could be the participants in an instance of the entity `married-Couple`.

9.1.1.2 Independent/dependent

An entity is *identifier-independent* (simply, *independent*) if each instance of the entity can be uniquely identified without determining its relationship to another entity. An entity is *identifier-dependent* (simply, *dependent*) if the unique identification of an instance of the entity depends upon its relationship to another entity. Expressed in terms of the foreign key, an entity is said to be dependent if any foreign key is wholly contained in its primary key (see 9.8). Otherwise, it is independent.

[75]Because identity-style modeling includes the concept of identity, there is no need for this restriction. If such a restriction were to apply to a class of objects, a uniqueness constraint to that effect could be declared. If this were done for every class, the result would be key-style classification. The key-style restriction that an entity class have at least one attribute is a consequence of the distinguishability restriction and is similarly not needed in the identity-style model.

9.1.1.3 Naming

The entity name is a noun phrase that describes a thing in the set that the entity represents. The noun phrase is in singular form, not plural. Abbreviations and acronyms are permitted; however, the entity name must be meaningful and consistent throughout the view.

9.1.2 Entity syntax

9.1.2.1 Graphic

An entity shall be represented by a rectangle:

a) A *dependent* entity shall be represented by a rectangle with rounded corners (see Figure 86).
b) An *independent* entity shall be represented by a rectangle with square corners (see Figure 86).

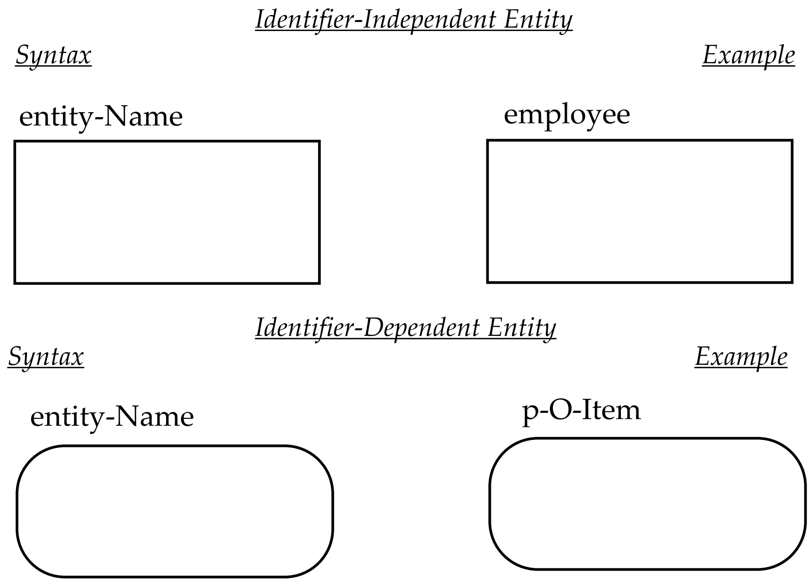


Figure 86—Key-style entity syntax

9.1.2.2 Label

a) Each entity shall be assigned a label.[76]
b) The label shall be placed either:
    1) Above or inside the rectangle when no attribute names are shown (see Figure 86), or
    2) Above the rectangle when attribute names are shown in the rectangle (see Figure 87).
c) In a view, an entity shall be labeled by either its entity name or one of its aliases.
d) An entity may be labeled by different names (i.e., aliases) in different views in the same model (see 9.11).

9.1.3 Entity rules

9.1.3.1 naming

a) Within a view, each entity shall have a unique name.
b) Within a view, the same meaning shall always apply to the same entity name.

[76]The integer number that could follow the entity name in [B13] has been dropped from this standard.

c) Within a view, the same meaning shall not apply to different entity names unless the names are aliases.

d) Within a view, no entity may have the same name as an attribute.

e) No view shall contain two distinctly named entities in which the names are synonymous. Two names are synonymous if either is directly or indirectly an alias for the other, or there is a third name for which both names are aliases (either directly or indirectly).

9.1.3.2 Attributes

a) In a completed key-based or fully attributed view, an entity shall have one or more attributes whose values uniquely identify every instance of the entity (see 9.7).

b) In a completed key-based or fully attributed view, an entity shall have one or more attributes that are either owned by the entity or migrated to the entity through a relationship (see 9.8).

9.1.3.3 Description narrative

a) A narrative description of the entity, along with a list of synonyms or aliases (if any), shall be stated in the glossary (see 9.11).

9.1.3.4 Relationships and foreign keys

a) An entity may have any number of relationships with other entities in the view.

b) If an entire foreign key is used for all or part of an entity's primary key, then the entity shall be identifier-dependent (see 9.8).

c) Conversely, if only a portion of a foreign key or no foreign key attribute at all is used for an entity's primary key, then the entity shall be identifier-independent (see 9.8).

9.2 Domain/value class

What was called domain in FIPS PUB 184 [B13] is subsumed by value class. Value classes as described in 5.3 and 6.4 may be used with key-style modeling. A domain in FIPS PUB 184 [B13] is a value class that

a) Is not a collection class,

b) Has no operations,

c) Has no constraints other than value list or value range constraints,

d) Has no class-level properties,

e) Has exactly one generic parent,

f) Has attributes that are all public and nonderived, and

g) Is never mapped to by an attribute of a different name.

9.3 Key-style view

A key-style *view* is a collection of entities and assigned value classes (attributes) assembled for some purpose. A view may cover the entire area being modeled or only a part of that area. A key-style *model* comprises one or more views (often presented in view diagrams representing the underlying semantics of the views), along with descriptions of the entities and value classes (attributes) used in the views.

Views serve the same purpose in key-style and identity-style modeling, that is, to collect together those concepts that must be considered together to make sense of an area being studied. Views allow the recombination of those concepts for reasoning about related areas.

9.3.1 Key-style view semantics

In key-style modeling, entities and value classes are described in a common glossary and mapped to one another in views. In this way an entity such as `employee` may appear in multiple views in multiple models and have a somewhat different set of attributes in each. In each view, it is required that the entity `employee` means the same thing. The intent is that `employee` be the class of all employees, that is, individual things are classified as belonging to the class `employee` on the basis of some similarity. It is that sense of what it means to be an employee that is stated in the glossary. Similarly, the value class `employee-Name` is described once and used as an attribute in appropriate views.

9.3.2 Key-style view syntax

9.3.2.1 Composition

a) The constructs depicted in a key-style view diagram, i.e., entities, attributes, relationships, and notes, shall comply with all syntax and rules governing the individual key-style constructs.

9.3.2.2 Presentation

a) The syntactical definitions of the IDEF1X language characterize the full set of IDEF1X constructs and their use.
b) This does not however, preclude *hiding* (i.e., optionally omitting the display of) certain constructs in order to allow an alternate presentation of a view. Many times this hiding is done to suppress detail not needed for a certain discussion, or to abstract a view to permit a broader view. An example of an alternate presentation might be the presentation of a fully attributed view showing only the entities and their relationships to allow the view to be reviewed from an entity-relationship perspective.
Examples of some of the possible constructs that may be hidden include
    1) Attributes
    2) Primary keys designations
    3) Foreign keys
    4) Role names
    5) Relationship names
    6) Cardinality specifications
    7) Category discriminators
    8) Alternate key designations
    9) Note identifiers
c) When constructs are hidden, all applicable syntactical and semantic rules shall still be enforced.

9.3.3 Key-style view rules

9.3.3.1 Naming

a) Each view shall have a unique name.
b) The view name shall appear on any presentation of the view, e.g., the view diagram.

9.3.3.2 Composition

a) Although an entity may be included in any number of views, it may appear only once within a given view.

9.3.3.3 Optional information

a) A view may have additional descriptive information including, for example, the name of the author, dates created and last revised, level (e.g., entity-relationship, key-based, fully attributed), completion or review status, and so on. A view should be accompanied by a statement of purpose and scope, as well as a brief description of the area covered.

b) A textual description of the view may also be provided. This description may contain narrative statements about the relationships in the view, brief descriptions of entities and attributes, and discussions of rules or constraints that are specified.

9.3.3.4 Levels

a) A model may contain views of different levels.[77]

9.4 Attribute

An *attribute* is a mapping from the instances of an entity to the instances of the value class for that attribute. This value class is also referred to as the *type* of the attribute. In key-style modeling, an attribute is associated with a specific entity.

Key-style view attributes are similar to identity-style view attributes in that they represent mappings, are single-valued (functions), and they may be declared either total or partial. However, in a key-style model, the additional attribute classifications available within identity-style modeling (e.g., class-level, constant, intrinsic) are not available.[78] Yet, overall, the essential notion of attribute is the same in key-style and identity-style modeling.

9.4.1 Attribute semantics

9.4.1.1 Attribute/attribute instance

In an IDEF1X key-style view, an *attribute type* (simply *attribute*) represents a type of property (characteristic) associated with an entity. An *attribute instance* is a specific property of an individual instance of the entity. An attribute instance is specified by both the type of property and its value, referred to as an *attribute value*. An instance of an entity, then, will usually have a single specific value for each associated attribute at a point in time. For example, `employee-Name` and `birth-Date` may be attributes associated with the entity `employee`. An instance of the entity `employee` could have the attribute values of “`Jenny Lynne`” and “`February 27, 1953`”, respectively.

9.4.1.2 Naming

Each attribute is identified by the unique name of its underlying value class. The name is expressed as a noun phrase that describes the characteristic represented by the attribute. The noun phrase is in singular form, not plural. Abbreviations and acronyms are permitted; however, the attribute name must be meaningful and consistent throughout the model. A narrative description, along with a list of synonyms or aliases (if any), must be recorded in the glossary.

In a key-style model, an attribute name is restricted to be the same as its value class name. This restriction is done to promote integration across views. A value class is described within an environment and applies to all views in that environment. Naming an attribute for its value class has the effect of achieving a common meaning for every attribute within an environment and, therefore, allows views to be cleanly merged.[79]

[77]The allowance for “Author Conventions” and the “`For Exposition Only`” view annotation in [B13] has been dropped from this standard.

[78]It can therefore be said that a key-style attribute is a property that retains only the ability to know, not to do.

9.4.1.3 Mapping completeness

In key-style models, an attribute may be designated as *optional*, i.e., the mapping is partial (some instances have no value for the attribute). An optional attribute has at most one value for an entity instance.

An attribute not designated as optional is, by default, *total* (mandatory). A mandatory attribute has exactly one value for each instance.

9.4.1.4 Attribute ownership

An attribute of an entity that is not a foreign key in that entity is said to be "owned" by that entity. Key-style modeling imposes a *single ownership rule*. This rule says that every attribute in a view must be owned by exactly one entity in the view.[80]

9.4.1.5 Attribute migration

In addition to an attribute being "owned" by an entity, an attribute may be present in an entity due to its "migration" through a relationship or through a generalization structure (see 9.8). For example, if every employee is assigned to a department, then the attribute `department-Number` could be an attribute of `employee` that has migrated through the relationship to the entity `employee` from the entity `department`. The entity `department` would be the owner of the attribute `department-Number`. Only primary key attributes may be migrated through a relationship. The attribute `department-Name`, for example, would not be a migrated attribute of `employee` if it was not part of the primary key for the entity `department`.

9.4.1.6 Attribute inheritance

In an entity-relationship (ER), key-based (KB) or fully attributed (FA) level key-style view, every attribute is owned by only one entity, but an attribute may be applicable to an entity via *inheritance*. The attribute `monthly-Salary`, for example, might apply to some instances of the entity `employee` but not all. Therefore, a separate but related category entity called `salaried-Employee` might be identified in order to establish ownership for the attribute `monthly-Salary` and to distinguish between employees in general and those who earn salaries. Since an actual employee who was salaried would represent an instance of both the `employee` and the `salaried-Employee` entities, attributes common to all employees (such as `employee-Name` and `birth-Date`) are owned attributes of the `employee` entity rather than just the `salaried-Employee` entity.

[79]The cost of naming an attribute for its value class is a proliferation of value classes (e.g., `hotel-Fahrenheit-Temperature`, `room-Fahrenheit-Temperature`), perhaps generalized as a common value class (e.g., `fahrenheit-Temperature`). This approach causes no problem in key-style models because, other than providing a set of "types" for attributes, there are few other demands on the value class generalization hierarchy.

Although the same rule could be adopted in an identity-style model, there would be problems. One is that, given operations on abstract value classes (e.g., `temperature`), there are many more demands on the value class generalization hierarchy; the proliferation of subclasses occasioned by the key-style naming requirement would be a distracting burden. Another problem is a conflict with polymorphism. In an identity-style model, it is not uncommon to have an attribute (more generally, a property) of a given name overridden by a property of the same name in a subclass. Often, both have the same value class, but in some cases they do not. If the attribute name was required to be the same as the value class name in an identity-style mode, there would be no way to name the attribute to provide for polymorphism.

[80]The single ownership rule originated in IDEF1, which did not have generalization. The intent of the rule is to promote normalized models with consistent business rules. The implicit assumption is "same name, same meaning." The more subtle assumption is that attributes of the same name in distinct classes may not have the same realization. In fact, this difference may be precisely the objective in an identity-style model, where the overriding of attributes in a subclass is used to specify attributes of the same name and same meaning but having differing realizations.

The single-ownership rule is in conflict with polymorphism for reasons very similar to those discussed above under naming. The single-ownership rule is, therefore, not appropriate in identity-style modeling. If such a rule were desired for some reason, there is nothing in key-style modeling to prevent it. It would be regarded as legal but unwise.

Such attributes are said to be “inherited” by the category entity. They are not included in the category entity’s list of attributes: they only appear in the list of attributes of the generic entity.

9.4.1.7 Primary key

In key-style modeling, an entity must have an attribute (or combination of attributes) whose value uniquely identifies every instance of the entity. This attribute or attributes form the *primary key* of the entity (see 9.7). For example, the attribute `employee-Number` might serve as the primary key for the entity `employee` while the attributes `employee-Name` and `birth-Date` might be nonkey attributes.

9.4.1.8 Derived attribute

In key-style modeling, there is no prohibition against derived attributes, but many modelers impose such a restriction in order to rule out an endless pursuit of derivable results.[81] One of the tenets of data modeling is that a stable, base set of data will allow the derivation of an infinite amount of information. The attempt to identify all the potential derivations on a data model can become a never-ending task.

9.4.2 Attribute syntax

9.4.2.1 Graphic

a) Attributes shall be shown by listing their names inside the associated entity box.
b) Attributes that specify the primary key shall be placed at the top of the list and separated from the other attributes by a horizontal line (see Figure 87).
c) An attribute that is not part of a primary key may have no value.
d) An attribute that may have no value is marked by the symbol “`(O)`” (an upper case O, for “optional”) following the attribute name.

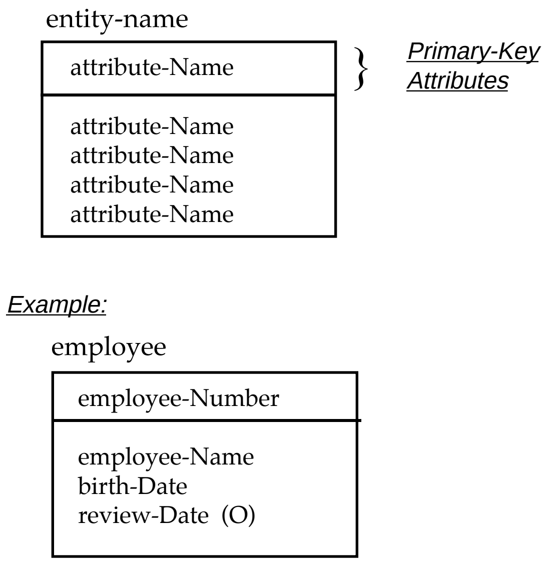


Figure 87—Attribute and primary key syntax

[81]Identity-style models, on the other hand, directly support derived attributes with somewhat less of a cultural bias against them. Derivation rules must be specified somewhere and, since all computation is done by objects (instances), a referenced attribute has to be derived by some object. In practice, the avoidance of an endless pursuit of derived results in identity-style modeling comes from a distinction between common enterprise objects, application-specific objects, and presentation objects.

9.4.3 Attribute rules

9.4.3.1 Naming

a) Each attribute shall be a value class used in an entity in a view.

b) Each attribute shall have a unique name, and the same meaning shall always apply to the same name. Furthermore, the same meaning may not apply to different names unless the names are aliases of each other.

c) In a view, an attribute shall be labeled by either its attribute name or one of its aliases.

d) In a view, if an attribute is an owned attribute in one entity and a migrated attribute in another entity, either

   1) It shall have the same name in both or

   2) It shall have a role name (or an alias for a role name) as the migrated attribute.

e) An attribute may be labeled by different names (i.e., aliases) in different views within the same model.

f) No view may contain two distinctly named attributes in which the names are synonymous. Two names are synonymous if either is directly or indirectly an alias for the other, or there is a third name for which both names are aliases (either directly or indirectly).

9.4.3.2 Attribute ownership

a) An entity may own any number of attributes.

b) In an entity-relationship, key-based or fully attributed view, every attribute shall be owned by exactly one entity.

9.4.3.3 Attribute migration

a) An entity may have any number of migrated attributes.

b) A migrated attribute shall be part of the primary key of a related parent entity or a generic entity.

9.4.3.4 Primary key

a) Every instance of an entity shall have a value for every attribute that is part of its primary key.

9.4.3.5 Function/multi-valued

a) No instance of an entity may have more than one value for an attribute associated with the entity.

9.5 Relationship

In earlier versions of IDEF1X,[82] specifications based on foreign keys were used to capture the underlying sense of dependency between entities. In key-style modeling, entities and relationships are classified based on the role that the foreign keys play in the entity. While such distinctions are not appropriate in a modeling style that distinguishes instances using identity rather than primary keys, the discussion here applies when key-style modeling is used. In an IDEF1X key-style view, relationships are used to represent associations between entities.[83]

[82]These earlier versions included [B3], [B13], and[B15].

[83]It can, therefore, be said that a key-style relationship is simply a relationship in which the identity of the parent participant is represented by a primary key of the parent held in the child.

9.5.1 Relationship semantics

9.5.1.1 Specific/nonspecific

A *specific relationship* (simply, *relationship*) is an association (connection) between two entities in which each instance of one entity (referred to as the parent entity) is associated with zero, one, or more instances of the second entity (referred to as the child entity). Furthermore, each instance of the child entity is associated with zero or one instance of the parent entity. For example, a specific relationship would exist between the entities `buyer` and `purchase-Order` if a buyer issues zero, one, or more purchase orders and each purchase order must be issued by a single buyer.

Such parent-child relationships are considered to be "specific" relationships because they specify precisely how instances of one entity relate to instances of another entity. By contrast, a *nonspecific relationship* may be used to represent a "many-to-many" association between entities. A nonspecific relationship (*many-to-many relationship*) is an association between two entities in which each instance of the first entity is associated with zero, one, or many instances of the second entity, and each instance of the second entity is associated with zero, one, or many instances of the first entity. For example, if an employee can be assigned to many projects and a project can have many employees assigned, then the connection between the entities `employee` and `project` can be expressed as a nonspecific relationship.

In the initial development of a model, it is often helpful to identify nonspecific relationships between entities. This nonspecific relationship may be replaced with specific relationships later in the model development by introducing a third entity, such as `project-Assignment`, which is a common child entity in specific relationships with the `employee` and the `project` entities. The new relationships would specify that an employee has zero, one, or more project assignments. Each project assignment is for exactly one employee and exactly one project. An entity introduced to resolve a nonspecific relationship is sometimes called an *intersection entity* (*associative entity*).

Nonspecific relationships may only remain in completed entity-relationship level key-style views. In a key-based or fully attributed level view, all associations between entities must be expressed as specific (parent/child) relationships.[84] In the remainder of this clause, use of the simple term "relationship" will denote "specific relationship" unless otherwise qualified.

9.5.1.2 Relationship type/instance

A key-style view diagram depicts the *type* (set) of relationships between two entities. A specific *instance* of the relationship associates specific occurrences of the entities. For example, "buyer John Doe issued Purchase Order number 123" is an instance of a relationship.

9.5.1.3 Relationship classification

A relationship is designated as *identifying* if the foreign key attributes it contributes are wholly contained in the primary key of the child entity. Otherwise, the relationship is designated as *nonidentifying*.

9.5.1.4 Cardinality

A relationship may specify its *cardinality*, i.e., how many child entity instances may exist for each parent instance. The following relationship cardinalities may be expressed from the perspective of the parent entity:[85]

a) Each parent entity instance must have at least one associated child entity instance.

[84]Many-to-many relationships may remain in corresponding levels of identity-style views.

[85]The "range" cardinality annotation in [B13] has been dropped from this standard. A note may be used to record the range.

b) Each parent entity instance may have zero or, at most, one associated child instance.
c) Each parent entity instance is associated with some exact number of child entity instances.
d) Each parent entity instance may have zero or more associated child entity instances (If no cardinality is explicitly stated from the perspective of the parent entity, this is the default.)

Cardinality may also be described from the perspective of the child entity, as will be discussed below.

A nonspecific relationship may specify the cardinality from both directions of the relationship.

9.5.1.5 Identifying relationship

If an instance of the child entity is identified by its association with the parent entity, then the relationship is referred to as an *identifying relationship*, and each instance of the child entity must be associated with <u>exactly one</u> instance of the parent entity. For example, if one or more tasks are associated with each project and tasks are only uniquely identified within a project, then an identifying relationship would exist between the entities, `project` and `task`. In other words,, the associated project must be known in order to identify one task uniquely from all other tasks (see 9.8). The child in an identifying relationship is always *existence-dependent* on the parent, i.e., an instance of the child entity may exist only if it is related to an instance of the parent entity. An identifying relationship is always mandatory from the perspective of the child instance.

9.5.1.6 Nonidentifying relationship

If every instance of the child entity can be uniquely identified without knowing the associated instance of the parent entity, then the relationship is referred to as a *nonidentifying relationship*. For example, although an existence-dependency relationship may exist between the entities `buyer` and `purchase-Order`, purchase orders may be uniquely identified by a purchase order number without identifying the associated buyer.

9.5.2 Relationship syntax

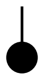
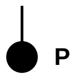
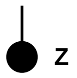
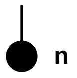
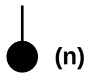
9.5.2.1 Graphic

a) A relationship shall be depicted as a line drawn between the parent entity and the child entity with a solid dot at the child end of the line.
b) The unconstrained, default cardinality from the perspective of the parent entity shall be zero, one, or many.
c) A “P” (for positive) shall be placed beside the dot to indicate a cardinality of one or more (see Table 14).
d) A “Z” shall be placed beside the dot to indicate a cardinality of zero or one (see Table 14).
e) If the cardinality is an exact number, a positive integer number shall be placed beside the dot (see Table 14).
f) Other cardinalities (e.g., “more than 3,” “exactly 7 or 9,” or ranges), may be recorded as notes that are placed beside the dot (see Table 14).
g) A nonspecific relationship shall be depicted as a line drawn between the two associated entities with a solid dot at each end of the line.
h) The cardinality of a nonspecific relationship may be expressed at both ends of the relationship using the notation shown in Table 14.

9.5.2.2 Identifying relationship

a) A solid line shall depict an identifying relationship between the parent and child entities (see Figure 88).
b) If an identifying relationship exists,

Table 14—Parent perspective cardinality syntax

Cardinality	Graphic	Cardinality expression
A solid dot indicates a lack of any cardinality constraint, i.e., zero, one, or more.		zero or more
A “P” beside a solid dot indicates one or more (at least one, and possibly more).		one or more
A “Z” beside a solid dot indicates zero or one (at most one).		zero or one
A positive integer beside the dot indicates a cardinality of an exact number.		exactly *n*
A note number (an integer enclosed in parentheses) indicates a cardinality specified in the body of the note.		reference to note `(n)` where cardinality is specified.

1) The child entity shall always be an identifier-dependent entity (represented by a rounded rectangle) and
2) The primary key attributes of the parent entity shall also be migrated primary key attributes of the child entity (see 9.8).

c) The parent entity in an identifying relationship shall be identifier-independent unless the parent entity is also the child entity in some other identifying relationship, in which case both the parent and child entities shall be identifier-dependent.

d) An entity may have one or more relationships with other entities. However, if the entity is a child entity in any identifying relationship, it shall always be shown as an identifier-dependent entity (represented by a rounded rectangle), regardless of its role in the other relationships.

9.5.2.3 Nonidentifying relationship

a) A dashed line shall depict a nonidentifying relationship between the parent and child entities.

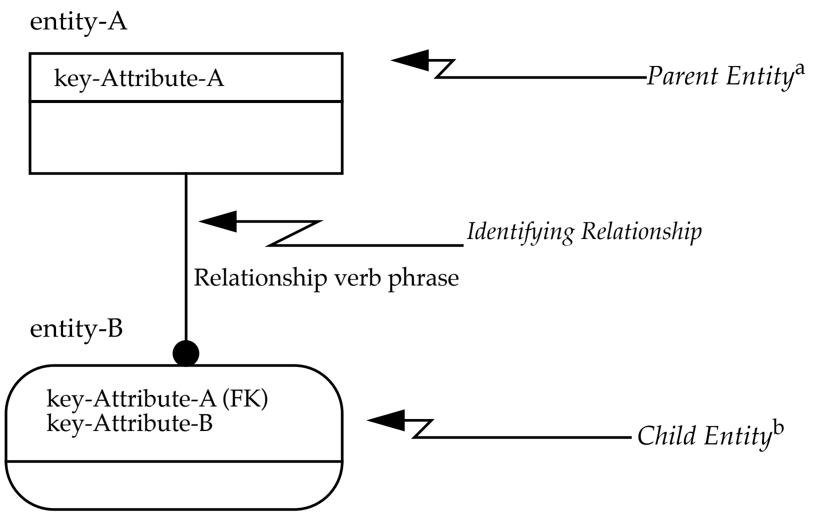
b) Both parent and child entities shall be identifier-independent entities in a nonidentifying relationship unless either or both are child entities in some other relationship that is an identifying relationship.

c) A nonidentifying relationship shall be designated as either mandatory or optional from the perspective of the child instance.

9.5.2.4 Mandatory nonidentifying relationship

a) In a mandatory (total) nonidentifying relationship, each instance of the child entity shall be related to exactly one instance of the parent entity (see Figure 89).

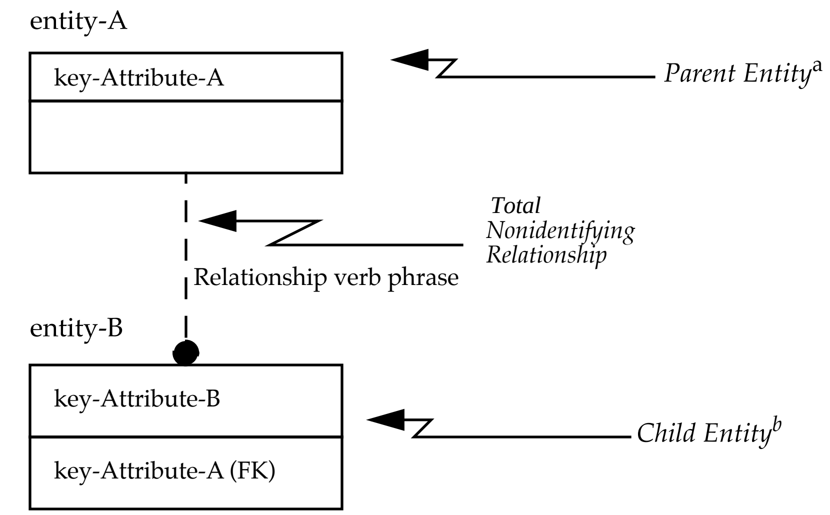
b) A mandatory nonidentifying relationship shall have no annotation on the relationship line adjacent to the parent entity rectangle (see Figure 89).



[a]The parent entity in an identifying relationship may be an identifier-independent entity (as shown) or an identifier-dependent entity depending upon other relationships.

[b]The child entity in an identifying relationship is always an identifier-dependent entity.

Figure 88—Identifying relationship syntax



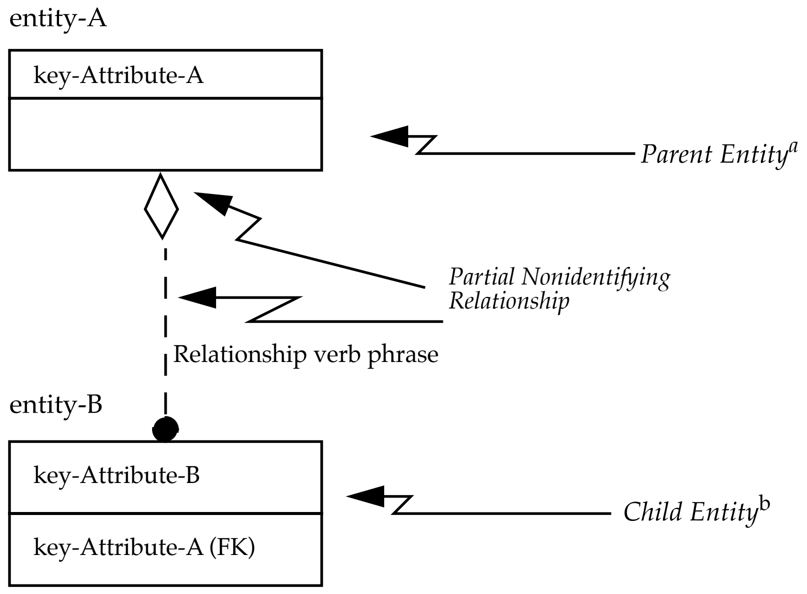
[a]The parent entity in a mandatory (total) nonidentifying relationship may be an identifier-independent entity (as shown) or an identifier-dependent entity if it is a child entity in some identifying relationship.

[b]The child entity in a mandatory (total) nonidentifying relationship will be an identifier-independent entity (as shown) unless it is also a child entity in some identifying relationship.

Figure 89—Mandatory (total) nonidentifying relationship syntax

9.5.2.5 Optional nonidentifying relationship

a) A dashed line with a small diamond at the parent end shall depict an *optional nonidentifying relationship* between the parent and child entities (see Figure 90).



[a]The parent entity in an optional (partial) nonidentifying relationship may be an identifier-independent entity (as shown) or an identifier-dependent entity if it is a child entity in some identifying relationship.

[b]The child entity in an optional (partial) nonidentifying relationship will be an identifier-independent entity (as shown) unless it is also a child entity in some identifying relationship.

Figure 90—Optional (partial) nonidentifying relationship syntax

b) In an optional nonidentifying relationship, each instance of the child entity shall be related to zero or one instances of the parent entity.

c) An optional nonidentifying relationship shall represent a conditional existence dependency. An instance of the child in which each foreign key attribute for the relationship has a value shall have an associated parent instance in which the primary key attributes of the parent are equal in value to the foreign key attributes of the child.

d) An optional nonidentifying relationship shall have a small diamond on the relationship line adjacent to the parent entity rectangle (see Figure 90).

9.5.2.6 Labeling

a) A relationship shall be given a name, expressed as a verb or verb phrase that is placed with the relationship line.

b) The name of each relationship between the same two entities shall be unique, but a relationship name need not be unique across the model.

c) The name for a relationship should usually be expressed in the parent-to-child direction such that a sentence can be formed by combining the parent entity name, the relationship verb phrase, a cardinality expression, and the child entity name.

For example, the statement “A project funds one or more tasks” could be derived from a relationship showing `project` as the parent entity, `task` as the child entity with a “`P`” cardinality symbol, and “`funds`” as the relationship verb phrase.

d) When a relationship is named from both the parent and child perspectives, the parent perspective shall be stated first, followed by the symbol "/" and then the child perspective.

e) The relationship shall be required to hold true when stated from the reverse direction, even if the child-to-parent relationship is not named explicitly.

   From the previous example, it is inferred that "a task is funded by exactly one project." The child perspective here is represented as "`is funded by`." The full relationship label for this example, including both parent and child perspectives, would be "`funds/is funded by`."

f) The parent perspective relationship verb phrase should be stated for all relationships.[86]

g) A nonspecific relationship is typically named in both directions (see Figure 91). For a nonspecific relationship, the relationship label shall be expressed as a pair of verb phrases placed beside the relationship line and separated by a slash ("`/`").

h) Since a nonspecific relationship has no notion of "parent" or "child" roles, the order of the verb phrases shall depend on the relative position of the entities, as follows:
   1) The first shall express the relationship from either the left entity to the right entity (if the entities are arranged horizontally) or the top entity to the bottom entity (if they are arranged vertically).
   2) The second portion of the relationship label shall express the relationship from the other direction, that is, either the right entity to the left entity or the bottom entity to the top entity, again depending on the orientation.
   3) Top-to-bottom orientation shall take precedence over left-to-right, so if the entities are arranged upper right and lower left, the first verb phrase describes the relationship from the perspective of the top entity.

i) For a nonspecific relationship, the relationship shall be labeled so that sentences can be formed by combining the entity names with the phrases.

   For example, the statements "A project has zero, one, or more employees" and "An employee is assigned zero, one, or more projects" can be derived from a nonspecific relationship labeled "`has/is assigned`" between the entities `project` and `employee`. (The sequence assumes the entity `project` appears above or to the left of the entity `employee`.)

9.5.3 Relationship rules

9.5.3.1 Composition

a) A relationship is always between exactly two entities.
b) These two related entities need not be distinct.
c) An entity may be associated with any number of other entities, as either a child or a parent.
d) An instance of a parent entity may be associated with zero, one, or more instances of the child entity, depending on the specified cardinality.
e) In a nonspecific relationship, an instance of either entity may be associated with zero, one, or more instances of the other entity, depending on the specified cardinality of each.

9.5.3.2 Identifying relationship/nonidentifying relationship

a) A relationship may be classified as one of the following:
   1) An identifying relationship, or
   2) A mandatory (total) nonidentifying relationship, or
   3) An optional (partial) nonidentifying relationship.
b) A nonidentifying relationship and a nonspecific relationship may be recursive, i.e., relating an instance of an entity to another instance of the same entity.
c) An identifying relationship may not be recursive.

[86]The second method of naming the relationship from the child perspective, i.e., using the direct object of the phrase in place of the entire verb phrase as in [B13], has been dropped from this standard.

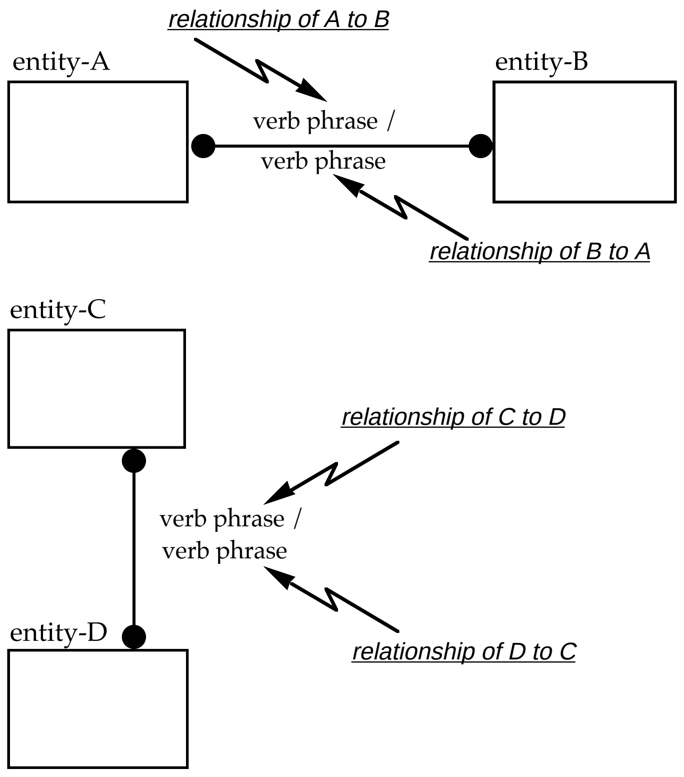


Figure 91—Nonspecific relationship syntax

9.5.3.3 Total relationship/partial relationship

a) In an identifying relationship, a child entity instance shall be associated with exactly one instance of its parent entity.

b) In a total (mandatory) nonidentifying relationship, a child entity instance shall be associated with exactly one instance of its parent entity.

c) Only a nonidentifying relationship may be partial, i.e., optional from the perspective of the child.

d) In a partial relationship, a child entity instance shall be associated with either zero or one instance of its parent entity.

9.5.3.4 Independent entity/dependent entity

a) If an entire foreign key is used for all or part of an entity's primary key, then the entity shall be classified as dependent.

b) If only a portion of a foreign key, or no foreign key attribute at all, is used for an entity's primary key, then the entity shall be classified as independent.

c) The child entity in an identifying relationship shall always be a dependent entity.

d) The child entity in a nonidentifying relationship shall be an independent entity unless the entity is also a child entity in some identifying relationship.

e) The parent entity in an identifying relationship shall be an independent entity unless it is also the child entity in some other identifying relationship.

f) In other than an entity-relationship level view, a category (subclass) shall always be classified as a dependent entity.

g) A category entity may not be a child in an identifying relationship unless the primary key contributed by that relationship is already completely contained within the primary key of the category entity.

9.5.3.5 Cyclic relationships

a) Relationship cycles are allowed; however, the cycle shall include at least one nonidentifying relationship.

9.5.3.6 Relationship label

a) Relationship labels (providing the forward and reverse verb phrases) shall be optional (although they should be used for clarity).
b) When a forward verb phrase is omitted, “has” should be used to read the relationship.

9.6 Entity generalization

A *generalization structure* (*categorization structure*) is used to represent a situation in which an entity is a *type* (*category*) of another entity. Generalization of value classes is discussed in 5.3. Generalization and categorization in key-style and identity-style views are based on the same basic principle. In key-style views, however, generalization is based only on common attributes and relationships; no behavior is represented in the views.

9.6.1 Entity generalization semantics

9.6.1.1 Generalization structure

Entities are used to represent the notion of “things about which we need information.” Since some real world things are categories of other real world things, some entities must, in some sense, be categories of other entities. For example, suppose employees are something about which information is needed. Although there is some information needed about all employees, additional information may be needed about salaried employees that is different, from the additional information needed about hourly employees. Therefore, the entities `salaried-Employee` and `hourly-Employee` are categories of the entity `employee`. In an IDEF1X view, these entities may be arranged in a generalization structure. A *generalization structure* is an identity connection between one entity, referred to as the *generic entity*, and another entity, referred to as a *category entity.*

In another case, a category entity may be needed to express a relationship that is valid for only a specific category or to document the relationship differences among the various categories of the entity. For example, a `full-Time-Employee` may qualify for a `benefit`, while a `part-Time-Employee` may not.

9.6.1.2 Category cluster

A *category cluster* is a set of one or more generalization structures having a common generic entity. An instance of the generic entity may be an instance of at most one of the category entities in the cluster, and each instance of a category entity is exactly one instance of the generic entity. Each instance of the category entity represents the same real-world thing as its associated instance in the generic entity. From the previous example, `employee` is the generic entity and `salaried-Employee` and `hourly-Employee` are the category entities. There are two generalization structures in this cluster, one between `employee` and `sal-aried-Employee` and one between `employee` and `hourly-Employee`.

Since an instance of the generic entity may not be an instance of more than one of the category entities in the cluster, the category entities are mutually exclusive. In the example, this rule implies that an employee cannot be both salaried and hourly. However, an entity may be the generic entity in more than one category cluster, and the category entities in one cluster are not mutually exclusive with those in others. For example, `employee` could be the generic entity in a second category cluster with `full-Time-Employee` and `part-Time-Employee` as the category entities. An instance of `employee` could be associated with an instance of either `salaried-Employee` or `hourly-Employee` and with an instance of either `full-Time-Employee` or `part-Time-Employee`. An instance may change its category without violating any language rules, for example, a full-time employee might begin working part time.

9.6.1.3 Complete/incomplete

In a *complete category cluster*, every instance of the generic entity is associated with an instance of a category entity shown in the view, i.e., all the possible categories are present. In the example, each employee is either full time or part time, so the second cluster is complete. In an *incomplete category cluster*, an instance of the generic entity may exist without being an instance of any of the category entities shown in the view, i.e., some categories are omitted from the view. For example, if some employees are paid commissions rather than an hourly wage or salary, the first category cluster would be incomplete.[87]

Although the generalization structures themselves are not named explicitly, each generic entity to category entity structure can be read as "can be." For example, "an `employee` *can be* a `salaried-Employee`." If the cluster is complete, each structure may be read as "must be." For example, "an `employee` *must be* a `full-Time-Employee` or `part-Time-Employee`." The structure is read as "is a/an" from the reverse direction. For example, "an `hourly-Employee` *is an* `employee`."

9.6.1.4 Discriminator

An attribute in the generic entity, or in one of its generic ancestors, may be designated as the discriminator for a specific category cluster of that entity. A *discriminator* is an attribute whose value determines the category of an instance of the generic. The values of the discriminator are one-to-one equivalent to the names of the category entities. In the previous example, the discriminator for the cluster including the salaried and hourly categories might be named `employee-Pay-Type`. If a cluster has a discriminator, that discriminator must be distinct from all other discriminators in the generic.

9.6.2 Entity generalization syntax

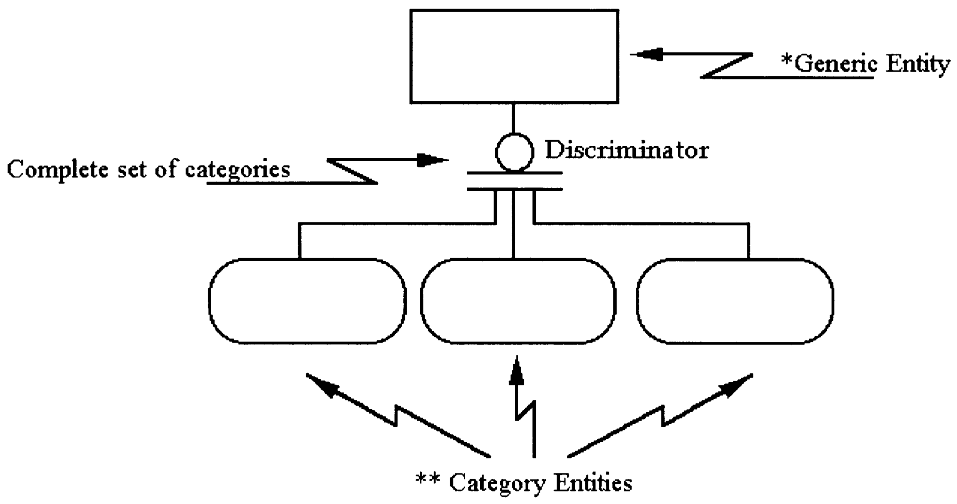
9.6.2.1 Category cluster

a) A category cluster shall be shown as a line extending from the generic entity to a circle that is underlined. Separate lines shall extend from the underlined circle to each of the category entities in the cluster. Each line pair, from the generic entity to the circle and from the circle to the category entity, shall represent one of the generalization structures in the cluster (see Figure 92 and Figure 93).
b) Category entities shall always be identifier-dependent.
c) The generic entity shall be independent unless its identifier has migrated through some relationship.

9.6.2.2 Complete/incomplete cluster

a) If the category cluster circle has a double underline, it shall indicate that the set of category entities is complete (see Figure 92).
b) A single line under the category cluster circle shall indicate an incomplete set of categories (see Figure 93).

[87]The single and double underline notation has a different meaning for key-style and identity-style views. In an identity-style view, the double underline indicates an *abstract class*, one in which each instance must also be an instance of one of the subtypes (categories). In an identity-style view, the double underline does not mean that all categories are displayed in the view, only that the class is abstract.



*The generic entity may be an identifier-independent entity (as shown) or an identifier-dependent entity depending on its relationships.

**Category entities will always be identifier-dependent entities.

Figure 92—Complete categorization structure syntax

9.6.2.3 Discriminator

a) If a category cluster has a discriminator, the name of the discriminator attribute shall be written with the category cluster circle.

9.6.3 Entity generalization rules

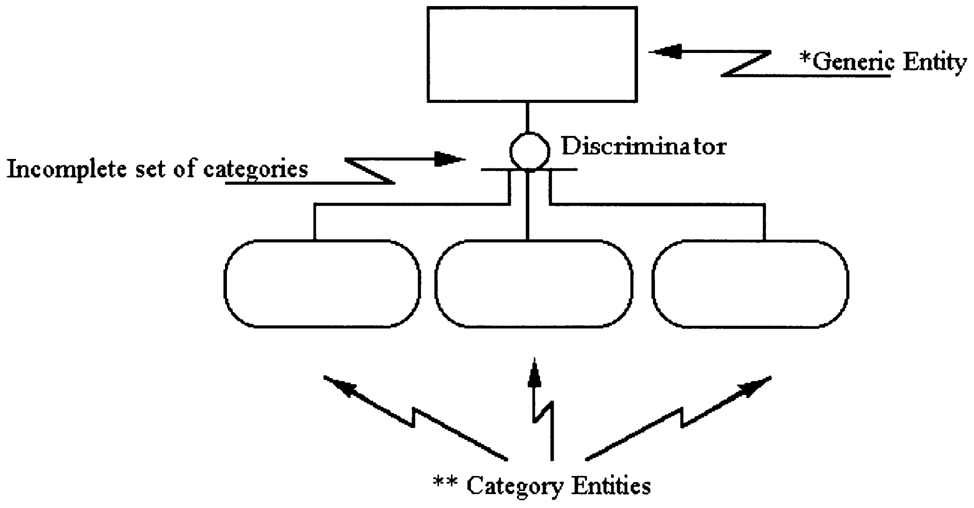
9.6.3.1 Generalization taxonomy

a) A category entity may have only one generic entity, i.e., it may only be a member of the set of categories for one category cluster.[88]
b) A category entity in one categorization structure may be a generic entity in another categorization structure.
c) An entity may have any number of category clusters in which it is the generic entity.

9.6.3.2 Category primary key

a) The primary key attribute(s) of a category entity shall be the same as the primary key attribute(s) of the generic entity. However, attribute role names may be assigned in the category entity.
b) A category entity may not be a child entity in an identifying relationship unless the primary key contributed by the identifying relationship is already completely contained within the primary key of the category.

[88]While multiple inheritance is supported for identity-style views, only single inheritance is available within key-style views.



*The generic entity may be an identifier-independent entity (as shown) or an identifier-dependent entity depending on its relationships.

**Category entities will always be identifier-dependent entities.

Figure 93—Incomplete categorization structure syntax

9.6.3.3 Generic ancestor

a) No entity may be its own generic ancestor, that is, no entity may have itself as a parent in a categorization structure, nor may it participate in any series of categorization structures that specifies a cycle.

9.6.3.4 Discriminator

a) If a discriminator is assigned,
   1) All instances of a category entity shall have the same discriminator value, and
   2) All instances of different categories shall have different discriminator values.
b) No two category clusters of a generic entity may have the same discriminator.
c) The discriminator (if any) of a complete category cluster may not be an optional attribute.
d) The values of the discriminator (if any) of a complete category cluster shall correspond one-to-one to the names of the categories in the cluster.
e) The values of the discriminator (if any) of an incomplete category cluster shall correspond one-to-one to the names of the categories shown for the cluster; however, additional values are permitted, corresponding to the names of categories not shown.

9.7 Primary and alternate key

A *key*, either primary or alternate, represents a uniqueness constraint over the values of properties. Uniqueness constraints were expressed as primary keys and alternate keys in earlier versions of IDEF1X because these versions did not include the notion of identity.[89] When the concept of identity is not available, a precise specification of *distinct instance* must use attributes; there is nothing else to use. Without identity, a simple way to state what distinct instance means is in terms of a uniqueness constraint over attributes that have a value for every instance—in other words, a primary key constraint. Key-style modeling continues to support the use of primary and alternate keys.

9.7.1 Primary and alternate key semantics

9.7.1.1 Primary key

The notion that instances must not agree on all attribute values was made precise in the original versions of IDEF1X with a *primary key* constraint. In practice, a constraint like the primary key constraint often occurs as a business rule. For example, customer numbers are assigned with the intent that they uniquely identify customers. If a uniqueness constraint reflects a business rule or is inherent in the sense of the entity, that uniqueness constraint should be stated.

9.7.1.2 Candidate key

A *candidate key* of an entity consists of one or more attributes for which no two instances of the entity will agree on the values. For example, the attribute `purchase-Order-Identifier` may uniquely identify an instance of the entity `purchase-Order`. A combination of the attributes `account-Identifier` and `check-Identifier` may uniquely identify an instance of the entity `check`.

In key-based and fully attributed views, every entity must have at least one candidate key. In some cases, an entity may have more than one candidate key. For example, the attributes `employee-Id` and `social-Security-Nbr` may both uniquely identify an instance of the entity `employee`. A candidate key may include attributes that have no value in certain instances. Such a candidate may not be chosen as the primary key.

9.7.1.3 Alternate key

When more than one candidate key exists, then one candidate key is designated as the *primary key* and each other candidate key is designated an *alternate key* (AK). If only one candidate key exists, then it is the primary key.

9.7.2 Primary and alternate key syntax

9.7.2.1 Primary key

a) Attributes that compose the primary key of an entity shall be placed at the top of the attribute list within the entity rectangle.
b) Attributes that compose the primary key of an entity shall be separated from the nonprimary key attributes by a horizontal line (see Figures 87 and 94).

9.7.2.2 Alternate key

a) An alternate key shall be assigned a unique integer number.

[89]In identity-style modeling, a uniqueness constraint is treated as simply another kind of constraint.

b) An alternate key shall be shown by placing `AK` plus the alternate key number in parentheses to the right of the attribute name, e.g., `(AK1)`, as shown in Figure 94.
c) An individual attribute may be identified as part of more than one alternate key.
d) A primary key attribute may also serve as part of an alternate key.

Alternate Key Syntax

```
attribute-Name (AKn)[ (AKm). . .]
or
attribute-Name (AKn[, AKm. . .])
```

where `n`, `m`, etc., uniquely identify each Alternate Key that includes the associated attributes, and where an Alternate Key consists of all the attributes with the same identifier.

Example

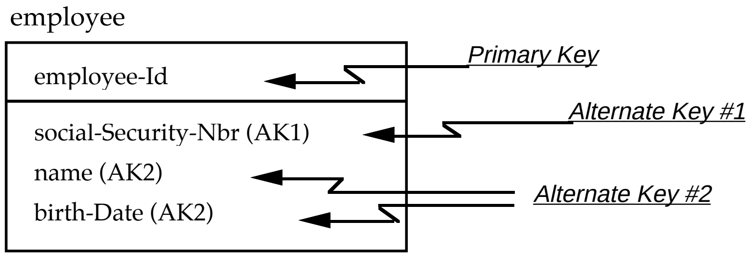


Figure 94—Alternate key syntax

9.7.3 Primary and alternate key rules

9.7.3.1 Primary key/alternate key composition

a) In a key-based or fully attributed view, every entity shall have a primary key.
b) In addition to a primary key, an entity may have one or more alternate keys specified.
c) A key (primary or alternate) may consist of a single attribute or combination of attributes.
d) Each instance of the entity shall have a value for each attribute included in the primary key.
e) An individual attribute may be part of more than one key (primary or alternate). This rule includes the case of a primary key attribute also serving as part of an alternate key. This rule includes the case of a foreign key (migrated) attribute being part of an alternate key.
f) A key (primary or alternate) shall contain only those attributes that contribute to the entity's unique identification.
g) If a primary key is composed of more than one attribute, the value of every nonkey attribute shall be functionally dependent upon the entire primary key.
h) Attributes that form primary and alternate keys of an entity shall either be owned by the entity or migrated through a relationship (see 9.8).
i) Every attribute that is not part of a key (primary or alternate) shall be functionally dependent only upon the primary key and each of the alternate keys. In other words, no such attribute's value may be determined by another such attribute's value.

9.7.3.2 Primary key/alternate key optionality

a) Each attribute specified as part of a primary key shall have a value.

b) An attribute specified as part of an alternate key may have no value.

9.7.3.3 Primary keys in generalization structures

a) A category entity shall have the same primary key as its generic entity.

9.8 Foreign key

A *foreign key* is an attribute or group of attributes in an entity that designates an instance of a related entity. Foreign keys were originally made a part of the IDEF1X notions of relationship and generalization because they were at the time the most definite, well-defined, and easily understood way to specify precisely what relationship and generalization meant. For example, the use of foreign keys permitted sample instance tables to be drawn in a clear and consistent way.

The concepts of relationship and generalization were made precise with foreign keys in earlier versions of IDEF1X. When identity is not available, a precise specification of relationship must use attributes; there is nothing else to use. Without identity, the only real choice was to express relationships in terms of foreign keys or to leave it unspecified. Key-style modeling continues to support the use of foreign keys.

9.8.1 Foreign key semantics

In key-style views, relationships and generalization structures are expressed using foreign keys. If a relationship or generalization structure exists between two entities, then all the attributes that form the primary key of the parent or generic entity are migrated as attributes of the child entity or inherited as attributes of the category entity. These attributes are referred to as *foreign key* attributes.

For example, if a relationship exists between the entity `project` as a parent and the entity `task` as a child, then the primary key attributes of `project` will be foreign key (migrated) attributes of the entity `task`. In this example, if the attribute `project-Id` is the primary key of `project`, then `project-Id` will also be a foreign key (migrated) attribute of `task`.

9.8.1.1 Foreign keys in generalization structures

In a generalization structure, both the generic entity and the category entity represent the same real-world thing. Therefore, the primary key for each category entity is inherited through the generalization structure from the primary key of the generic entity. For example, if `salaried-Employee` is a category entity of the generic entity `employee` and the attribute `employee-Id` is the primary key for the entity `employee`, the attribute `employee-Id` will also be the primary key for `salaried-Employee`.

9.8.1.2 Foreign keys in relationships

A foreign key attribute may be used as either a partial or complete primary key, as an alternate key, or as a nonkey attribute within an entity. If all the primary key attributes of a parent entity are migrated as part of the primary key of the child entity, then the relationship through which the attributes were migrated is an *identifying relationship*. If any of the migrated attributes are not part of the primary key of the child entity, then the relationship is a *nonidentifying relationship* (see 9.5.1).

For example, if tasks are only uniquely numbered within a project, then the migrated attribute `project-Id` will be combined with the owned attribute `task-Id` to specify the primary key of `task`. The entity `project` will have an identifying relationship with the entity `task`. If on the other hand, the attribute `task-Id` is always unique, even across projects, then the migrated attribute `project-Id` will be a nonkey attribute of the entity `task`. In this case, the entity `project` will have a nonidentifying relationship with the entity `task`.

When only a portion of a migrated primary key becomes part of the primary key of the child entity, with the remainder becoming nonkey attribute(s) of the child, the contributed foreign key is called a *split key.* If a key is split, the relationship is nonidentifying.

9.8.1.3 Multiple relationships

In some cases, a child entity may have multiple relationships to the same parent entity. The primary key of the parent entity will appear as a foreign key attribute in the child entity for each relationship. For a given instance of the child entity, the values of the migrated attributes may be different for each relationship, i.e., two different instances of the parent entity may be referenced. A bill of materials structure, for example, can be represented by two entities `part` and `assembly-Structure` (see Figure 96). In this example, the entity `part` has a dual relationship as a parent entity to the entity `assembly-Structure`. The same part sometimes acts as a component from which assemblies are made, i.e., a part may be a component in one or more assemblies, and sometimes it acts as an assembly that itself has one or more component parts. If the primary key for the entity `part` is `part-Nbr`, then `part-Nbr` will appear twice in the entity `assembly-Structure` as a migrated attribute. However, since an attribute of a given name may appear only once in any entity, the two occurrences of `part-Nbr` in `assembly-Structure` are merged unless a role name is assigned to one or both.

9.8.1.4 Role naming

When the same foreign key attribute migrates into an entity through more than one relationship or is inherited through a generalization structure, a *role name* may need to be assigned to each occurrence to distinguish among them. If an instance of the entity can have one value for one occurrence and a different value for another occurrence, then each occurrence of the migrated attribute must have a different name. Typically, each occurrence is given a role name although one occurrence may retain the name of its primary key source.

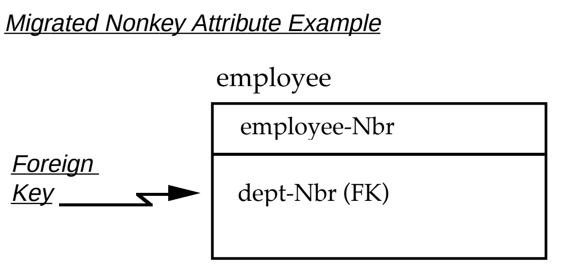
On the other hand, if each instance of the entity must have the same value for two or more migrated attribute occurrences, each occurrence of the migrated attribute must have the same name. In Figure 96, role names of `component-Nbr` and `assembly-Nbr` have been assigned to distinguish between the two migrated attribute occurrences of `part-Nbr`.

Attribute role naming may also be used with a single occurrence of a migrated (inherited) foreign key attribute. Although not required in this circumstance, a role name may convey more precisely the attribute's meaning (i.e., clarify the role it plays) within the context of the entity.

9.8.2 Foreign key syntax

9.8.2.1 Foreign key representation

a) A foreign key shall be depicted by placing the names of each foreign key attribute inside the entity rectangle.
b) Each foreign key attribute label shall consist of the attribute name followed with the letters `FK` in parentheses, i.e., `(FK)`, as shown in Figure 95.
c) If any foreign key attribute does not belong to the primary key of the child entity, the attribute shall be placed below the line, and the entity shall be classified as identifier-independent with respect to this relationship (see 9.5).
d) If all migrated attributes belong to the primary key of the child entity,
   1) Each shall be placed above the horizontal line, and
   2) The entity rectangle shall be drawn with rounded corners to indicate that the identifier (primary key) of the entity is dependent upon an attribute migrated through a relationship.
e) A foreign key (migrated) attribute may be part of an alternate key.



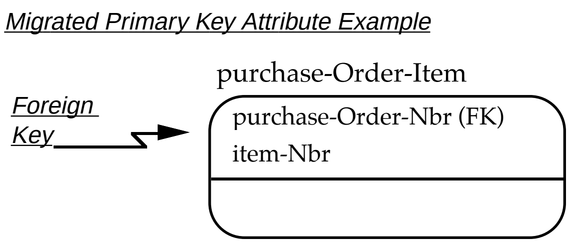


Figure 95—Foreign key syntax

9.8.2.2 Role naming

a) When an attribute label contains a role name,
   1) The role name shall precede the foreign key attribute name, and
   2) A period (“.”) shall be used to separate the role name and the original name, with no spaces immediately before or after the period (see Figure 96).

b) When an attribute with a role name is migrated or inherited into another entity, only the role name shall be displayed in that entity.

9.8.3 Foreign key rules

9.8.3.1 Primary key/foreign key correspondence

a) Every primary key attribute of a parent entity in a relationship shall be a foreign key (migrated) attribute in the related child entity.

b) Every primary key attribute of a generic entity in a generalization structure shall be a foreign key (inherited) attribute in the related category entity.

c) Every primary key attribute of a generic entity in a generalization structure shall be part of the category entity’s primary key.

9.8.3.2 Foreign key/primary key correspondence

a) An entity shall contain a set of foreign key attributes for each relationship in which it is the child.

b) An entity shall contain a set of foreign key attributes for the generalization structure in which it is the category entity.

c) Every foreign key attribute of a child or category entity shall represent an attribute in the primary key of a related parent or generic entity.

d) Every foreign key attribute shall reference one and only one of the primary key attributes of the parent. An attribute `a` references another attribute `b` if `a = b` or `a` is a direct or indirect subtype of

Role Name Syntax:

```
role-Name.attribute-Name (FK)
```

Example

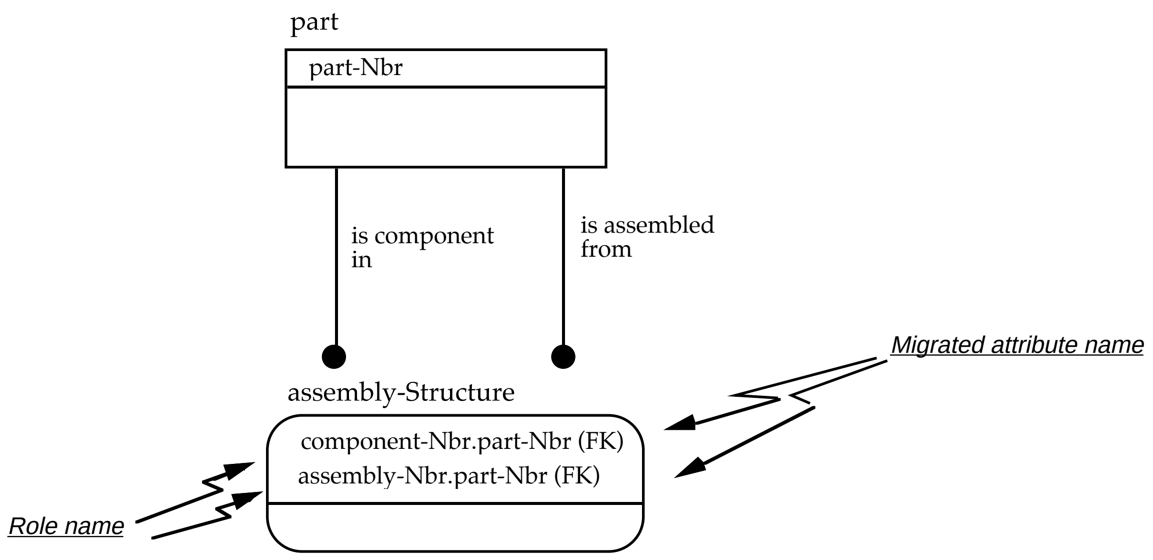


Figure 96—Key-style role name syntax

`b`. An attribute `a` is considered a subtype of `b` if `a` is an alias for `c` and `c` is a subtype of `b`, or `a` is a subtype of `c` and `c` is an alias for `b`.[90]

e) A child entity may not contain two entire foreign keys identifying the same instance of the same ancestor (parent or generic) for every instance of the child unless these foreign keys are contributed via separate relationship paths containing one or more intervening entities between the ancestor and the child (see also 9.9).

f) A foreign key attribute may be part of more than one foreign key provided that the attribute always has the same value for these foreign keys in any given instance of the entity.

g) The number of attributes in the set of foreign key attributes shall be the same as the number of attributes of the primary key of the parent or generic entity.

9.8.3.3 Naming/role naming

a) The name of a foreign key attribute may be
   1) A role name,
   2) An alias for a role name, or
   3) The same name as the original (owned) attribute in the related entity.

b) Each role name assigned to a foreign key attribute shall be unique within the view. However, the same role name may be assigned to multiple foreign key attributes to state a common ancestor constraint, as described in 9.9.

c) A role name shall be a value class name and, as such, shall be a noun phrase.

d) A role name shall conform to the naming rules of a value class name.

[90]The intent of this rule is that for every role name it be clear exactly what it is a role name for.

9.9 Common ancestor constraint

The constraint language is used to state constraints in identity-style modeling. Without this language, many constraints cannot be stated formally in key-style modeling; notes are used instead to provide informal statements of these constraints. However, one type of constraint that can be stated in a key-style view is the *common ancestor constraint.*

A common ancestor constraint is expressed by the use of foreign keys and (possibly) role names for foreign keys with constraint notes,[91] as illustrated in Figures 98 through 100. This discussion is presented to support those continuing to use foreign keys and wishing to express common ancestor constraints in the key-style manner.

9.9.1 Common ancestor constraint semantics

A *common ancestor constraint* involves two or more paths between a child entity and one of its ancestors. Each path is a relationship or generalization (or a series of such relationships) in which the child in one is the parent in the next.

For example, if a `hotel` entity has two child entities, `room` and `tv`, and each of these has a common child called `tv-In-A-Room`, then there are two paths between `hotel` and `tv-In-A-Room`, one through `room` and one through `tv`. A common ancestor constraint describes a restriction on the instances of the ancestor entity (e.g., `hotel`) to which each instance of the descendent entity (e.g., `tv-In-A-Room`) may be related. A common ancestor constraint can state that either a descendent instance must be related to the <u>same</u> ancestor instance through each path, or that it must be related to a <u>different</u> ancestor instance through each path.

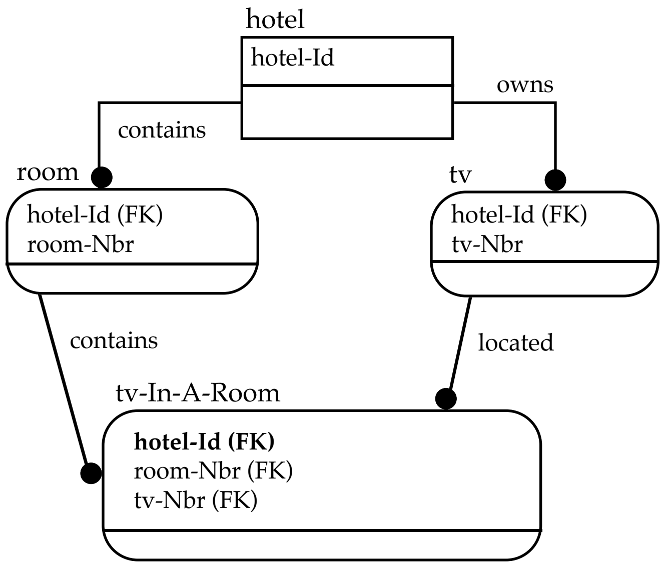


Figure 97—Common ancestor required to be the same

[91]Identity-style modeling uses the constraint language to state such business rules. The two are equivalent. For example, in key-style modeling the standard meaning of a single foreign key in a child having a common ancestor implies the following constraint: *for a tv-in-a-room, the hotel that contains the room must be the hotel that owns the tv.* In identity-style modeling, this constraint would be declared explicitly. The specification language to state this constraint is given in 6.7. Common ancestor constraints can be detected syntactically in a key-style view based upon the definitions of relationships, foreign keys, role names, and value class hierarchy. The corresponding constraint for an identity-style view can, therefore, be generated automatically.

In the example above, a common ancestor constraint might state that "the hotel that contains the room must be the hotel that owns the TV," i.e., a `tv-In-A-Room` instance must be related to the same `hotel` instance through both paths. The view that expresses this constraint is shown in Figure 97.

On the other hand, an alternative (perhaps nonsensical) distinct ancestor constraint could say "hotels cannot use TVs that belong to them," i.e., a `tv-In-A-Room` instance must be related to a different `hotel` instance through each path. The view that expresses this constraint is shown in Figure 98.

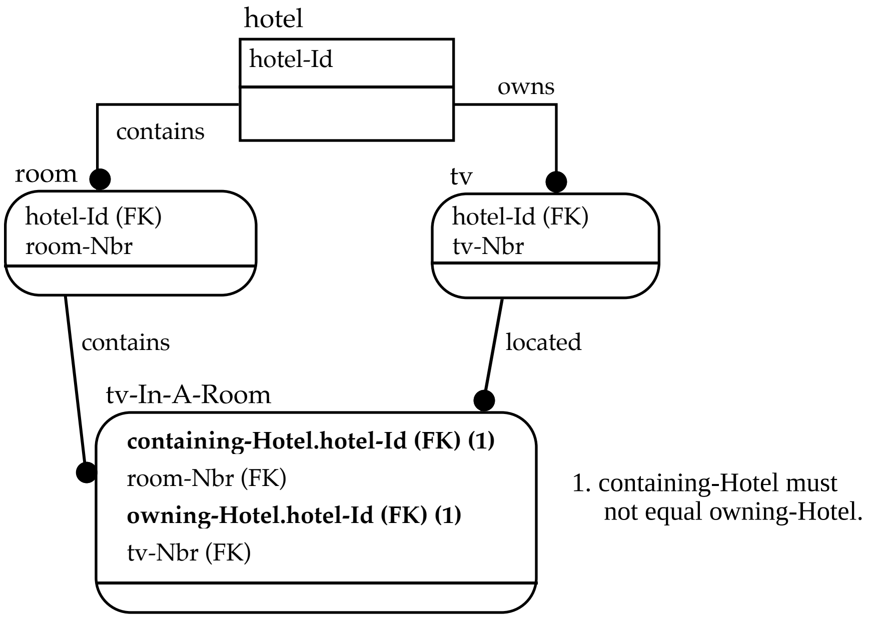


Figure 98—Distinct ancestor

In this example, the third possibility is that "hotels can use TVs belonging to any hotel." This example would imply that the related instances of `hotel` may be either the same or different. Since there is no restriction in this situation, no common ancestor constraint note is needed.

9.9.2 Common ancestor constraint syntax

There is no specific syntax for expressing a key-style common ancestor constraint, other than correct use of the involved modeling constructs.

9.9.3 Common ancestor constraint rules

9.9.3.1 Nonidentifying relationship

a) If any of the paths includes a nonidentifying relationship, a note should be used to record the constraint, as shown in Figure 100.

9.9.3.2 Identifying relationship

However, if each of the path segments is an identifying relationship, then the primary key of the ancestor entity will migrate all the way to the descendent entity along all paths, resulting in multiple occurrences of the migrated attribute in the descendent (see, for example, Figure 97). In this case, role names may be needed in conjunction with the common ancestor constraint. There are four possible situations:

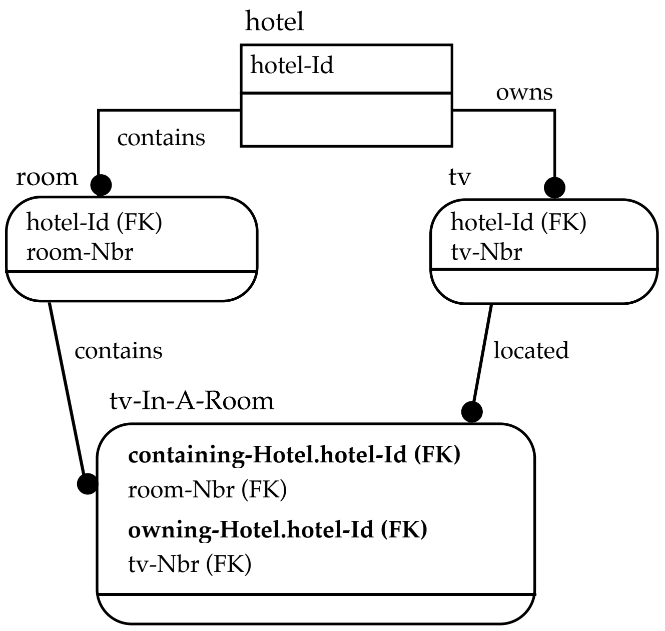


Figure 99—Common ancestor with no restriction

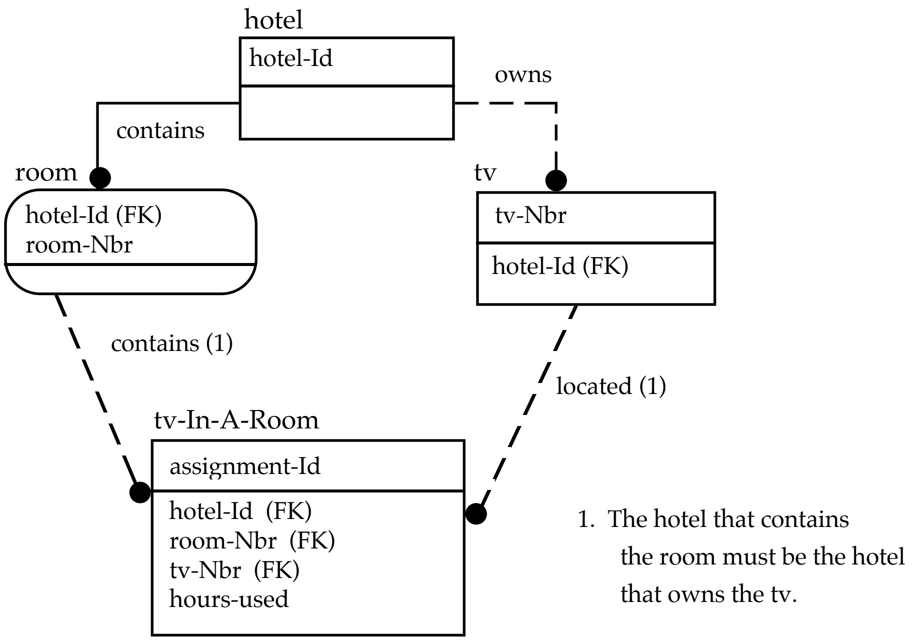


Figure 100—Common ancestor constraint note

a) The business rule states that the ancestor instances must always be the same. This rule means a descendent instance must be related to the same ancestor instance through all paths. In this situation, either
   1) No role names shall be assigned, or
   2) The same role name shall be assigned to all occurrences of the migrated attribute in the descendent entity.

Giving the same role name to all occurrences is sufficient to express the common ancestor constraint, so a constraint note is not needed. This situation is illustrated in Figure 97, corresponding to the identity-style constraint example shown in Figure 70.

b) The business rule states that the ancestor instances must always be different. This rule means a descendent instance must be related to a different ancestor instance through each path. In this situation,
   1) A different role name shall be assigned to each occurrence of the migrated attribute in the descendent, and
   2) A common ancestor constraint note shall be added, stating that the values must be different.

   This situation is illustrated in Figure 98.
c) The business rule states that all of the ancestor instances may be the same or may be different. In this situation,
   1) A different role name shall be assigned to each occurrence of the migrated attribute in the descendent, but
   2) No common ancestor constraint note need be added.

   Common ancestor constraint notes are not needed in this case because giving the occurrences different role names allows, but does not require, their values to be different. This situation is illustrated in Figure 99.
d) The business rule states that some of the ancestor instances may be the same or may be different, and others must be the same or must be different. In this case, multiple common ancestor constraints shall be stated, one for each of the situations described above.

9.10 Key-style view level

A literal translation of key-style view levels to identity-style view levels by a direct mapping of modeling construct is not possible because the goals of a modeling style determine the constructs. For example, identity-style modeling focuses on behavior as well as structure, and different constructs may be needed at different levels to assist in reasoning about factors important at that point in time.

The modeling style (key style or identity style) determines which variation of level applies. However, the fundamental notion of view levels is the same for both key- and identity-styles. Each level is intended to be distinct, defined in terms of the modeling constructs to be used. Any view is to be clearly at one level. This is done for two reasons. First, limiting each level to the appropriate set of modeling constructs promotes modeling what is appropriate to the level and only what is appropriate to the level. Second, having distinct levels provides a clear work product definition for management.

There are four levels in key-style modeling.[92] Like the levels of identity-style views (see 8.2), each key-style view level must balance the admittedly conflicting goals of any view: be understandable to users and be useful to developers. The three conceptual schema levels of key-style modeling—entity-relationship, key-based, and fully attributed—differ in the syntax and semantics that each allows. The primary differences are:

a) Entity-relationship views specify no keys.
b) Key-based views specify keys and some nonkey attributes.
c) Fully attributed views specify keys and all nonkey attributes.

These three view levels provide the structural information needed to design efficient databases for a physical system. At a fourth level, the key-style graphic syntax is often used informally to describe the physical database structure. This level can be very useful in re-engineering current systems and provides a method for deriving data structure descriptions from existing data resources.

[92]See Annex B for a comparison of identity-style and key-style concepts and constructs.

The views in adjacent levels relate to each other by a mapping (transformation). The mapping is enabled by employing a consistent set of modeling concepts. Levels do not imply a particular pattern of development, e.g., waterfall, iterative, or fountain. The methodological development pattern determines the scope of the views and the order in which they are produced, but not their content. The content of a level is independent of the methodological development pattern. Table 15 summarizes the levels of key-style modeling.

Table 15—Summary of levels (key style)

Level of view	Characteristic modeling constructs	Primary intent
1	Entities, relationships, illustrative attributes **(Entity relationship level)**	Specification and management of major areas of reusable assets and the applications and projects that use them.
2	Entity-relationship level plus keys and illustrative attributes **(Key-based level)**	Architecture and integration of features, prototypes, and releases within a project as well as across projects and applications.
3	Key-based level plus all attributes **(Fully attributed level)**	Complete specification of all semantics for a project or project release, independent of the implementation platform.
Technology-dependent levels	Database specifications **(Implementation level)**	Complete specification in terms of implementation platform constructs. May include multiple additional levels such as transform and implementation.

9.10.1 Key-style view level semantics

9.10.1.1 Level 1 (entity-relationship level)

An entity-relationship level view contains entities and relationships between entities. It may depict attributes for purposes of illustrating the nature of an entity.

This level may not contain any key declarations (primary, alternate, or foreign). Since an entity-relationship level view does not specify any keys, entities need not be distinguished as being dependent or independent, and relationships need not be distinguished as being identifying or nonidentifying. An entity-relationship level view may contain many-to-many (nonspecific) relationships.

9.10.1.2 Level 2 (key-based level)

The key-based level supports representation and reasoning about the most important concepts in the area of interest. The entities in this level are generalizations or other important, discovered entities—at least initially. An entity is "discovered" in the sense that it represents a concept already present in the minds of the people who understand the area. Key-based level views also include entities that have been "invented" (typically by abstracting from the discovered entities) to promote system resiliency in the face of change.

Key-based level views must be specific enough to support technical integration decisions. This level provides a consistent key structure, which is a prerequisite for integrated databases. This level is in many ways the most important and the most difficult. It requires deep insights into the needs of the enterprise and the rare technical ability to be both abstract and precise.

When fully attributed views are available over the scope of the key-based views, the key-based views can be updated to include all the entities, attributes, and relationships important to integration and reuse.

9.10.1.3 Level 3 (fully attributed level)

The fully attributed level completely specifies all entities. The fully attributed view begins as a subset view of a key-based view. All attributes are added.

9.10.1.4 Level 4 (implementation level)

The implementation level includes all entities needed for implementation of the fully attributed view on the chosen platform.[93] The implementation level view typically begins with a default transformation of the fully attributed level entities. The same considerations discussed in 8.2 for the Level 4 identity-style view apply to the key-style implementation level. Some rules enforced in Level 3 views may not be enforced at Level 4. For example, a Level 4 view will often show data redundancy that exists in an implemented system.

9.10.2 Key-style view level syntax

9.10.2.1 Level 1 (entity-relationship level)

a) In an entity-relationship level view, a relationship may be shown as either a solid or dashed line. At this level, solid and dashed lines are considered equivalent since no keys are expressed.

b) In an entity-relationship level view, an entity rectangle may not include an internal horizontal line (i.e., as used to separate the primary keys from the nonkey properties in other level key-style views) since no keys are expressed.

9.10.2.2 Level 2 (key-based level)

a) In a key-based level view, a relationship shall be shown as either a solid line (identifying relationship) or dashed line (nonidentifying relationship).

b) In a key-based level view, an entity rectangle shall include an internal horizontal line, used to separate the primary key attribute from the non-primary-key properties.

c) In a key-based level view, an entity rectangle shall be designated as either independent or dependent.

9.10.2.3 Level 3 (fully attributed level)

a) A fully attributed level view shall have the same display requirements as a key-based level view.

9.10.2.4 Level 4 (implementation level)

Data Definition Language (DDL) code is a textual form of the database management system (DBMS) view. The syntax is specific to each implementation platform and is not covered by this standard.

9.10.3 Key-style view level rules

Table 16 summarizes the modeling constructs appropriate to the various levels. The Implementation Level is part of a future version of this standard.

[93]This level was not defined in [B13].

Table 16—View levels and content

Construct	Level		
	Entity-relationship	**Key-based**	**Fully attributed**
Entities	Yes	Yes	Yes
Relationships	Yes	Yes (no many-to-many)	Yes (no many-to-many)
Generalizations	Yes	Yes	Yes
Primary keys	No (see Note)	Yes	Yes
Alternate keys	No (see Note)	Yes	Yes
Foreign keys	No (see Note)	Yes	Yes
Nonkey attributes	Typically, no (see Note)	Some	Yes
Notes	Yes	Yes	Yes
Normalized?	No	Yes	Yes
NOTE—Attributes are not distinguished as key or nonkey and are allowed, but not required, in entity-relationship level views. Optionality is not specified.			

9.10.3.1 Level 1 (entity-relationship level)

Some of the rules described in previous sections do not apply to all levels of views. The following exceptions are made for entity-relationship level views.

a) An entity need not have any attributes specified.
b) Entities do not have primary or alternate keys specified.
c) No entity has any migrated attributes (i.e., entities do not have foreign keys).
d) Entities are not required to be distinguished as identifier-independent or identifier-dependent. Category entities are considered to be dependent entities.
e) Parent cardinality (one, or zero or one) is unspecified in relationships.
f) Relationships are not required to be distinguished as identifying or nonidentifying.
g) Entity-relationship views may contain generalization structures.
h) Discriminator properties for category clusters are optional.

9.10.3.2 Level 2 (key-based level)

a) A key-based view shall contain entities, relationships, primary keys, and foreign keys.
b) The entities of a key-based view shall be distinguished as either dependent or independent.
c) The relationships of a key-based view shall be distinguished as either identifying or nonidentifying.
d) The parent cardinality for each nonidentifying relationship shall be designated as either mandatory or optional.
e) Each category cluster may have a discriminator property assigned.
f) Nonspecific relationships are prohibited.
g) Each entity of a key-based view shall contain a primary key and, if it has additional uniqueness constraints, an alternate key for each constraint.
h) Each entity of a key-based view shall contain a foreign key for every relationship or generalization structure in which it is the child or category, respectively.
i) A key-based view may contain nonkey attributes.

9.10.3.3 Level 3 (fully attributed level)

a) A fully attributed view has the same requirements as a key-based view.
b) A fully attributed view shall contain all nonkey attributes that are relevant to the subject of the view.

9.11 Key-style glossary

The general requirements for glossary entries are provided in 8.4. For key-style views, the following applies.

9.11.1 Key-style glossary semantics

Each key-style model shall be accompanied by narrative descriptions of all views, entities, and value classes (attributes). Narrative descriptions are held in a glossary common to all models within the context of the stated purpose and scope.

An alias is one of the alternative names by which an entity or value class (attribute) might be known. A list of aliases for an entity or value class (attribute) may be recorded in the glossary.

9.11.2 Key-style glossary syntax

No specified syntax exists for key-style glossaries.

9.11.3 Key-style glossary rules

For each view, entity, and value class (attribute), the glossary shall contain the following elements:

9.11.3.1 Name

a) The name shall be the unique name, defined in accordance with IDEF1X lexical rules.
b) The name shall be meaningful and should be descriptive in nature.
c) Abbreviations and acronyms shall be permitted.

9.11.3.2 Description narrative

a) The narrative description shall be a single declarative description of the common understanding of an entity or value class (attribute).
b) The narrative description shall be a single narrative description of the content of the view.
c) For an entity or value class (attribute), the narrative description shall apply to all uses of the associated entity or value class (attribute) name.

9.11.3.3 Aliases

a) The narrative description associated with an entity or value class (attribute) shall apply exactly and precisely to each of the aliases in its alias list.
b) Name variations to support computer automation may be listed as aliases.
c) A view may not have an alias.

9.11.3.4 Additional information

a) Optionally, additional information regarding the view, entity, or value class (attribute) may be provided, e.g., the name of the author, date of creation, date of last modification.
b) For a view, this additional information might also include level (e.g., entity relationship, key-based, fully attributed) completion or review status, and so on.

9.12 Key-style notes

Notes of a general nature and notes that document specific constraints are an integral part of the model. These notes may accompany the view graphics.

9.12.1 Key-style notes semantics

Several different types of assertions are made with foreign keys and role names. Assertions that cannot be made using role names are stated in notes. Such an assertion might specify a boolean constraint between two or more relationships. For example, an “exclusive OR” constraint states that for a given parent entity instance if one type of child entity instance exists, then a second type of child entity instance will not exist.

9.12.2 Key-style notes syntax

9.12.2.1 Graphic

a) A note that documents a specific constraint shall be represented in the view graphics by the symbol `(n)` placed adjacent to the impacted object (entity, relationship, or attribute).
b) A note that is general in nature shall be represented in the view graphics by the symbol `(n)` placed adjacent to the impacted object (entity, relationship, attribute, or view name).
c) The `n` in the symbol `(n)` shall be the identifier of the note in which the text of the note is documented.
d) A note identifier shall be a nonzero, unsigned integer.

9.12.3 Key-style notes rules

9.12.3.1 Note identifier

a) Note identifiers shall be unique within a view.

9.12.3.2 Note body

a) A single body of note text shall apply to the same note identifier if that note identifier is repeated within a view.

9.13 Key-style lexical rules

The lexical rules for this standard are provided in 4.2.3. When a “pure” key-style model (one that employs only key-style modeling constructs) is developed, a revision to these rules is needed to maintain compatibility with earlier versions of IDEF1X, particularly that version described in FIPS PUB 184 [B13].

In IDEF1X$_{97}$, class and property names are case sensitive. Terms beginning with an uppercase letter are considered *variables*, whereas terms beginning with a lowercase letter are considered *names*. Variables are used only in declarations made using the constraint language, and this language is not used with pure key-style models.

It is important that IDEF1X models that were compliant with earlier language standards remain compliant under this version. Therefore, the following lexical rule is provided for key-style modeling:

a) In a key-style view, when the intent is to provide a view consistent with FIPS PUB 184 [B13],
    1) The use of value classes shall be restricted to be consistent with FIPS PUB 184 [B13] domains, as specified in 9.2, and
    2) Entity and attribute names shall be case insensitive.

10. Formalization

10.1 Introduction

10.1.1 Objectives

The purpose of the formalization is to state precisely what the modeling constructs of IDEF$_{object}$ mean by providing for each construct a mapping to an equivalent set of sentences in a formal language. The graphic language and RCL can then be considered a practical, concise way to express the equivalent formal sentences.

IDEF$_{object}$ is based on the object model and logic. IDEF$_{object}$ is formalized using only a limited subset of logic—essentially what is covered in an introductory course.

Part of the formalization relies on a metamodel for IDEF$_{object}$. The relations assigned by interpretations in the formalism can be viewed informally as the familiar sample instance tables used in IDEF$_{object}$. The metamodel can be used independently of the detailed formalism.

The formalization is intended to support such areas as executable models, code generation, transformations to and from other modeling styles, and integration with other kinds of models. Each of these requires the precisely defined semantics provided by a formalization.

The immediate objective of the formalization of IDEF$_{object}$ is to provide a formal meaning for the constructs of IDEF$_{object}$ and, therefore, provide a formal meaning for any IDEF$_{object}$ view.

10.1.2 First order language, theory, and *model*

First order logic is a formal language analog of those aspects of natural language that are used to describe and reason about individual things and the relations among them.

a) An individual is denoted by a term, where a term is a constant, a variable, or a function symbol applied to terms.
b) A relation among individuals is denoted by a predicate symbol applied to terms.
c) An assertion about the relations among individuals is made by a logical sentence, which is
    1) A single proposition or multiple propositions connected by logical connectives such as `and`, `or`, and `if then`, where a proposition is
        i) A logical constant, where the logical constants are `true` and `false`,
        ii) A predicate symbol applied to terms;
    2) Or a quantified logical sentence, where the variables are quantified by `for all` and `for some.`
d) A logical sentence is closed if every variable is quantified.

(In a first order language, a variable can denote only an individual; in a higher order language, a variable can denote a predicate or function.)

A first order theory consists of a first order language in which the constant, function, and predicate symbols are restricted to a certain vocabulary, plus a set of closed, logical sentences (called axioms) in that language. A view is formalized as a first order theory.

An interpretation of a theory assigns to the constants of the theory elements from a nonempty set representing the individuals in a universe of discourse (UOD). An interpretation also assigns a function to each function symbol and a relation to each predicate symbol, where the elements in the functions and relations come from the UOD. (The term *relation* is being used here in its mathematical sense as a set of n-tuples, not in the specialized database sense.) As a result, the sentences in the theory become sentences about the UOD, and their truth is determined according to the reality of the UOD. In this way, each sentence is either true or false in the interpretation. An interpretation is a *model* for a theory if all axioms of the theory are true in the interpretation.

(The terminology of object modeling and logic collide on the term "model." In this clause, the term "view" is used as it is in the other clauses. The italicized term "*model*" is used in the logic sense: an interpretation of a theory for which all the axioms of the theory are true. The nonitalicized term "model" is used in the usual informal sense.)

10.1.3 Definition of correctness for a view

An IDEF$_{object}$ view is intended to be a conceptual model of a relevant subset of the things of concern to the enterprise. This UOD has an independent existence and reality outside any view of it. At any point in time, the UOD is in a certain state; in other words, certain individuals exist and have certain relationships to other individuals.

For any state of the UOD, some sentences are true and other sentences are false. For example, in a given state, the sentence that the part named top has a quantity on hand of 17 is either true or false. Similarly, the sentence that every vendor has a distinct vendor number is either true or false. Some states of the UOD are possible; others are impossible. For example, it is possible that a part named top has a quantity on hand of 17. It is impossible that the quantity on hand be Tuesday.

Over time, the state of the UOD changes. For example, the quantity on hand can become 23 as a result of a adding 6 to a part's inventory. Certain constraints have to hold in every state. For example, the constraint that every vendor have a distinct vendor number must be true in every state of the UOD. Certain rules govern the transition from one state to another. For example, adding 6 to a quantity on hand of 17 must yield 23.

The UOD encompasses all possible states.

For a view, state means the extents of each class and the values of all nonderived attributes and participant properties. If any value changes, the result is a new state. The theory for a view covers all states and state transitions.

An IDEF$_{object}$ view is correct if it matches the UOD in relevant ways. An IDEF$_{object}$ view is correct if:

a) For all possible states of the UOD, there is a corresponding state of the view in which
   1) All constraints of all classes are met,
   2) Every possible next state of the UOD corresponds to a next state of the view that can be reached from the current state of the view by a property of a class, and
   3) No impossible next state is reachable by a property of a class,
b) For all impossible states in direct conflict with the view, some constraint of some class is not met.

Formally, a modeler constructs a theory of the relevant portion of the enterprise so that the *models* of the theory match exactly the possible states and state transitions of the UOD. In other words, an IDEF$_{object}$ theory is correct if the sentences it insists be true (the axioms) are indeed true for all possible states and state transitions of the UOD and are false for all impossible states or state transitions.

IDEF$_{object}$ instance diagrams or tables for a view are a representation for the state of the view (see Figure 101).

In the context of the formalization, the sample instance tables present a portion of an interpretation for the theory. If all the constraints are met and all responsibilities whose preconditions are satisfied can be met without raising exceptions, then the sample instances are a portion of a *model* of the theory.

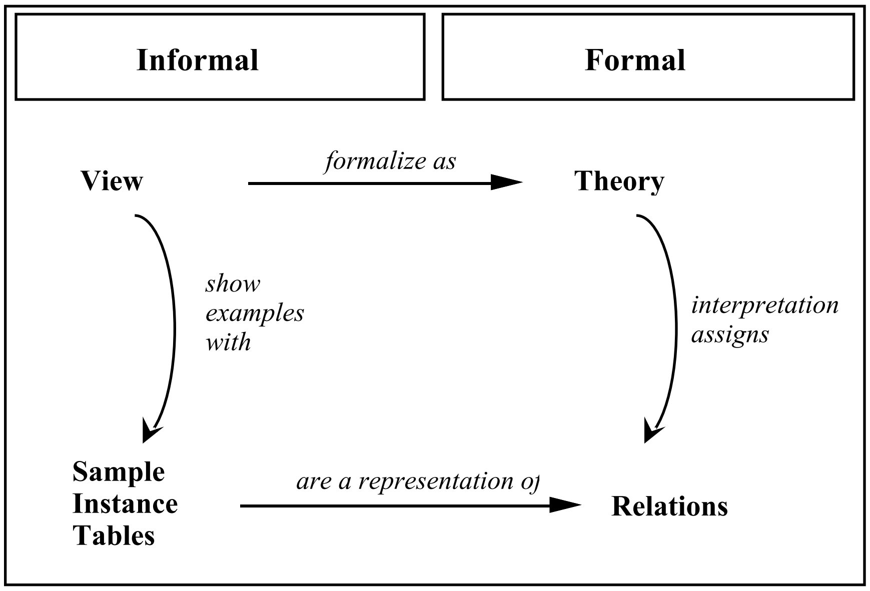


Figure 101—Informal and Formal correspondence

10.1.4 Formalizing a view by a theory

An IDEF$_{object}$ view consists of one or more classes, relationships, properties, and constraints expressed as diagrams and RCL. The formalization procedure generates a corresponding IDEF$_{object}$ first order theory.

In order to formalize an IDEF$_{object}$ view, the view is restated as instances and property values in the metamodel of IDEF$_{object}$. This is done by translating the graphical expression of the view into declaration RCL. The declaration RCL and realization RCL are mapped to logical sentences in definition clausal form. The mapping rests on the fact that in RCL, an object message is a logical proposition. These sentences become axioms in the theory for the view. The theory also includes the axioms common to all IDEF$_{object}$ theories, such as the clauses for the function and predicate symbols included in the vocabulary of the theory and the clauses for dynamic binding. In general, the axioms make statements about the metamodel and the current state of the view.

A state of a view is its set of instances and their nonderived attribute and participant property values. Initially, the state of a view is just what is declared into the metamodel for the view. An update message issued by an RCL query or within the realizations of a responsibility results in a new state. If a query or responsibility fails, the state remains unchanged.

An IDEF$_{object}$ theory uses a fixed set of function and predicate symbols. The user-defined RCL messages and realizations are mapped to formal propositions and axioms using predominately just two predicate symbols: has and is. A class named `Cn` is formally denoted by the term `#Cn`, where `#` is a function symbol, constrained by axioms to be a one-to-one function into a range disjoint from all others. There is no need to quantify over the fixed set of function and predicate symbols, so a first order language is sufficient to formally define such concepts as dynamic binding.

10.1.5 Formalization of IDEFobject

The formalization of IDEF$_{object}$ has two phases. First, a procedure is given whereby a valid IDEF$_{object}$ view can be restated as a first order theory in order to state precisely the semantics of a valid IDEF$_{object}$ view. Second, the procedure is applied to a metamodel of IDEF$_{object}$ in order to define formally the set of valid IDEF$_{object}$ views.

Applying the mapping to a valid meta model `Vm` of IDEF$_{object}$ yields a theory `Tm` of IDEF$_{object}$. A population of `Vm` is valid if and only if it is a *model* for `Tm`. This formally defines the set of valid IDEF$_{object}$ views. In other words, an IDEF$_{object}$ view is valid if and only if it is a valid population of the metamodel, or equivalently, its population of the metamodel is isomorphic to a *model* of the theory of the metamodel. `Vm` can be proven a valid view by proving that `Vm` is a valid population of `Vm`.

10.2 IDEF$_{object}$ metamodel

Throughout, the metamodel is used as a point of orientation. For this reason, it is presented early in the formalization, although strictly speaking it has no formal meaning without appeal to the material that follows it. Initially, it can be seen as a view like any other.

IDEF$_{object}$ can be used to model IDEF$_{object}$ itself. Such metamodels can be used for various purposes, such as repository design, tool design, or specification of the set of valid IDEF$_{object}$ models. Depending on the purpose, somewhat different models result. There is no "one right model." For example, a model for a tool that supports building models incrementally must allow incomplete or even inconsistent models. The metamodel for formalization emphasizes alignment with the concepts of the formalization. Incomplete or inconsistent models are not provided for.

A metamodel `Vm` for IDEF$_{object}$ is a model (i.e., a view) of the IDEF$_{object}$ constructs that is expressed using those constructs, so that there exists a valid instance of Vm that is a description of `Vm` itself. Every view implicitly includes the IDEF$_{object}$ metamodel and the formalization of a view includes the formalization of the metamodel.

The metamodel is based on the following ideas:

a) The elementary type of knowledge is that a class instance has a property value, represented in the form
```
        C: I has P: V.
```
b) The axioms for the interface of a metamodel are generated by populating the metamodel with itself by declarations of the above form.
c) The axioms for the realizations of a metamodel are generated by applying rewrite rules to the (RCL) realizations, represented in the form
    `C: I has P: V if`$_{def}$ `Sentence.`

The metamodel does not include properties to do dynamic binding. It was felt that it would be clearer to do the formalization of dynamic binding entirely in predicates and avoid the potential misunderstandings inherent in defining a language in terms of itself.

10.2.1 Interfaces

Figure 102 is an IDEF1Xobject class diagram for the metamodel.

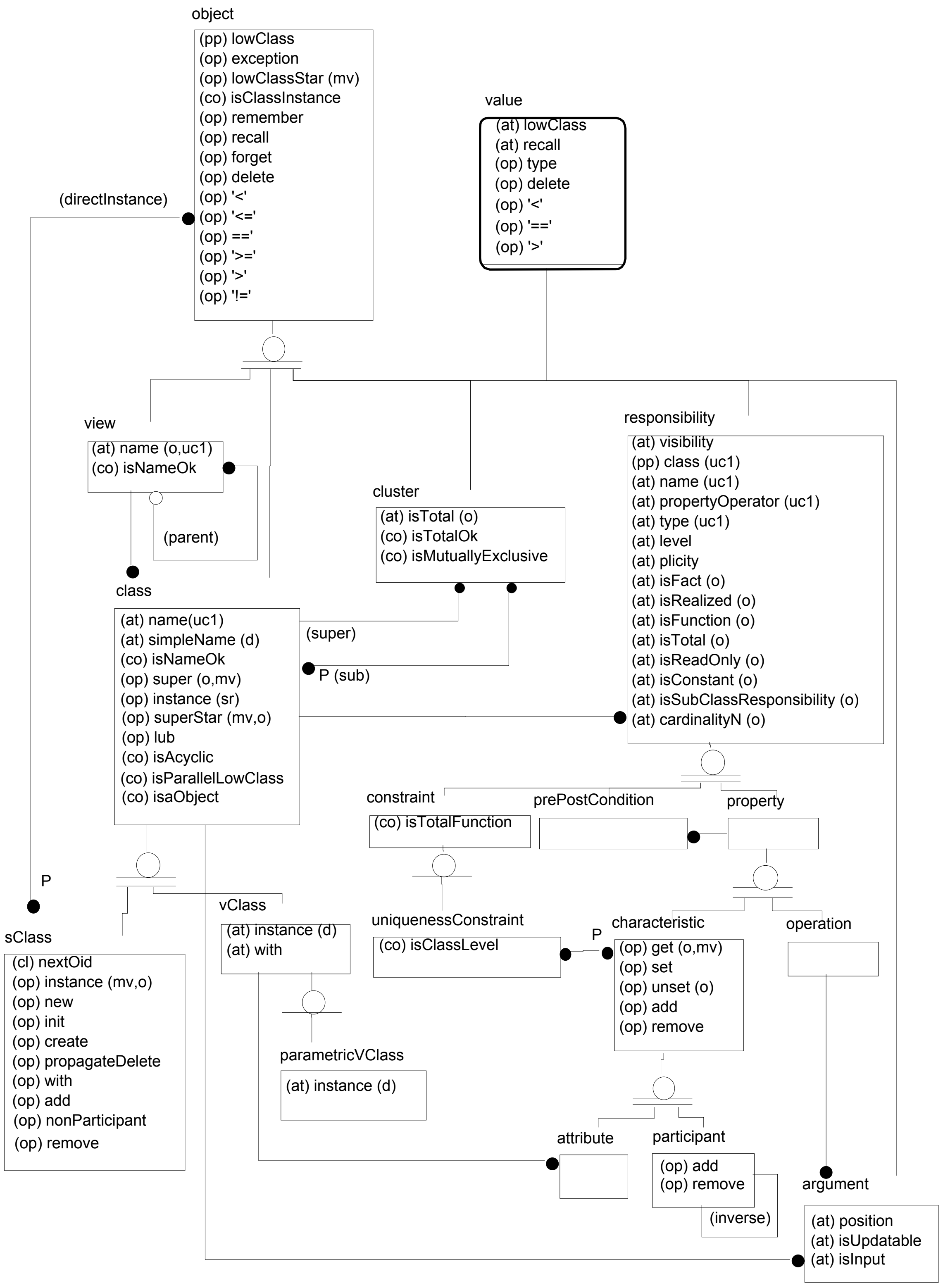


Figure 102—IDEFobject Class Diagram for the IDEFobject Metamodel

10.2.2 Realizations

Most of the realizations for the responsibilities are included here to provide a single point of reference. Further explanations are found in the subclauses on the formalization of the modeling constructs.

10.2.2.1 object

10.2.2.1.1 (op) object:lowClassStar

An object `I` has a `lowClassStar` of `C` if `I` has `C` as a direct or indirect `lowClass`.

```
object: Self has lowClassStar:C if_def
      Self has lowClass: C
      or Self has lowClass..lowClassStar: C.
```

10.2.2.1.2 (co) object:isClassInstance

Every object `I` is a direct or indirect instance of `class`.

```
object: Self has isClassInstance if_def
      Self has lowClassStar: #class.
```

10.2.2.2 view

10.2.2.2.1 (co) view:isNameOk

The name must be a qualified name, top down from a top (no parent) view.

```
view: Self has isNameOk if_def
      (if Self has parent..name: Vn
      then
           Self has name: (Vn:Name)
      endif).
```

10.2.2.3 class

10.2.2.3.1 (op) class:lub

`Self` has a least upper bound `Lub` with a class `C` if `Lub` is a common superclass and no subclass of `Lub` is a common superclass.

```
class: Self has lub: [C,Lub] if_def
      Self has superStar: Lub,
      C has superStar: Lub,
      not (Self has superStar: Lub2,
           C has superStar: Lub2,
           Lub != Lub2,
           Lub2 has superStar: Lub)
```

10.2.2.3.2 (op) class:super

`Self` has `super:S` if `Self` is a subclass in a cluster where `S` is the superclass.

```
class: Self has super: S if_def
```

```
      Self has highCluster..super: S.
```

10.2.2.3.3 (op) **class:superStar**

`Self` has `superStar:S` if `Self` is `S` or `Self` is a direct or indirect subclass of `S`.

```
class: Self has superStar: S if_def
      Self = S
      or Self has super..superStar: S.
```

10.2.2.3.4 (co) **class:isNameOk**

The name must be a qualified name, top down from a top view.

```
class: Self has isNameOk if_def
      Self has view..name: Vn,
      Self has name: (Vn:Name).
```

10.2.2.3.5 (co) **class:isaObject**

Every class is `object` or a subclass of `object`.

```
class: Self has isaObject if_def
      Self = #object
      or Self has superStar: #object.
```

10.2.2.3.6 (co) **class:isAcyclic**

No class can be its own superclass, directly or indiectly.

```
class: Self has isAcyclic if_def
      not (Self has super..superStar: Self).
```

10.2.2.3.7 (co) **class:isParallelLowClass**

If `C` has a subclass `S`, then the metaclass (`lowClass`) of `C` cannot be a superclass of the metaclass of `S`.

```
class: Self has isParallelLowClass if_def
      if   Self has lowClass: C,
           Self has super: S,
           S has lowClass: CS,
           not C = CS
      then
           not CS has isaStar: C
      endif).
```

10.2.2.3.8 (co) **class:isaInstanceConsistency**

For any class `C` with super class `S`, `C` inherits from `C′` if `S` inherits from `C′`. Because inheritance is done along both instance of (`lowClass`) and kind of (super class) relations, the two must be constrained to be consistent.

```
class: Self has isaInstanceConsistency if_def
      forall (Self has super..lowClass: C): (
           Self has lowClassStar..isaStar: C
```

10.2.2.4 sClass

10.2.2.4.1 (op) sClass:create

```
sClass: Self has create:PnVs if_def
     Self has new: I,
     if PnVs has last:I
     then
          Self has init: PnVs
     else
          Self has init(PnVs..insertLast(I))
     endif.
```

10.2.2.4.2 (op) sClass:new

```
sClass: Self has new:I if_def
     Self has nextOid: N,
     if #N = I
     then
          Next is N + 1,
          Self has nextOid:=Next
     else
          assert not #object has instance: I
     endif,
     Self super has new: I,
     Self has directInstance:+= I.
```

10.2.2.4.3 (op) sClass:init

```
sClass: Self has init:PnVs if_def
     PnVs has last:I,
     forall (PnVs has member: (Pn:V)): ( I has Pn:=V).
```

10.2.2.4.4 (op) sClass:add

```
sClass: Self has add:PnVs if_def
     PnVs has last:I
     forall (  I has class..highCluster: Clu,
               Self has superStar..highCluster: Slu):
            ( not (Clu = Slu)),
     forall (Self has superStar:S, I has lowClass: S):
            (I has lowClass:-= S),
     I has lowClass:+= Self,
     Self has init: PnVs.
```

10.2.2.4.5 (op) sClass:delete

```
object: Self has delete if_def
     if not Self has isBeingDeleted
     then
          Self has isBeingDeleted:= true,
          forall (Self has lowClass: C):
               (C has propagateDelete: Self,
               Self has lowClass:-= C)
     endif.
```

10.2.2.4.6 (op) sClass:propagateDelete

```
sClass: Self has propagateDelete: I ifdef
     Self has nonParticipant:I,
     forall (Self has super:C, not C = #object): (C has
     propagateDelete: I)
```

10.2.2.4.7 (op) sClass:nonParticipant

```
sClass: Self has nonParticipant: I ifdef
     forall (  Self has responsibility:P,
               P has class: #participant,
               P has name: Pn,
               P has inverse: Pi,
               Pi has name: Pni,
               I has Pn:Ii,
               Is is {Isib where Ii has Pni: Isib},
               Is has count:N
               ):
          (I has Pn:-= Ii,
          NIs is N - 1,
          if Pi has cardinality: M, NIs < M or Pi has isTotal,
          NIs < 1
          then
               Ii has delete
          endif).
```

10.2.2.4.8 (op) sClass:remove

```
sClass: Self has remove: I ifdef
     forall (Self has sub: C, I has class:C): (C has remove: I),
     Self has nonParticipant:I,
     if I has lowClass: Self
     then
          I has lowClass:-=Self,
          forall (Self has super:S, not (I has class: S)):
               (I has lowClass:+=S)
     endif,
     forall (Self has highCluster:Clu, Clu has isTotal, Clu has
     super: S):
          (S has remove: I)
```

10.2.2.4.9 (op) sClass:instance

`Self` has `instance:I` if `I` is an instance of `Self`.

```
sClass: Self has instance: I ifdef
     Self has directInstance: I
     or
     Self has lowCluster..sub..instance: I.
```

10.2.2.4.10 (op) sClass:with

```
sClass: Self has with:PnVs ifdef
```

```
     PnVs has last:I,
     Self has instance:I,
     forall (PnVs has member: PnV): (I has PnV).
```

10.2.2.5 vClass

10.2.2.5.1 (op) vClass:with

```
vClass: Self has with:PnVs if_def
     PnVs has last: I,
     Self has instance:I,
     Self has responsibility: UcN,
     UcN has class: #uniquenessConstraint,
     forall (UcN has characteristic..name: Pn): (
          PnVs has member: (Pn:V)),
     Vs is [V where UcN has characteristic..name: Pn,
          PnVs has member: (Pn:V)],
     UcN has name: UCN,
     I has UCN:Vs,
     forall (  Self has superStar..responsibility: CO,
               CO has class: #constraint,
               CO has name: COn):
            (I has COn).
```

10.2.2.6 cluster

10.2.2.6.1 (co) cluster:isTotalOk

For a total cluster, every instance of the superclass must be an instance of one of the subclasses.

```
cluster: Self has isTotalOk if_def
     if   Self has isTotal
     then
          Self has super: S,
          not (S has directInstance: I)
     endif.
```

10.2.2.6.2 (co) cluster:isMutuallyExclusive

For a cluster, no instance of the superclass is an instance of more than one of the subclasses.

```
cluster: Self has isMutuallyExclusive if_def
     Self has super: S,
     forall (S has instance: I): (
          forall (Self has sub: B, I has class: B,
               Self has sub: BB, I has class: BB): (not B = BB)
          ).
```

10.2.2.7 constraint

10.2.2.7.1 (co) constraint:isTotalFunction

A constraint must be a total function.

```
constraint: Self has isTotalFunction if_def
      Self has isTotal,
      Self has isFunction.
```

10.2.2.8 uniquenessConstraint

10.2.2.8.1 `(co) uniquenessConstraint:isClassLevel`

Level must be class.

```
uniquenessConstraint: Self has isClassLevel if_def
      Self has level: class.
```

10.2.2.9 Value

10.2.2.9.1 `(op) value: delete`

Always false because a value class instance cannot be deleted.

```
      value: Self has delete if_def false
```

10.2.2.9.2 `(op) value: '<'`

For two value class instances, X and Y, X < Y if they are not equal and X<Y according to the inherited '<'.

```
      value: Self has '<': (V:value) if_def
            Self != V,
            Self super has '<': V
```

10.2.2.9.3 `(op) value: '=='`

For two value class instances, X and Y, X == Y if they have a common superclass for which all properties of a uniqueness constraint agree.

```
      value: Self has '==': (V:value) if_def
            Superclass is Self..lowClass..lub(V..lowClass)..isaStar,
            Superclass has (responsibility:UC)..class..name:
                  uniquenessConstraint,
            forall (UC has characteristic..name: Pn): (Self..Pn == V..Pn)
```

10.2.2.9.4 `(op) value: '>'`

For two value class instances, X and Y, X > Y if they are not equal and X>Y according to the inherited '>'.

```
      value: Self has '>': (V:value) if_def
            Self != V,
Self super has '>': V
```

10.3 Definition clausal form

Definition clausal form is expressively equivalent to this standard form of first order logic, i.e., any logical sentence can be transformed into an equivalent set of clauses.

10.3.1 Truth symbols

The truth symbols are `true` and `false`.

10.3.2 Constant symbols

A constant symbol denotes an individual. The constant symbols are any positive integer, any character string bounded by single quotes, or any alphanumeric character string beginning with a lowercase letter not designated to be a function or predicate symbol. In addition, the special symbols `[]` and `{}` (denoting the empty list and empty set respectively in the intended interpretation) are constants. For example, `3`, `part`, `'standard vendor'`, `[]`, and `qty_on_hand` are constants.

10.3.3 Variable symbols

A variable symbol denotes an individual, but just which individual is unknown. A variable symbol is any alphanumeric character string beginning with an uppercase letter, possibly subscripted, or tic'd. The underscore is considered an uppercase letter. An underscore standing alone, _, is an abbreviation for a variable symbol not otherwise used. For example, `PartName`, `_`, `Part_Type`, `X`, and `X'` are variables.

10.3.4 Function symbols

A function symbol denotes a function. Each function symbol has an associated arity, a positive integer specifying the number of arguments it takes.

10.3.5 Terms

A term denotes an individual. A term consists of a constant, a variable, or a function application where each argument to the function application is a term.

10.3.6 Function application

A function application is a function symbol applied to arguments, where each argument is a term. Function applications are written in prefix form, such as `f(X)` or infix form, such as `X:1`, or unary prefix form, such as `#7`. A function application denotes the result of applying a function. A function application `f(X)` denotes the result of applying the function denoted by f to the value denoted by `X`. For example if `X` is 1 and `f` is {<1,2>,<3,4>}, `f(X)` is 2, `f(2)` is undefined because 2 is not in the domain of f, and `f(W)` is undefined because `W` has no value.

10.3.7 Predicate symbols

A predicate symbol denotes a relation. Each predicate symbol has an associated arity, a positive integer specifying the number of arguments it takes.

10.3.8 Proposition

A proposition is a truth symbol, or a predicate symbol applied to arguments, where each argument is a term. Propositions are written in prefix form, such as `p(X,Y)` or infix form, such as `X = 1`. A proposition `p(X,Y)` is true if and only if the tuple <`X`,`Y`> appears in the relation denoted by the predicate symbol `p`. For example if `X` is 1, `Y` is 2, and `p` is `{<1,2>,<3,4>},` `X=1` is true, `p(X,Y)` is true, `p(2,3)` is false, and `p(Y,Z)` is neither true nor false because `Z` has no value.

10.3.9 Sentence

A sentence is a proposition, or a negated sentence, or logically connected sentences, or a quantified sentence. A sentence is true or false if and only if (iff) every proposition in the sentence is true or false. The truth value of a sentence depends on the truth value of the propositions in the sentence according to the rules for negation, logical connectives, and quantification.

The sentence `~F` is the negation of the sentence `F`. `~F` is true if `F` is false and false if `F` is true.

The truth of a sentence `H` formed by logically connecting two sentences `F` and `G` is given by Table 17.

Table 17—Example of logically connected sentences

Logical connective	Symbol	H	H is true if
Conjunction	$\wedge$	$F \wedge G$	`F` is true and `G` is true
Disjunction	$\vee$	$F \vee G$	`F` is true or `G` is true
Implication	$\rightarrow$	$F \rightarrow G$	`F` is false or `G` is true
Implication	$\leftarrow$	$G \leftarrow F$	`F` is false or `G` is true

The universal quantification of a sentence `F`, written $\forall$`(X)(F)`, is true if `F` is true for every possible assignment to the variable `X` of a value `v` from the universe of discourse.

The existential quantification of a sentence `F`, written $\exists$`(X)(F)`, is true if `F` is true for some assignment to the variable `X` of a value `v` from the universe of discourse.

Above, all occurrences of `X` within `F` are within the scope of the quantifier. A variable that is within the scope of a quantifier is bound; otherwise the variable is free. If an `X` appears within the scope of more than one quantifier, it is bound by the innermost quantifier. A sentence is closed if all variables in the sentence are bound.

A variable name used within multiple quantifiers, for example,

$$\forall(X)(\ p(X,Y)\ \wedge\ \exists(X)q(Y,X)\ )$$

names distinct variables. Using distinct names for distinct variables, for example,

$$\forall(W)(\ p(W,Y)\ \wedge\ \exists(X)q(Y,X)\ )$$

does not change the formal meaning but is usually clearer.

$\forall$`(X1,X2, …, Xn)(F)`, is an abbreviation for $\forall$`(X1)`$\forall$`(X2) …` $\forall$`(Xn)(F)`
$\exists$`(X1,X2, …, Xn)(F)`, is an abbreviation for $\exists$`(X1)`$\exists$`(X2) …` $\exists$`(Xn)(F)`
$\forall$`(*)(F)`, the universal closure of a sentence `F`, is an abbreviation for
$\forall$`(X1,X2,…,Xn)(F)`
where `X1, X2, … Xn` are all the free variables in `F` in their order of first appearance.
$\exists$`(*)(F)`, the existential closure of a sentence `F`, is an abbreviation for
$\exists$`(X1,X2,…,Xn)(F)`
where `X1, X2, … Xn` are all the free variables in `F` in their order of first appearance.
$\exists$`1(X)(F)`, the unique existential quantifier of a sentence `F`, is an abbreviation for

$\exists$(X,X')(F $\wedge$ X = X' $\wedge$ $\forall$(X)(F $\rightarrow$ X = X'))

10.3.10 Clause

A clause is a sentence that has the form

$\forall$(*)(H $\leftarrow$ B)

where H (called the head) is a proposition and B (called the body) is a sentence. Because the quantification is always universal,

H $\leftarrow$ B

is usually written, with the universal closure being understood. An equivalent form is

$\forall$(X)(H $\leftarrow$ $\exists$(Y)B)

where X are the variables in H and Y are the variables in B but not in H. Informally, the clause can be read

"H is true for values of X if B is true for those values and some values for Y."

There are two special cases—either H or B can be empty.

H	means $\forall$(*)(H). This is equivalent to H $\leftarrow$ true.
$\leftarrow$ B	means $\exists$(*)(B). This corresponds to a query that asks whether B is true.

The syntax for definition clausal form is

```
Clause →
     {Proposition } ← Sentence
     or Proposition ← {Sentence }
Proposition →
     true
     or false
     or PredicateSymbol(Term { , Term }* )
     or Term InfixPredicateSymbol Term
Term →
     ConstantSymbol
     or VariableSymbol
     or FunctionApplication
FunctionApplication →
     FunctionSymbol(Term { , Term }* )
     or Term InfixFunctionSymbol Term
     or UnaryFunctionSymbol Term
Sentence →
     Proposition
     or Negation
     or Conjunction
     or Disjunction
     or Implication
     or ExistentialQuantification
     or UniversalQuantification∧
     or (Sentence)
Negation → ~ Sentence
```

```
Conjunction → Sentence ∧ Sentence
Disjunction → Sentence ∨ Sentence
Implication → Sentence → Sentence
ExistentialQuantification → ∃(Variables)(Sentence)
UniversalQuantification → ∀(Variables)(Sentence)
Variables → Variable { , Variable }*
```

10.3.11 Closed world assumption

The classic example of the closed world assumption is a bus schedule, which states the existing connections explicitly, but all the non existent connections implicitly (by not listing them). The closed world assumption is also a common and natural assumption in databases and programs. Only what is true is recorded and anything that is not recorded or cannot be derived from what is recorded is assumed to be false.

For a set of clauses `S`, the closed world assumption is the assumption that the sentences represent all there is to be known about the relations represented by the heads of the sentences. The closed world assumption manifests itself by treating the clauses in `S` as the if sides of an implied if-and-only-if. In other words, the only way a head is true is if it is implied true by the sentences. The only-if is known as the completion of `S`, written `comp(S)`.

For example, for a set `S`

```
a(Y) ← b(Y)
a(2)
b(3)
b(4) ← c(X)
```

the completion applies the following steps:

a) Remove constants as arguments
```
a(Y) ← b(Y)
a(Y) ←  Y = 2
b(Y) ← Y = 3
b(Y) ← Y = 4 ∧ c(X)
```

b) Add clauses saying that any predicate symbol for which no head was specified cannot be true.
```
c(X) ← false
```
[This useless tautology will be reversed in step e) to something useful.]

c) Combine identical heads, renaming variables as needed.
```
a(Y) ← b(Y)∨ Y = 2
b(Y) ← Y = 3 ∨ (Y = 4 ∧c(X))
c(X) ← false
```

d) Make the existential quantification explicit for variables appearing only in the bodies.
```
a(Y) ← b(Y)∨ Y = 2
b(Y) ← (Y = 3 ∨ (Y = 4 ∧ ∃(X)c(X)))
c(X) ← false
```

e) Reverse the implications giving `comp(S)`.
```
a(Y) → (b(Y) ∨ Y = 2)
b(Y) → (Y = 3 ∨ (Y = 4 ∧ ∃(X)c(X)))
c(X) → false
```

The formalization adopts the closed world assumption. If, at the penultimate step of the formalization of a view, the theory includes a set of clauses `S`, then the final step is to add `comp(S)` to the theory. Within the clauses, an exception predicate is used to indicate a result is neither true nor false because a condition has been detected that falls outside the intended interpretation of the theory. This provides a pragmatic limitation on the closed world assumption.

10.4 Vocabulary

In this subclause, the constant, function, and predicate symbols of an IDEF$_{object}$ theory are specified. Formally, this clause merely specifies the symbols, their arity, and the syntactic form to be used. Informally, this clause also summarizes their intended meaning and use.

10.4.1 Constant symbols

The constant symbols are as follows:

`[ ]`	the empty list
`{ }`	the empty set
`“”`	the empty string
`‘’`	the empty identifier
`facts`	the initial state
`bot`	the least, bottom type, not implemented by any class
$\texttt{true}_t$	an instance of `#boolean`
$\texttt{false}_t$	an instance of `#boolean`

In addition, any constant symbol that occurs in any axiom of the theory is considered a constant in the vocabulary of the theory.

10.4.2 Function symbols

A fixed set of function symbols is used in the formalization. To help describe the intended use of the function symbols, the argument values are taken from Table 18.

Table 18—Argument values for function symbols

Argument value name	Argument value
`I`	An instance of a state class
`K`	A constant
`L`	A list
`T`	A type
`P`	A property
`Pf`	A fact property
`Pn`	The name of property `P` or `Pf`

Table 18—Argument values for function symbols *(continued)*

Argument value name	Argument value
`PO`	A property operator; one of `:`, `:=`, `:!=`, `:+=`, `:-=`. Each is a function symbol.
`S`	A state
`Si`	The input state
`So`	The output state
`V`	A value
`Value`	A variable or the result of `value(P,V,Value)`
`X, Y`	A term
`Xs`	A list of variables

The function symbols, each illustrated with an example of their application and intended meaning, are described in Table 19.

Table 19—Sample function symbols

Function symbol	Example of application and meaning
`#X`	Names an element of the UOD.
`X PO Y`	Names an element of the UOD.
`super(X)`	Names an element of the UOD.
`[X|L]`	Denotes the list that is the same as list `L` except it has one more element, `X`, as the first element.
`list(T)`	Denotes the type of a list in which all members are of type `T`.
`value(Pf,V,Value)`	Denotes a value of a value class for which `Pf` has value `V`.
`remember(I,Pf,V,S)`	Denotes a state `S'`. In `S'`, `I`'s value for `Pf` is `V`; in `S` it is not.
`forget(I,Pf,V,S)`	Denotes a state `S'`. In `S`, `I`'s value for `Pf` is `V`; in `S'` it is not.
`listof(k,Xs)`	Used in `is(CI,Si,Ws,listof(k,Xs),So)` ← `Sentence`. `Ws` is the list of `W` for which the `Sentence` is true. The `Sentence` must be read-only, i.e., `Si = So`.
`foreach(k,Xs)`	Used in `is(CI,Si,Ws,foreach(k,Xs),So)` ← `Sentence`. `Ws` is the list of `W` for which the `Sentence` is true, in order, taking an initial input state `Si` into a final, cumulative output state, `So`.
`foreach(k,Xs,Acc)`	Used in `is(CI,Si,Ws,foreach(k,Xs,Acc),So)` ← `Sentence`. `Ws` is the list of `W` for which the `Sentence` is true, in order, taking an initial input state `Si` into a final, cumulative output state, `So`, and taking the initial value of the accumulator, `Acc`, to the final value.

Each of the functions in Table 19 is total and 1 to 1. All the ranges are disjoint.

Additional function symbols are defined as a part of the included base theories.

10.4.3 Predicate symbols

A fixed set of predicate symbols is used in the formalization. This clause specifies the predicate symbols, their arity, and the syntactic form used. Each is listed with a brief summary of the intended meaning.

To help describe the intended use of the predicate symbols, the argument values are taken from Table 20.

Table 20—Argument values for predicate symbols

Argument value name	Argument value
`C`	A class
`CI`	A sender. `CI = Cs:Is`
`Cs`	The sender class
`I`	The nominal receiver
`I′`	The actual receiver. `I=I′` unless there is inheritance from a class `I′` to an instance `I`.
`IRs`	The set of `I′:R′` where `R′` is a reachable, matching responsibility and `I′` is its receiver. `I′:R` is the minimum member of `IRs.`
`Is`	The sender instance
`K`	A constant
`L`	A list
`P`	A responsibility
`Pn`	A responsibility name
`PnT`	`Pn PO T`
`PnV`	`Pn PO V`
`PO`	A property operator; one of :, :=, :!=, :+=, :-=. Each is a function symbol.
`POn`	The property operator name, one of `get`, `set`, `unset`, `add`, or `remove`
`QPnT`	The qualified property name of `R. QPnT = Cn:Pn PO T`
`R`	The selected responsibility. For an explicit responsibility, `R=#(Cn:PnT)`. For an implicit responsibility `R=#(-Cn:Pn:T)`
`RPnV`	`R:V or PnV`
`S`	A state
`Si`	The input state
`So`	The output state
`T`	The annotated argument type or a list of annotated argument types for `R`. The annotated type `T = +T′` where `T′` is the type of an input argument. `T = T′` for `T′` the type of an output argument.
`V`	The argument value or a list of argument values
`Value`	A variable or the result of `value(P,V,Value)`
`X, Y`	A term
`Xs`	A list of variables

The predicate symbols, each illustrated with an example of their application and intended meaning, are described in Table 21.

Table 21—Sample predicate symbols

Predicate symbols	Example of application and meaning
`W` $\in$ `Ws`	`W` is a member of list `Ws`.
`I` ε_S`l C`	In state `S`, `I` is a direct instance of class `C`, i.e., `C` is a `lowClass` of `I`. The reflexive, transitive closure of ε_S`l` is ε_S`l*` and the irreflexive, transitive closure of ε_S`l` is ε_S`l+`.
`I` ε_S `C`	In state `S`, `I` is an instance of class `C`, i.e., the `lowClass` of `I` is a subclass of `C`. The reflexive, transitive closure of ε_S is ε_S`*` and the irreflexive, transitive closure of ε_S is ε_S`+`.
`I` ι_S `T`	In state `S`, `I` is of type `T`, i.e., the `lowClass` of `I` is a subtype of `T`.
`T <:`$_S$ `T'`	In state `S`, `T` is a subtype of `T'`.
`X = Y`	Term `X` equals term `Y` according to the equality axiom.
`accept(S,V,T)`	In state `S`, `V` is acceptable as a type `T`.
`bind(Cs:Is, S, I, RPnV, POn, IRs, I', R, V, T)`	Relates `Cs, Is, S, I` and `RPnV` to the actual receiver `I'` and realization `R` to be used for a message `I has RPnV`, based on the inheritance search order, visibility, and argument values and types. `POn, IRs`, and `T` are also determined.
`build(V,T,V')`	`V'` matches `V` on `V`'s input values and has otherwise unused variables for the output values. `build(T,T') T'` is a list like `T` , but has otherwise unused variables as its element
`cardinalityOk (CI, S, I, POn, R, V, T)`	The total, function, and cardinality N constraints are met when the `I has POn:V` message was issued. None of these constraints are checked for a nonget property operator to an implicit realization `R`. A `cardinality N` constraint is checked only for a get property operator to a read-only responsibility `R`.
`cardinalitySolutions(CI,S, I, POn, R, V, T, Solutions, Cnt)`	The message `I has POn:V` is true for `Cnt` distinct values `V`. `Solutions` is the set of such values.
`count(L,N)`	`L` has `N` members.
`exception(R,X)`	`R` is an exceptional object outside the intended scope of the theory, with supplemental information `X`. An axiom ensures that in a *model* no exception is true.
`fact(S, I, P, V)`	In state `S`, state class instance `I` has fact property `P` value `V`.
`fact(Value,P,V )`	The `Value` of a value class instance has a fact property `P` value `V`.
`floor(S,super(Is:Cs),LC, Pl,L,C)`	In state `S`, a send to super by the sender `Cs:Is` establishes a floor `LC`, `Pl`, `L` and `C` that must be less than that for the selected responsibility `R`.
has`(CI,Si,I,RPnV,So)`	Sender `CI` in input state `Si`, sends to object `I` for a responsibility `R` (`RPnV` is `R:V`) or a resonsibility named `Pn` in one of the property operator forms (`RPnV` is `Pn:V` or `Pn:=V` or `Pn:!=V` or `Pn:+=V` or `Pn:-=V`) with a value of `V`, and the output state is `So`. Every message is in this form.
has`(CI,Si,I,P,V,So)`	For class instance `CI` in input state `Si`, object `I` has a responsibility `P` value of `V` and the output state is `So`. The head of every realization is in this form.
is`(CI,Si,X,Expr,So)`	For sender `CI` in input state `Si`, term `X` is the oid of the object denoted by the term `Expr`, and the output state is `So`. An `Expr` is either a `Literal` or an arithmetic expression.
`isType(S,T)`	In state `S`, `T` is a type.

Table 21—Sample predicate symbols *(continued)*

Predicate symbols	Example of application and meaning
C isa_S C′	In state S, class C is a subclass of class C'. isa_S* is the reflexive transitive closure and isa_S+ is the irreflexive transitive closure.
isList(L)	L is a list.
isState(S)	S is a state.
lessThan(S, [LC,Pl,L,C,T], [LC′,Pl′,L′,C′,T′])	The inheritance order is ascending on LowClass, Plicity, Level, Class, Type.
lub(S,List,T)	In state S, every member of List is type T, and there is no distinct subtype of T for which this is true.
match(Cs:Is, S, I,QPnT,V,R,Pn,L,Pl,T)	R is visible to Cs and Is, matches QPnT in name and property operator, has type T that accepts V, is at level L, and has plicity Pl (implicit or explicit).
minimum(S,IRs,[R,I′, LC,Pl,L,C,T])	For any responsibility R′ that matches, is reachable, and is above the floor, R = R′ or R is less than R′.
noDup(L)	L has no duplicates.
parsePnV(PnV, Pn, POn, V, QPnT)	Relates PnV to Pn, POn, V, and QPnT.
parseRV(R: V, Pn, POn, T, V, QPnT, C)	Relates R:V to R's Pn, POn, T, QPnT and C.
post(R,R′)	For responsibility R, the post-condition is R′.
postOk(Si, IRs, V, T, So)	In input state Si and output state So, the post-condition for I has R:V is met.
pre(R,R′)	For responsibility R, the pre-condition is R′.
preOk(Si, IRs, V, T)	In input state Si, the pre-condition for I has R:V is met.
reach(S,I,LC,C,I′)	I can reach class C along an inheritance path. I ε_Sl* LC and LC isa_S* C.
theoryRep(Cn,Self,Rep)	For base type Cn, instance Self, the theory's representation is Rep.
visible(Cs:Is,S,I,R)	In state S, the sender Cs:Is can see R for a message to I for R.

Additional predicate symbols are defined as a part of the included base theories.

10.5 Axioms of base theories

An IDEF$_{object}$ theory is built upon an equality axiom and theories for lists, pairs, identifiers, characters, strings, integers, and reals.

10.5.1 Equality axiom

The equality axiom,

```
    X = X
```

means that two terms are equal if and only if they have the same function symbol, the same number of arguments, and all corresponding arguments are equal.

The ranges of all the functions in the vocabulary are distinct because of the equality axiom. Since the function symbols are distinct, so must be the ranges of the functions in any *model*.

For the same reason, all constants in the vocabulary are distinct.

10.5.2 List

The vocabulary consists of the following:

`[]`	a constant.
`[_|_]`	an arity 2 function symbol
`isList`	an arity 1 predicate symbol
`∈`	an arity 2 infix predicate symbol
`count`	an arity 2 predicate symbol
`noDup`	an arity 1 predicate symbol

The axioms are as follows:

```
 isList([])
 isList([X|Xs]) ← isList(Xs)

 X ∈ L ← isList(L) ∧ L = [X|_]
 X ∈ L ← isList(L) ∧ L = [_|Xs] ∧ X ∈ Xs

 count(L,N) ← L = [] ∧ N = 0
 count(L,N) ← isList(L) ∧ L = [_|Xs] ∧ count(Xs,M) ∧ N is 1 + M

 noDup(L) ← L = []
 noDup(L) ← isList(L) ∧ L = [X|Xs] ∧ ~(X ∈ Xs) ∧ noDup(Xs)
```

The notation `[X1, X2, …, Xn]` is an abbreviation for `[X1 | [ X2 | … | Xn | [ ] ] ]`.

10.5.3 Pair

The vocabulary consist of the following:

`isPair`	an arity 1 predicate symbol

The axioms are as follows:

```
 isPair(X:Y)
```

10.5.4 Character

The vocabulary consist of the following:

`isCharacter`	an arity 1 predicate symbol

The notation `a` is an abbreviation for `‘a’`.

10.5.5 Identifier

The vocabulary consist of the following:

`‘’`	a constant.
`prefixIdentifier`	an arity 2 function symbol
`isIdentifier`	an arity 1 predicate symbol

The axioms are as follows:

```
isIdentifier(‘’)
isIdentifier(C) ← isCharacter(C)
isIdentifier(prefixIdentifier(C,I)) ←
     isCharacter(C),isIdentifier(I), ~(I = ‘’)
```

The notation `abc` is an abbreviation for `‘abc’`, which is an abbreviation for

```
prefixIdentifier(‘a’,prefixIdentifier(‘b’,‘c’)).
```

10.5.6 String

The vocabulary consist of the following:

`“”`	a constant.
`PrefixString`	an arity 2 function symbol
`isString`	an arity 1 predicate symbol

The axioms are as follows:

```
isString(“”)
isString(prefixString(C,S)) ← isCharacter(C),isString(S)
```

The notation `“abc”` is an abbreviation for

```
prefixString(a,prefixString(b,prefixString(c, “”))).
```

10.5.7 Integer

The vocabulary consist of the following:

`0`	a constant.
`iPlus`	an arity 3 predicate symbol
`iMinus`	an arity 3 predicate symbol
`iTimes`	an arity 3 predicate symbol

`iDivideby`	an arity 3 predicate symbol
`iExp`	an arity 3 predicate symbol
`isInteger`	an arity 1 predicate symbol
`asReal`	an arity 2 predicate symbol

The axioms are assumed.

10.5.8 Real

The vocabulary consist of the following:

`0.0`	a constant.
`rPlus`	an arity 3 predicate symbol
`rMinus`	an arity 3 predicate symbol
`rTimes`	an arity 3 predicate symbol
`rDivideby`	an arity 3 predicate symbol
`rExp`	an arity 3 predicate symbol
`isReal`	an arity 1 predicate symbol
`asInteger`	an arity 2 predicate symbol

The axioms are assumed.

10.6 Rewriting an IDEF$_{object}$ view to definition clausal form

An IDEF$_{object}$ view is translated into a theory in the definition clausal form language described in 10.3 and 10.4. The translation is done in three steps.

10.6.1 Declare instances of the metamodel for a view

In order to generate the theory for a view, the graphics are restated as RCL declarations. The declarations declare that the metamodel contains the view being formalized. A declaration declares the value of a class instance responsibility.

```
        Cn: I has Pn: V.
```

This statement can be read as class `Cn`’s instance `I` has a property `Pn` value of `V`.

Fully qualified names are used for `Vn`, `Cn`, and `Tn`.

10.6.1.1 View

For the metamodel view, declare

```
        view:#metamodel has lowClass: #view.
```

For any other view Vn with parent view V, declare

```
    view:#Vn has lowClass: #view.
    view:#Vn has name: Vn.
    view:#Vn has parent: V.
```

10.6.1.2 Class

For each state class `Cn` in the view `Vn`, declare

```
sClass:#Cn has lowClass: #sClass.
sClass:#Cn has view: #Vn.
sClass:#Cn has name: Cn.
```

For each value class `Cn` in the view `Vn`, declare

```
vClass:#Cn has lowClass: #vClass.
vClass:#Cn has view: #Vn.
vClass:#Cn has name: Cn.
```

For each parametric value class `Cn(T1,T2,…,Tn)` in the view `Vn`, declare

```
parametricVClass:#(Cn:[T1,T2,…,Tn]) has lowClass:
#parametricVClass.
parametricVClass:#(Cn:[T1,T2,…,Tn]) has view: #Vn.
parametricVClass:#(Cn:[T1,T2,…,Tn]) has name: (Cn:[T1,T2,…,Tn]).
```

10.6.1.3 Generalization

For each class `Cn` that is the superclass for a cluster, where each cluster under a superclass is assigned a constant K unique within the superclass, declare

```
cluster:#(Cn:cluster:K) has lowClass: #cluster.
cluster:#(Cn:cluster:K) has super: #Cn.
```

If the cluster is total, declare

```
cluster:#(Cn:cluster:K) has isTotal.
```

For each subclass `Cn’`, declare

```
cluster:#(Cn:cluster:K) has sub: #Cn’.
```

10.6.1.4 Relationship

For each relationship between two classes,

a) Arbitrarily but consistently, designate one class the parent and the other the child.
Let
`PCn` = the fully qualified name of the parent class.
`PCsn` = the simple, unqualified name of the parent class.
`CCn` = the fully qualified name of the child class.
`CCsn` = the simple, unqualified name of the child class.
`PRn` = the role name of the parent class. If no role name is specified, `PRn` = `PCsn`.
`CRn` = the role name of the child class. If no role name is specified, `CRn = CCsn.`
b) Declare the inverses.

```
participant:#(-PCn:CRn:CCn) has inverse: #(-CCn:PRn:PCn).
participant:#(-CCn:PRn:PCn) has inverse: #(-PCn:CRn:CCn).
```

c) Specify the participant properties.
If the parent has no participant property with a get property operator for the child, then apply the participant declaration rules to the following as though it had been specified in the parent.

```
     CRn: #PCn Completeness Multiplicity Cardinality
```

where
Completeness = `optional` unless no dot or solid dot `P` or solid dot `N > 0`
Multiplicity = `single valued` if no dot or hollow dot or solid dot `Z` or solid dot `N < 2`
Cardinality = `cardinalityN: N` if solid dot `N > 1`
If the child has no participant property with a get property operator for the parent, then apply the participant declaration rules to the following as though it had been specified in the child.

```
     PRn: #CCn Completeness Multiplicity Cardinality
```

where
Completeness = optional unless no dot or solid dot `P` or solid dot `N > 0`
Multiplicity = `single valued` if no dot or hollow dot or solid dot `Z` or solid dot `N < 2`
Cardinality = `cardinalityN: N` if solid dot `N > 1`

10.6.1.5 Participant

For each class `Cn`, for each participant property named `Pn`, with property operator `PO`, and inverse class `Cn'`, declare

```
Let  OID = #(Cn:Pn PO Cn')  if the property is not suffixed (in)
     OID = #(Cn:Pn PO +Cn')  if the property is suffixed (in)
participant:OID has lowClass: #participant.
participant:OID has level: instance.
participant:OID has plicity: explicit.
```

If the specification is `Pn` or `Pn: _` (i.e., the get property operator) and the participant is a fact, declare the implicit participant instance as follows:

```
Let  OID = #(- Cn:Pn:Cn)
participant:OID has lowClass: #participant.
participant:OID has level: instance.
participant:OID has isFact.
participant:OID has plicity: implicit.
```

10.6.1.6 Attribute

For each class `Cn`, for each attribute named `Pn`, with property operator `PO`, level `L` (instance or class), and type `Tn`, declare

```
Let  OID = #(Cn:Pn: boolean)  if there is no argument
     OID = #(Cn:Pn PO Tn) if the property is not suffixed (in)
     OID = #(Cn:Pn PO +Tn)  if the property is suffixed (in)
attribute:OID has lowClass: #attribute.
attribute:OID has level: L.
attribute:OID has plicity: explicit.
```

If the specification is `Pn` or `Pn: _` (i.e., the get property operator) and the attribute is a fact, declare the implicit attribute instance as follows:

```
Let  OID = #(- Cn:Pn:Tn)
```

```
    attribute:OID has lowClass: #attribute.
    attribute:OID has level: L.
    attribute:OID has isFact.
    attribute:OID has plicity: implicit.
```

10.6.1.7 Operation

For each class `Cn`, for each operation named `Pn`, with property operator `PO`, level `L` (instance or class), with `N` arguments of type `Tni`, where `Tni = object` if no type is specified, and `Ti = +Tni` if (in) is specified, otherwise `Ti = Tni`, declare

```
    Let QPnT = Cn:Pn : boolean           if N = 0
        QPnT = Cn:Pn PO T1               if N = 1
        QPnT = Cn:Pn PO [T1, T2, …, TN]  if N > 1
        OID = #QPnT
    Operation:OID has lowClass: #operation.
    operation:OID has level: L.
    operation:OID has plicity: explicit.
```

For each argument `K` of type `Tn`, `K = 1` to `N`, declare

```
        argument:#(QPnT:K) has position:K.
        argument:#(QPnT:K) has type:#Tn.
```

If updatable, declare

```
        argument:#(QPnT:K) has isUpdatable.
```

If an input, declare

```
    argument:#(QPnT:K) has isInput.
```

10.6.1.8 Constraint

For each class `Cn`, for each constraint named `Pn`, level `L` (instance or class), declare

```
    constraint:#(Cn:Pn:boolean) has lowClass: #constraint.
    constraint:#(Cn:Pn:boolean) has level: L.
    constraint:#(Cn:Pn:boolean) has plicity: explicit.
```

10.6.1.9 Uniqueness constraint

For each class `Cn`, for each uniqueness constraint `N` with `M` properties, declare

```
    Let QPnT = Cn:ucN: T1                if M = 1
        QPnT = Cn:ucN: [T1, T2, …, TM]   if M > 1
        OID = #QPnT
    uniquenessConstraint: OID has lowClass: #uniquenessConstraint.
    uniquenessConstraint: OID has class: #Cn.
    uniquenessConstraint: OID has name: ucN.
    uniquenessConstraint: OID has level: class.
    uniquenessConstraint: OID has plicity: explicit.
```

For each property `K` with type `T`, `K = 1` to `M`, in order of appearance in the graphic,

```
        Tk = +T
```

10.6.1.10 Responsibility

For each responsibility, with oid `OID` as determined above, with a class named `Cn` and a property named `Pn`, declare

```
responsibility:OID has class: #Cn.
responsibility:OID has name: Pn.
responsibility:OID has propertyOperator: POn.
```

where `POn` is

`get` if the property operator is `:`
`set` if the property operator is `:=`
`unset` if the property operator is `:!=`
`add` if the property operator is `:+=`
`remove` if the property operator is `:-=`

```
responsibility:OID has visibility: Vis.
```

where `Vis` is

`public` if the property is unannotated
`protected` if the property is annotated by "|"
`private` if the property is annotated "||"

If the responsibility is a fact, declare

```
responsibility:OID has isFact.
```

If there is not a suffix `multi valued`, declare

```
responsibility:OID has isFunction.
```

If there is not a suffix `optional`, declare

```
responsibility:OID has isTotal.
```

If there is a read-only suffix or the responsibility is a constraint, or an attribute or participant property with a get property operator, declare

```
responsibility: OID has isReadOnly.
```

If there is a constant suffix, or the responsibility is a uniqueness constraint, declare

```
responsibility: OID has isConstant.
```

If there is a cardinality N suffix, declare

```
responsibility: OID has cardinalityN: N.
```

10.6.2 Rewrite RCL to definition form clauses

RCL is rewritten to definition clausal form by rewrite rules that translate RCL into definition form clauses. There are three sets of rewrite rules.

a) The RCL used in queries and realizations is rewritten to a syntactically simpler but equivalent form by the mapping

   $\mathcal{M}_{\mathcal{S}}$: `RCL` → `Rcl`

b) The RCL queries, declarations, and realizations are rewritten to an intermediate set of definition form clauses by the mappings

   $\mathcal{M}_{\mathcal{Q}}$: `QueryRCL` → `Clause`

   $\mathcal{M}_{\mathcal{D}}$: `DeclarationRCL` → `Clause`

   $\mathcal{M}$: `RealizationRCL` → `Clause`

   The clauses produced by this mapping use predicate symbols, such as an arity 3 has, that are not part of the vocabulary of an IDEF$_{object}$ theory. The $\mathcal{V}$ mapping adds the remaining arguments.

c) The definition form clauses are rewritten to add arguments for the sender and the input and output states to the propositions.

   $\mathcal{V}$: `Sender × State × Clause × State` → `Clause`

   The clauses produced by the $\mathcal{V}$ mapping use only predicate symbols that are part of the vocabulary of an IDEF$_{object}$ theory.

A rewrite rule of the form

LHS => RHS

    $\&$ `XXX`

means to replace the LHS by the RHS and also add `XXX` to the set to which the rewrite rules are being applied.

The symbols in the rules are the syntactic symbols in the RCL syntax, augmented by abbreviations (see Table 22).

Table 22—Symbols in rewrite rules

Symbol	Meaning
`Cn`	Class name
`Pn`	Responsibility name
`OID`	`StateClassOid`
`K`	`SimpleObject`
`Self, Var, CnVar, Ws`	`Variables`
`Type`	`TypeLiteral`
`Arg`	`Argument`
`Args`	`Arguments`
`PO`	`PropertyOperator`

Table 22—Symbols in rewrite rules *(continued)*

Symbol	Meaning
`I, V`	`Objects`
`PV`	`ResponsibilityValue`
`PnV`	`Pn:V` or `Pn:=V` or `Pn:!=V` or `Pn:+=V` or `Pn:-=V`
`B, F, G,` and `H`	`Sentences`
`Head`	`Cn : Variable has Pn { PO Args ]`
`W,X,Y`	Meta symbols denoting lists of variables
`X`	List of `Variables` free in the LHS
`p`	Predicate symbol
`k`	Constant symbol not otherwise used
$\texttt{true}_t$	Term (not the logical constant `true`)

If a construct matches the LHS of multiple rewrite rules, the first matching rule in the list is used.

10.6.2.1 Declaration RCL

Declaration RCL is always stated in the context of a view. A declaration `Cn:OID has Pn:K` means that `Cn:OID` has `Pn:K` is a fact. An axiom to that effect is added to the theory for the view.

$$\mathcal{M}_{\mathcal{D}}(\texttt{Cn:OID has Pn:K}) \quad \texttt{=> fact(facts, OID, \#(Cn':Pn:T), K)}$$

where `Cn′:Pn:T` is the qualified property name of the direct or inherited property for `Pn`.

10.6.2.2 Query RCL

$\mathcal{M}_{Q}(\texttt{Sentence})\texttt{=>} \leftarrow \mathcal{M}(\texttt{Sentence})$

10.6.2.3 Realization RCL

The renaming rule for quantification says that, for example, `∃(W)(F)` is exactly equivalent to `∃(Z)(F′)` where `F′` is `F` with all free occurrences of `W` replaced with `Z`. The formalization of `exists`, `forall`, `not`, and `if` assumes that this rule has been applied to uniquely name the quantification variables.

A variable `X` in a sentence `F` is a *quantification variable* if

`X` appears free in `F`,
`X` does not appear in the head,
`X` does not appear free in the body less `F`.

A variable appears free in a sentence if it is not bound by an `exists` or `forall` within the sentence.

Every variable is quantified by applying, in order, these rules.

a) For an RCL `exists F`, the quantification is `∃(W)(F)`.

For an RCL `forall F: G`, the quantification is `∀(W)((F → ∃(Y)(G))`.

b) For an RCL `not F`, the quantification is `~∃(W)(F)`.
For an RCL `if F then G`, the quantification is `∃(W)(F) → ∃1(W)(F) ∧ ∃(W)(F ∧ G)`.

c) For an RCL `Head if`$_{def}$ `F`, the quantification is `∀(Z)(Head ← ∃(W)(F)`.

`W` represents the quantification variables in `F`. `Y` represents the quantification variables in `G`. `Z` represents the variables in the head. If there are no such variables, there is no quantification.

If a variable appearance is within multiple `exists`, `forall`, `not`, or `if`, it is quantified by the innermost for which it qualifies.

Realization RCL is mapped to clausal form according to rewrite rules below.

The first group of rules rewrite RCL to simpler RCL.

If the left side contains `had` instead of `has` or `was` instead of `is`, then the right side contains `had` in place of `has` and `was` in place of `is`.

```
𝓜𝓢(Cn: Self has Pn)           => Cn: Self Pn: (true_t: boolean)
𝓜𝓢(Self { super } has Pn)     => Self { super } has Pn: true_t
𝓜𝓢(Object.. { PathExpr } ResponsibilityValue) =>
     𝓜𝓢(Object has { PathExpr } ResponsibilityValue)
𝓜𝓢(I has { PathExpr } PropertyExpr .. ResponsibilityValue) =>
     𝓜𝓢(I has { PathExpr } PropertyExpr: V),
     V has ResponsibilityValue
𝓜𝓢(I has { PathExpr } (PropertyExpr: SimpleObject) ..
ResponsibilityValue) =>
     𝓜𝓢(I has { PathExpr } PropertyExpr: SimpleObject,
     SimpleObject has ResponsibilityValue
𝓜𝓢(I has Pn(Objects): V) => 𝓜𝓢(I has Pn(Objects,V))
𝓜𝓢(I has Pn(Object1, Object2, …, Objectn ) )=>
     𝓜𝓢(V1 is Object1),
     𝓜𝓢(V2 is Object2),
     ...
     𝓜𝓢(Vn is Objectn),
     I has Pn: [V1, V2, …, Vn ]
𝓜𝓢(I has Pn(Object) )=>
     𝓜𝓢(V is Object),
     I has PnV
𝓜𝓢(Object { super } has { PathExpr } ResponsibilityValue ) =>
     I is Object,
     𝓜𝓢(I { super } has { PathExpr } ResponsibilityValue )
𝓜𝓢(Object1 RelOp Object2) =>
```

    I1 is Object1,
    $\mathcal{M}_{\mathcal{S}}$(I1 has RelOp(I2))

$\mathcal{M}_{\mathcal{S}}$(V is Object ..*{* PathExpr *}* PropertyExpr) =>

    $\mathcal{M}_{\mathcal{S}}$(Object has ..*{* PathExpr *}* PropertyExpr:V)

$\mathcal{M}_{\mathcal{S}}$(variable Var: Type is Object)=> variable Var: Type, $\mathcal{M}_{\mathcal{S}}$(Var is Object)

$\mathcal{M}_{\mathcal{S}}$(Var: Type is Object)   => variable Var: Type, $\mathcal{M}_{\mathcal{S}}$(Var is Object)

$\mathcal{M}_{\mathcal{S}}$(Var is SimpleObject)   => Var = SimpleObject

$\mathcal{M}_{\mathcal{S}}$(X is Object1 : Object2)   => V1 is Object1, V2 is Object2, X = V1 : V2

$\mathcal{M}_{\mathcal{S}}$(X is list(Object)   => X is [Y where Y is $\mathcal{M}_{\mathcal{S}}$(Object)]

$\mathcal{M}_{\mathcal{S}}$(X is set(Object)   => $\mathcal{M}_{\mathcal{S}}$(Y is list(Object), X is set(list:Y))

$\mathcal{M}_{\mathcal{S}}$(X is bag(Object)   => $\mathcal{M}_{\mathcal{S}}$(Y is list(Object), X is bag(list:Y))

$\mathcal{M}_{\mathcal{S}}$(X is [Object])   => $\mathcal{M}_{\mathcal{S}}$(X is list(Object))

$\mathcal{M}_{\mathcal{S}}$(X is { Object })   => $\mathcal{M}_{\mathcal{S}}$(X is set(Object))

$\mathcal{M}_{\mathcal{S}}$(X is { })   => $\mathcal{M}_{\mathcal{S}}$(X is set(list:[]))

$\mathcal{M}_{\mathcal{S}}$(I is {Object | Set }   )=> $\mathcal{M}_{\mathcal{S}}$(X is Object, Xs is Set, I is set(list:[X|Xs]))

$\mathcal{M}_{\mathcal{S}}$(I is {Object | List })   => $\mathcal{M}_{\mathcal{S}}$(X is Object, Xs is List, I is [X|Xs])

$\mathcal{M}_{\mathcal{S}}$(I is Cn(PV1,PV2,…,PVn))   =>

    I is Cn with (PV1,PV2,…,PVn)

$\mathcal{M}_{\mathcal{S}}$(I is Cn with (PV1,PV2,…,PVn)=>

    $\mathcal{M}_{\mathcal{S}}$(v,1,PV1),
    $\mathcal{M}_{\mathcal{S}}$(v,2,PV2),
    ...
    $\mathcal{M}_{\mathcal{S}}$(v,n,PVn),

#Cn has with:[$\mathcal{M}_{\mathcal{S}}$(a,1,PV1) $\mathcal{M}_{\mathcal{S}}$(a,2,PV2),…, $\mathcal{M}_{\mathcal{S}}$(a,n,PVn),I]

$\mathcal{M}_{\mathcal{S}}$(v,J,Pn(Object1,Object2,… ,Objectm)) =>

    $\mathcal{M}_{\mathcal{S}}$(VJ1 is Object1),
    $\mathcal{M}_{\mathcal{S}}$(VJ2 is Object2),
    ...
    $\mathcal{M}_{\mathcal{S}}$(VJm is Objectm),

$\mathcal{M}_{\mathcal{S}}$(a,J,Pn(Object1,Object2,… ,Objectm)) => Pn:[VJ1,VJ2,…,VJm]

$\mathcal{M}_{\mathcal{S}}$(a,J,Pn(Object1)) => Pn:VJ1

```
$\mathcal{M}_{\mathcal{S}}$(a,J,Pn:V) => Pn:V
$\mathcal{M}_{\mathcal{S}}$(a,J,Pn) => Pn:true$_t$
$\mathcal{M}_{\mathcal{S}}$(X is set( Objects   => $\mathcal{M}_{\mathcal{S}}$(Y is list(Objects), X is set(list:Y))
$\mathcal{M}_{\mathcal{S}}$(X is set( Objects   => $\mathcal{M}_{\mathcal{S}}$(Y is list(Objects), X is set(list:Y))
$\mathcal{M}_{\mathcal{S}}$(X is bag( Objects   => $\mathcal{M}_{\mathcal{S}}$(Y is list(Objects), X is bag(list:Y))
$\mathcal{M}_{\mathcal{S}}$(X is [ Objects ]    => $\mathcal{M}_{\mathcal{S}}$(X is list(Objects))
$\mathcal{M}_{\mathcal{S}}$(X is { Objects }    => $\mathcal{M}_{\mathcal{S}}$(X is set(Objects))
$\mathcal{M}_{\mathcal{S}}$(X is list( Object1, Object2,…, Objectn )=>
     $\mathcal{M}_{\mathcal{S}}$(Y1 is Object1),
     $\mathcal{M}_{\mathcal{S}}$(Y2 is Object2),
     ...
$\mathcal{M}_{\mathcal{S}}$(Yn is Objectn),
X is [ Y1, Y2,…, Yn ]
$\mathcal{M}_{\mathcal{S}}$(V is UnaryOp Object) =>
     $\mathcal{M}$(I is Object),
     $\mathcal{M}$(I has ‘UnaryOp’:V)
$\mathcal{M}_{\mathcal{S}}$(V is Object1 BinaryOp Object2) =>
     $\mathcal{M}$(I1 is Object1),
     $\mathcal{M}$(I2 is Object2),
     $\mathcal{M}$(I1 has ‘BinaryOp’:[I2,V]
$\mathcal{M}_{\mathcal{S}}$(V is Object where Sentence) =>
     $\mathcal{M}$(Sentence),
     $\mathcal{M}$(V is Object)
$\mathcal{M}_{\mathcal{S}}$(if F then G else H endif) =>
     $\mathcal{M}_{\mathcal{S}}$(if F then G endif $\wedge$ if not F then H endif))
$\mathcal{M}_{\mathcal{S}}$(if F then G endif) => if $\mathcal{M}_{\mathcal{S}}$(F) then $\mathcal{M}_{\mathcal{S}}$(G) endif
$\mathcal{M}_{\mathcal{S}}$(F , G) => $\mathcal{M}_{\mathcal{S}}$(F) , $\mathcal{M}_{\mathcal{S}}$(G)
$\mathcal{M}_{\mathcal{S}}$(F or G) => $\mathcal{M}_{\mathcal{S}}$(F) or $\mathcal{M}_{\mathcal{S}}$(G)
$\mathcal{M}_{\mathcal{S}}$(not F) => not $\mathcal{M}_{\mathcal{S}}$(F)
$\mathcal{M}_{\mathcal{S}}$(forall F: G) => forall $\mathcal{M}_{\mathcal{S}}$(F) : $\mathcal{M}_{\mathcal{S}}$(G)
$\mathcal{M}_{\mathcal{S}}$(for Accs all F: G    => for $\mathcal{M}_{\mathcal{S}}$(Accs) all $\mathcal{M}_{\mathcal{S}}$(F) : $\mathcal{M}_{\mathcal{S}}$(G)
$\mathcal{M}_{\mathcal{S}}$(Head if$_{def}$ B)          => $\mathcal{M}_{\mathcal{S}}$(Head if$_{def}$ B, post true) B has no post
$\mathcal{M}_{\mathcal{S}}$(Head if$_{def}$ B)          => $\mathcal{M}_{\mathcal{S}}$(Head if$_{def}$ pre true, B)  B has no pre
$\mathcal{M}_{\mathcal{S}}$(Head if$_{def}$ B)          => $\mathcal{M}_{\mathcal{S}}$(Head if$_{def}$ pre true, B, post true)  B has
no pre no post
```

```
𝓜_𝓢(Cn: Self has Pn PO Args if_def
        pre PreSentence1,
        ...
        pre PreSentenceN,
        Sentence,
        post PostSentence1,
        ...
        post PostSentenceM)      =>
                                 Cn: Self has Pn PO Args if_def Sentence
                                 &
                                 Cn: Self has (-Pn) PO Args if_def
                                       PreSentence1
                                       or ...
                                       or PreSentenceN
                                 &
                                 Cn: Self has (+Pn) PO Args if_def
                                       PostSentence1,
                                       …,
                                 PostSentenceM
```

The next group of rewrite rules rewrite RCL to clausal form. Here, `I` and `V` are `SimpleObjects`.

```
𝓜(Head if_def B)              => 𝓜(Head) ← 𝓜(Sentence)
𝓜(Cn:Self has Pn PO Args)     => has(#Cn:Self,#(Cn:Pn PO 𝓜(Types)),
Variables)
     & 𝓜_𝓓(responsibility: #(Cn:Pn PO Types) has isRealized)
𝓜(variable Variable : Type)  => Variable ι 𝓜(Type)
𝓜(+Type)                      => + 𝓜(Type)
𝓜(class_qname)                => #class_qname
𝓜(class_qname:[ Types])       => #(class_qname:𝓜([Types]))
𝓜([Type1,Type2,…Typen])       => [𝓜(Type1), 𝓜(Type2),… 𝓜(Typen)])
𝓜(I has Pn PO V)              => has(I, Pn PO V)
𝓜(I had Pn PO V)              => had(I, Pn PO V)
𝓜(Self super has Pn PO V)     => has(super(Self:#Cn), Pn PO V)
𝓜(Self super had Pn PO V)     => had(super(Self:#Cn), Pn PO V)
𝓜(not F)                      => ~∃(W)( 𝓜(F))
𝓜(F, G)                       => 𝓜(F) ∧ 𝓜(G)
𝓜(F or G)                     => 𝓜(F) ∨ 𝓜(G)
𝓜(if F then G endif)          =>
    ~∃(W)( 𝓜(F)) ∨ (∃1(W)( 𝓜(F)) ∧ ∃(Y)( 𝓜(F ∧ G)))
𝓜(exists F)                   => ∃(W)( 𝓜( F))
𝓜(forall F: G)                 =>
  𝓜(Ws is list(W where F )) ∧ is(Ws,foreach(k,X))
  & is(Ws,foreach(k,X)) ←
```

```
            Ws = [W|Rest] → 𝓜(G) ∧ is(Rest,foreach(k,X))
𝓜(for Acc all F: G)          =>
     𝓜(Ws is list(W where F )) ∧ is(Ws,foreach(k, X, Acc))
     & is([W|Rest],foreach(k, X, Acc')) ←
            𝓜(
            𝓜_S(Acc is accumulator(initial(Acc'..previous),final: _)),
             G,
            𝓜_S(Acc'' is
            accumulator(initial(Acc..current),final(Acc'..final))
            ) ∧
            is(Rest,foreach(k,X,Acc''))

𝓜(Ws is [SimpleObject where F ]) => is(Ws,listof(k,X))
          & is(Ws,listof(k,X)) ←
            isList(Ws) ∧
            ∀(W)( 𝓜(F) → SimpleObject ∈ Ws ) ∧
            ∀(W)( SimpleObject ∈ Ws → 𝓜(F) )

𝓜(assert F)          => 𝓜( not F) → exception('assertion failure: F')
     𝓜(Other)        => Other
```

In the mapping rules above

— Arguments are mapped to types and variables according to the syntax for arguments (see Table 23).

Table 23—Mapping of arguments to types and variables

Arguments	Types	Variables
`Var`	`object`	`Var`
`Var:Type`	`Type`	`Var`
`Var (in)`	`+object`	`Var`
`Var:Type (in)`	`+Type`	`Var`
`[Arg1,Arg2,…,Argn]`	`[Type1, Type2,…, Typen]`	`[Var1,Var2,…,Varn]`

— *W* represents the quantification variables in `F`. If there are no such variables, omit the quantification.
— *Y* represents the quantification variables in `G`. If there are no such variables, omit the quantification.
— The properties `Pn1` through `Pnn` must constitute a uniqueness constraint `ucN` for a value class `Cn` where `V'j = Vi` if `Pni` is the `j`th component of `ucN` of class `Cn`.
— `Self super` can be used only within a realization. `Cn` is the class for which the realization is specified.
— `had` and `was` can be used only within a post sentence.

10.6.3 Add arguments for sender and state

Update is formally defined by considering an update operation to map the state of the view to a new state. The state of a view is its set of facts. The initial state is the set of facts declared for the view. Each proposition relates an input state to an output state. To carry this out, `Si` and `So` are added as arguments to the proposition and the proposition becomes

```
has(Si, I, C:=P, V', So).
```

If a proposition fails, then the state is unchanged.

It is assumed that the intended effect of the update mappings is cumulative, in the order the sentences are written.

An RCL sentence consists of logically connected propositions, such as the conjunction

```
I has Pn: V, I' has Pn':V'.
```

If there are no state changes, then the order of evaluation does not matter. But if there are state changes, then it does matter. The formalization assumes that the order of evaluation, if it should matter, is the order in which the propositions are written left to right. The sentence above is mapped to

```
        ∃ (S)( has(Si,I,Pn:V,S) ∧ has(S,I',Pn':V',So) ).
```

The output state `S` of the left conjunct is the input state to the right conjunct. This is a purely declarative conjunction, exactly equivalent to

```
        ∃ (S)( has(S,I',Pn':V',So) ∧ has(Si,I,Pn:V,S) ).
```

Either conjunction has the same effect as solving the RCL conjunction in the order written.

10.6.3.1 Query

A query sentence is true if it is logically implied by the view and its instances. The initial state for a query is whatever the declarations for the metamodel and the view declared. The declared facts are the input state for the query.

```
𝒱( Si, ← B, So) => 𝒱(facts,B,So)
```

10.6.3.2 Realization

The update mapping is defined for each syntactic form of clause, proposition, and sentence. Universally quantified sentences will raise an exception if any updates occur.

```
𝒱(Si, has(C:I,P,V) ← B,       So)   => has(C:I,Si, I,P,V, So) ←
𝒱(C:I,Si, B, So)
𝒱(CI,Si, has(I,P),            So)   => has(CI,Si, I,P, So)
𝒱(CI,Si, is(V,Ex) ← B,        So)   => is(CI,Si, V,Ex, So) ← 𝒱(CI,Si, B,
So)
𝒱(CI,Si, is(V,Ex)             So)   => is(CI,Si, V,Ex, So)
𝒱(CI,Si, V ι T                So)   => V ι_Si T ∧ Si = So
𝒱(CI,Si, ~F,                  So)   => ~∃(S)( 𝒱(CI,Si, F, S)) ∧ Si = So
```

```
𝒰(CI,Si, F ∧ G,              So)   => 𝒰(CI,Si, F, S) ∧ 𝒰(CI,S, G, So)
𝒰(CI,Si, F ∨ G,              So)   => 𝒰(CI,Si, F, So) ∨ 𝒰(CI,Si, G, So)
𝒰(CI,Si, ∃(Variables)(F),So)       => ∃(Variables)(𝒰(CI,Si, F , So) )
𝒰(CI,Si, ∀(Variables)(F),So)       =>
      ∀(Variables)(𝒰(CI,Si, F , So) ∧
      ~(Si = So) → exception(‘updates occurred ’, F) )
𝒰(CI,Si, Other,              So)   => Other ∧ Si = So
```

10.6.3.3 Pre-conditions

The update mapping for a pre-condition ensures no state change by doing $\mathcal{U}$ `(CI,Si,Clause,Si)`.

10.6.3.4 Post-conditions

The update mapping for a post-condition does $\mathcal{U}_{post}$`(CI,Si,Clause,So)`. $\mathcal{U}_{post}$ uses the initial `Si` for *had* and the final `So` for everything else. The update mapping for a post-condition ensures no state change.

```
    𝒰post (CI,Si, had(I,P,V),So) => has(CI,Si,I,P,V,Si)
    𝒰post (CI,Si,Other,So) => 𝒰 (CI,So,Other,So)
```

10.7 Formalization of the modeling constructs

This clause provides an overview of the relation of the graphics, RCL, metamodel, and axioms for each of the modeling constructs.

The following features have been omitted as a simplification. Their inclusion would complicate the formalization without affecting it in any substantial way.

— Aliases. They are assumed to have been replaced with their real names.
— Type `any`. All uses of `any` are assumed to have been replaced with `object`.
— Intrinsic properties and dependent classes. These are derivative ideas based on the notions of total, constant, and function.
— Arithmetic. Axioms for integer and real arithmetic are assumed.
— Changes to a value of an attribute used for the denotation of a metamodel instance, such as `#Cn` denoting a class where `Cn` is the class name.

Formally, there is a firm distinction between a symbol and its value in an interpretation. There is a further distinction between a symbol's value in any interpretation and its value in the intended interpretation. To fully maintain these distinctions, a string of symbols such as

1 + 3

would be described by something like

`1 + 3`	is the result of applying the function assigned by the interpretation to the function symbol `+` to the value assigned by the interpretation to the constant symbol `1` and the value assigned by the interpretation to the constant symbol `2`, where in the intended interpretation, the symbol `1` is assigned the integer `1` and the symbol `2` is assigned the integer `2` and the symbol `+` is assigned the integer addition function.

In the less formal style used in the descriptions in this clause, `1 + 3` is the result of applying the integer addition function to `1` and `3`.

All such statements are an abbreviation for the more formal version.

10.7.1 Objects

An object is a discrete thing, distinct from all other objects. Each object has an intrinsic, immutable *identity* (oid), independent of its property values and classification. An oid is abstract: it is always denoted indirectly, by a function application or a literal.

Throughout the formalization, in any expression such as "an object `V`," `V` should be understood to be the oid of the object.

10.7.2 Views

A view is a collection of classes and other views. The anonymous top-level view contains the classes representing the metamodel of IDEF$_{object}$.

Views are nested in view hierarchies; every user-defined view has one parent view. Every user-defined view has a unique, fully qualified name, `Vn`. For a view with the simple name Vsn and the top-level anonymous view as parent view, `Vn = Vsn`. For a view with the simple name Vsn and a parent view with the fully qualified name `Vn'`, `Vn = Vn':Vsn.`

For a view `V` with the fully qualified name `Vn`, `V = #Vn`. In other words, the # function maps `Vn` to the oid `V`. Throughout the formalization, in any expression such as "a view `V`," `V` should be understood to be the oid of a view.

The formalization assumes that fully qualified names are used for all classes.

10.7.3 Classes

Every object is classified into one or more classes and is an *instance* of each of those classes. The set of objects classified into a class is the *extent* of the class. Each class has a set of responsibilities. A responsibility is a constraint or a property, and a property is an attribute, participant property, or operation. A non-derived attribute or participant property is called a *fact property.* A value of a fact property is called a *fact*.

Every class is defined in exactly one view and has a unique name `Csn` within that view. For a parametric class, such as `set(T)`, the simple, unqualified name, `Csn`, is `set(T)` in the graphics and RCL, but for the purpose of formalization, `Csn = set:[T]`.

Every class has a unique, fully qualified name, `Cn`. For a class defined in the metamodel, `Cn = Csn`. For a class defined in any other view, `Cn = Vn:Csn`, where `Vn` is the fully qualified name of the view.

For a class `C` with the name `Cn`, `C = #Cn`. Throughout the formalization, in any expression such as "a class `C`," `C` should be understood to be the oid of a class.

There are two kinds of classes: state classes (sClass) and value classes (vClass).

10.7.3.1 State class

The objects in a *state class* are changeable in two ways: instances are created and deleted, and the facts about an instance can change. The identity of a state class object is denoted by an expression of the form `#DeclTerm`. For example, every class is an instance of the state class named `class` and the oid of an instance named `Cn` is `#Cn`. Formally, `#Cn` is a function application: the function denoted by `#` is applied to the name `Cn` to yield an oid.

Every class implements a type.

Table 24 shows expression forms that denote state class instance oids in the formalization.

Table 24—Expression forms denoting state class instance oids

Expression form	Oid of
`#Cn`	Class
`#(Cn:Pn PO T)`	Responsibility
`#(-Cn:Pn PO T)`	Implicit responsibility
`#(Cn:(-Pn) PO T)`	Pre-condition of responsibility `#(Cn:Pn PO T)`
`#(Cn:(+Pn) PO T)`	Post-condition of responsibility `#(Cn:Pn PO T)`

`PO` is one of the property operators, `:`, `:=`, `:!=`, `:+=`, `:-=`.

For a responsibility, `T` is an annotated type or a list of annotated types of the arguments. For an input argument, `T = +T′` where `T′` is the type of the argument. For an output argument, `T` is the type of the argument.

10.7.3.2 Value class

The objects in a value class do not change; they are pure values. The set of instances is fixed and the facts about an instance are fixed.

Table 25 shows expression forms that denote value class instance oids in the formalization.

Table 25—Expression forms denoting value class instance oids

Expression form	Oid of
`String`	An instance of string
`Identifier`	An instance of identifier
`Integer`	An instance of integer
`Real`	An instance of real
`truet`	An instance of boolean
`falset`	An instance of boolean
`SimpleObject : SimpleObject`	An instance of pair(T1,T2)
`SimpleObjectList`	An instance of list(T)
`{}`	The empty set
`#Cn:value(P,V,Value)`	An instance of a value class.

10.7.3.3 Collection and pair classes

The built-in pair and collection classes set, bag, and list are parametric value classes. Every class has a name, `Cn`. For the collection and pair classes, for any type `T`, `T'`, the names of the classes are as follows:

```
pair:[T,T']
collection:[T]
list:[T]
set:[T]
bag:[T]
```

A parametric value class can be used as the type of a variable or argument with an RCL `TypeLiteral` such as `set(pair(identifier,object))`. The corresponding expression denoting the instance of the state class `parametricVClass`, `#Cn`, is `#(set:[#(pair:[#identifier,#object])])`.

10.7.4 State

The set of all facts for all state class instances constitutes the *state* of the views. The initial state is just what is declared by declaration RCL. An update messages issued by an RCL query or within the realizations of a responsibility produces a whole new state. If a query or responsibility fails, no updates are made. (The updates made by successful nested messages are effectively backed out.)

The concept of state is formalized by an abstract data type. Axioms are given defining the known properties of the initial state, the constructors taking a state into a new state, a recognizer, and a selector that gets a fact based on a given state.

10.7.4.1 Initial state

The declaration clauses for a view constitute the axioms that define properties of the initial state, denoted by the constant symbol `facts`. A declaration `Cn:OID has Pn:K` declares it a fact that the class named `Cn` has an instance with an oid of `OID` and that instance has a property named `Pn` with a value of `K`. For each such declaration, the theory for the view acquires an axiom to that effect.

```
fact(facts, OID, #(Cn':Pn:T), K)
```

where `Cn':Pn:T` is the qualified property name of the direct or inherited property for `Pn`.

10.7.4.2 Constructors

The constructors `remember` and `forget` have the signature $\mathcal{I} \times \mathcal{P} \times \mathcal{V} \times \mathcal{S} \rightarrow \mathcal{S}$ where

$\mathcal{I}$ is the set of instances
$\mathcal{P}$ is the set of properties
$\mathcal{V}$ is the set of values
$\mathcal{S}$ is the set of states

10.7.4.3 Recognizer

The `isState` axioms defines what a state is.

```
isState(facts)
isState(remember(I,P,V,S)) ← isState(S)
isState(forget(I,P,V,S)) ← isState(S)
```

10.7.4.4 Selector

The arity 4 fact predicate

```
fact(S, I, P, V)
```

means that in state `S`, instance `I` has property `P` value `V`. The arity 4 fact predicate is used by the implicit realization for the recall property of state class instances.

```
fact(remember(I,P,V,S’),I,P,V) ← true

fact(remember(I’,P’,V’,S’),I,P,V) ← ~( I=I’ ∧ P=P’ ∧ V=V’ ) ∧
fact(S’,I,P,V)

fact(forget(I,P,V,S’),I,P,V) ← false

fact(forget(I’,P’,V’,S’),I,P,V) ← ~( I=I’ ∧ P=P’ ∧ V=V’ ) ∧
fact(S’,I,P,V)
```

10.7.5 Value

In concept, all instances of all value classes always exist. A literal specifies an instance by giving the values of properties constituting a uniqueness constraint. The class author defines a realization for the uniqueness constraint that derives the instance’s fact property values from the argument values, then says that the instance has those fact property values.

The concept of value is formalized by an abstract data type. Axioms are given defining the initial, constant value and a selector that gets a fact based on the value. There are no updates.

10.7.5.1 Initial value

The initial value for a value class named `Cn` is `#Cn:value(P,V,Value)`. The initial value of the value ADT is `value(P,V, Value)`.

10.7.5.2 Selector

The selector is the arity 3 fact predicate

```
fact(Value, P, V)
```

means that `Value` has property `P` value `V`. The arity 3 fact predicate is used by the implicit realization of the recall property of a value class instance.

```
fact(value(P’,V’,Rest),P,V) ←
     if P = P’
     then
          V = V’
     else
          fact(Rest,P,V)
     endif
```

10.7.6 Generalization

Generalization is concerned with the definition of objects. There is a single top class, called `object`. Every other class has at least one superclass. The meaning is that an object that is an instance of a class is also an instance of each superclass of that class.

10.7.6.1 Subclass

`C` isa_S `C′` means that in state `S`, `C` is a direct subclass of class `C′`.

```
    C isa_S C′ ←
    fact(S,C,#(class:highCluster:object),Clu) ∧
    fact(S,Clu,#(cluster:super:object),C′)
```

The reflexive, transitive closure of isa_S is isa_S* and the irreflexive, transitive closure is isa_S+.

The subclass to superclass relation is acyclic.

```
    C isa_S+ C ← false
```

10.7.6.2 LowClass

A subclass is said to be lower than its superclass. If an object is an instance of a class `C` and not an instance of any subclass of `C`, then `C` is a *lowclass* of the object. Every object has at least one lowclass. A value class instance has exactly one lowclass.

$I \ \varepsilon_S l \ C$

means that in state `S`, `I` is a direct instance of `C`, its lowClass.

For state class instance `I`,

```
I ε_Sl C ← fact(S,I,#(object:lowClass:object),C)
```

For value class instance `I`, including the collection and pair classes,

```
   I ε_Sl #Cn ←
        I = #(Cn:Type):value(P′,V′,Value),
        fact(S,#Cn,#(parametricVClass:nameCn:identifier),Cn),
        fact(S,#Cn,#(parametricVClass:type:list(class)),Type)
   I ε_Sl #Cn ← I = #Cn:value(P′,V′,Value), ~(Cn = Cn′:Type,isList(Type))
   I ε_Sl #pair(T1,T2) ← I = V1 : V2 ∧ V1 ε_Sl T1 ∧ V2 ε_Sl T2
   I ε_Sl #list(T) ← isList(I),lub(S,I,T)
   I ε_Sl #boolean ← I = true_t ∨ I = false_t
   I ε_Sl #identifier ← isIdentifier(I)
   I ε_Sl #character ← isCharacter(I)
   I ε_Sl #string ← isString(I)
   I ε_Sl #integer ← isInteger(I)
   I ε_Sl #real ← isReal(I)
```

The reflexive, transitive closure of $\varepsilon_S l$ is $\varepsilon_S l*$ and the irreflexive, transitive closure of $\varepsilon_S l$ is $\varepsilon_S l+$.

Except for `#class`, the lowclass relation is acyclic.

```
 C ε_Sl+ C ← (C = #class)
```

10.7.6.3 Instance

$I \ \varepsilon_S \ C$

means that in state S, I is an instance of C.

$I \ \varepsilon_S \ C \leftarrow I \ \varepsilon_{S1} \ C' \wedge C' \ \text{isa}_S{*} \ C$

10.7.7 Type

A class implements a type if it has all the responsibilities of the type. An object has type T if the object is an instance of a class that implements type T. Every class implements a type of the same name. Class #Cn implements type Cn. A type T is a subtype of type T′ if T includes all the responsibilities of T′. Unlike a class, a type does not have instances. Subtype is not the same as subclass. Subclass implies subtype, but not the other way round.

The object type relation is formalized by the ι predicate and subtyping by the <: predicate.

10.7.7.1 isType

```
    isType(S,T)
```

means that in state S, T is a type.

```
    isType(S, bot)
```
$\quad$ isType(S, Cn) $\leftarrow$ #Cn $\text{isa}_S{*}$ #object

10.7.7.2 Subtype Of

$T <:_S T'$

means that in state S, T is a subtype of T′. Subtype is reflexive and transitive.

For all types T, T′

$\text{bot} <:_S T$
$T <:_S T' \leftarrow T \ \text{isa}_S{*} \ T'$
#(Cpn:T) <: #(Cpn:T′) $\leftarrow$ T <: T′
[] <: []
[T|Ts] <: [T′|Ts′] $\leftarrow$ T <: T′, Ts <: Ts′

10.7.7.3 Type Of

$V \ \iota_S \ T$

means that in state S, object V is type T.

$V \ \iota_S \ T \leftarrow V \ \varepsilon_{S1} \ \#Cn \wedge Cn <:_S T$

From these definitions, it follows that for all I, T, T′, if $I \ \iota_S \ T$ and $T <:_S T'$ then $I \ \iota_S \ T'$.

10.7.7.4 Least Upper Bound

```
lub(S,List,T)
```

means that in state `S`, every member of List is type `T`, and `T` is the least type for which this is true.

The type `bot` is the only type that is not implemented by any class. Therefore, no object has type `bot`. The type `bot` is the least upper bound of an empty collection of types. So a collection class literal for an empty collection, such as `list(T):[]`, has the lowclass `list(bot).` This is the only use of the type `bot`.

$$\begin{array}{l} \texttt{lub(S,List,T)} \leftarrow \\ \qquad (\texttt{List = []}) \rightarrow (\texttt{T = bot}) \wedge \\ \qquad \forall(\texttt{M})(\texttt{M} \in \texttt{List} \rightarrow \texttt{M}\ \iota_S \texttt{T}) \wedge \\ \qquad \forall(\texttt{T'})(\texttt{lub(S,List,T')} \rightarrow \texttt{T} <:_S \texttt{T'}) \end{array}$$

10.7.8 Responsibility

Each class has a set of responsibilities. A responsibility is a constraint or a property; and a property is an attribute, participant property, or operation.

The responsibilities for a class are stated with the graphics. For example, Figure 103 states that class `Cn` has an operation named `Pn` that has two arguments.

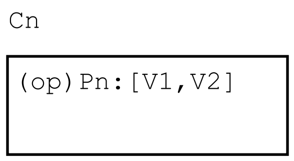


Figure 103—A responsibility of a class stated graphically

`Cn` is a class name, `Pn` is a responsibility name, and `[V1, V2]` is the list of arguments. In this example, no types are specified for the arguments, so each argument's type is `object`.

In the formalization, every responsibility has at least one argument. If a responsibility has no arguments specified in the graphics, a single (output) argument of type boolean is assumed.

Every responsibility is defined in exactly one class. Every responsibility has a unique, fully qualified name,

```
      QPnT = Cn:Pn PO Type
```

where `Cn` is the fully qualified name of the class, `Pn` is the simple name of the responsibility, `PO` is the property operator (i.e., `:, :=, :!=, :+=, or :-=`), and `Type` is the type of the single argument or a list of the types of the multiple arguments. For the example, the qualified property name, `QPnT`, is `Cn:Pn:[object,object].`

As part of the formalization, the graphics are restated using declaration RCL, such as

```
      operation: #(Cn:Pn:[object,object]) has lowClass: #operation.
      operation: #(Cn:Pn:[object,object]) has class:#Cn.
```

Here, `#Cn` denotes an instance of the class named `class` and `#(Cn:Pn:[object,object])` denotes an instance of the class named `operation` that is associated with `Cn`.

Intuitively, such declarations populate the metamodel. The relevant fragment is shownin Figure 104.

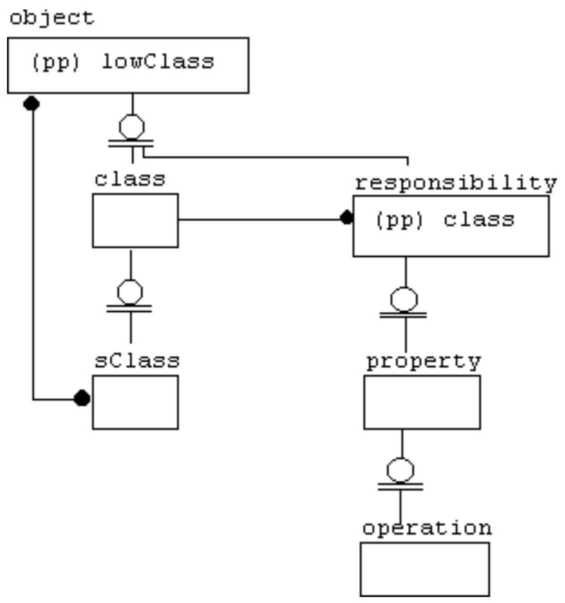


Figure 104—Metamodel fragment showing responsibility

For a responsibility R with the fully qualified name QPnT, R = #QPnT. In other words, in the intended interpretation, the # function maps QPnT to the oid R. Throughout the formalization, in any expression such as “a responsibility R,” R should be understood to be the oid of a responsibility.

Formally, a responsibility is a *relation* R_{CnPn}—a set of N-tuples, where N is the number of arguments plus 3. For the example, R_{CnPn} is a set of 5-tuples of the form < Si,I,V1,V2,So >, where Si is the input state, I is the receiving instance, V1 is the value of the first argument, V2 is the value of the second argument, and So is the output state. All such relations are derivable from a single arity 6 *has* predicate where *has*(CI,Si,I,P,V,So) means that with input state Si, instance I has a property P value V and the output state is So. (CI is not relevant here). R_{CnPn} is derivable by

```
R_CnPn = { <Si,I,V1,V2,So> where
      has(CI,Si,I,#(Cn:Pn:[object,object]),[V1,V2],So) }
```

If the arguments are partitioned into input and output arguments, for example V1 input and V2 output, then the relation R_{CnPn} defines a *relational mapping* $M_{CnPn} : E_S \times E_{Cn} \times E_{object} \rightarrow E_{object} \times E_S$. The relational mapping can be declared total or partial, and single-valued or multi-valued. The default is total, single valued.

A single-valued relational mapping is a functional mapping. The unqualified term “mapping” means a functional mapping.

10.7.9 Realization

A realization states the necessary and sufficient conditions that the receiver object has the property value. Syntactically, a realization is

```
class_qname: Variable has responsibility_name {:Variables} if_def
Sentence.
```

In Figure 105, `Cn` is a class name, `Pn` is a responsibility name, and `V` is the argument variable (list of argument variables). No argument types are specified in the example.

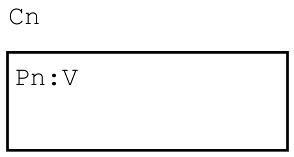


Figure 105—Class `Cn` graphically

The realization RCL is

$$\texttt{Cn: Self has Pn: V if}_{\texttt{def}}\texttt{ Sentence.}$$

Self is the receiver—the object being asked to meet the responsibility. The sentence typically contains propositions (often messages) using the variables in `V`. These propositions so constrain `V` that they define values for the variables in `V`.

Read declaratively, a realization says that the responsibility is met (i.e., it is true that `Cn: Self has Pn: V`) if the `Sentence` is true. Read procedurally, a realization says that to solve for the output variables, solve the sentence. Informally, solution can be thought of as a computation as in any programming language.

In the formalization, the realization RCL is mapped by a mapping $\mathcal{M}$ to

$$\mathcal{has}\texttt{(CI, Si, Self, \#(Cn:Pn:Type), V, So)} \leftarrow \mathcal{M}\texttt{(Sentence)}.$$

Formally, solution means that the input state, receiver, and input arguments map (by the relational mapping) to the output arguments and output state. With an adequate proof technique, solution means proof. In other words, to prove that

$$\mathcal{has}\texttt{(CI, Si, Self, \#(Cn:Pn:Type), V, So)}$$

is true, prove that the `Sentence` is true. (The implication states only the if direction. The only-if direction is explained in 10.3.11.)

10.7.10 Relationships

A relationship relates the instances of one class to the instances of another (possibly the same) class. In other words, a relationship is binary and bidirectional. Each class has a participant property that has as its value the identity of a related instance. A participant property that is a fact has implicit realizations. The implicit realizations maintain the consistency of the participant properties—instance `I` is related to instance `I'` if and only if `I'` is related to `I`. Consistency is maintained by two rules.

Message to participant	Also do for inverse
`I has Pn :+= I'`	`I' has Pn' :+= I`
`I has Pn :-= I'`	`I' has Pn' :-= I`

The message to the participant is governed by the same rules as for any other property-operator-implicit realization. No relationship constraints are checked by `add` or `remove`.

If an `add`, a request is made to add to the inverse participant property.

```
participant: Self has add:[I,V] if_def
Self super has add:[I,V],
I has inverse: P,
P has add:[V,I]
```

If a `remove`, a request is made to remove from the inverse participant property.

```
participant: Self has remove:[I,V] if_def
Self super has remove:[I,V],
I has inverse: P,
P has remove:[V,I]
```

10.7.11 Creating a state class instance

The `create`, `new`, and `init` properties of `sClass` provide a way to create and initialize any state class. The modeler can override these properties, or define their own constructor properties using only the new property of `sClass`.

The `create` property of `sClass` creates an instance of a state class. The message to create an instance of the class named `Cn` has the following form

```
#Cn has create([Pn1:V1,Pn2:V2, …, Pnn:Vn],I)
```

where

each `Pni:Vi` is a direct or inherited property `Pni` having initial value `Vi`. In the optional `I` argument, `I` is the oid of the created instance. `I` can be a variable or `#Constant`. As an example,

```
#(dogView:dog) has create([tag:25071],#luke)
```

`Create` gets a new instance and initializes it.

`New` is an instance level property of the metaclass `sClass`. It has both implicit and explicit realizations. The explicit `new:I` gets the next oid `#N` and if `#N` is not equal to `I` asserts no current oid is equal to I, and sends to the superclass for the implicit `new:I`. The implicit `new:I` has an axiom causing `So` to include the fact that `I` has lowclass `#Cn`. The explicit `new:I` then adds `I` to the oids and adds `I` as a direct instance.

The initialization is done by setting each property to its value. An `init` property explicitly defined for a class overrides the init in `sClass`.

10.7.12 Adding an instance to a state class

The `add` property of `sClass` adds an instance to a state class. As an example,

```
#(dogView:dog) has add([tag:23],#buck)
```

The message to add an existing instance to the class named `Cn` has the following form:

```
#Cn has add([Pn1:V1,Pn2:V2, …, Pnn:Vn],I)
```

where each `Pni:Vi` is a direct or inherited property of `Cn` and each `Pni` is to have an initial value `Vi`. The argument `I` is the oid of the instance to be added. The realization for `add`

a) Verifies that no superclass of `Cn` is in the same cluster as any class of `I`,
b) Removes any superclasses of `Cn` from the lowclasses of `I`,
c) Adds `Cn` as a lowclass of `I`,
d) Initializes `I`.

The result is that every superclass of `Cn` that is not already a class of `I` becomes a class of `I`. If any `Pni` is a property of an existing class of `I`, and is not a property of an added class, it is reset to `Vi`. Adding an instance `I` to class `Cn` when `Cn` already contains I initializes `I`.

10.7.13 Deleting a state class instance

An instance of a state class is deleted by sending a delete message to it. Delete is a responsibility of `object`. An object deletes itself by sending a `propagateDelete` to each `lowClass` and then removing the lowclass. When all lowclasses have been removed, the object is effectively deleted. It can no longer be sent messages. The `isBeingDeleted` marker prevents access to partially deleted objects because such access could find a cardinality violation and raise an exception.

The `sClass` operation `propagateDelete` takes as `argument` its instance `I` to be deleted. `I` is told to stop participating in any relationships and a `propagateDelete` is sent to each of its superclasses, but not to `#object`. At `object` (i.e., `Self = #object`), all the participant property `lowClass` values would be removed. Such removal would prevent `I` from being sent any messages.

To make `I` a nonparticipant, each participant property value `I'` is removed, and `I'` is deleted if need be in order to prevent a violation of its cardinality constraint.

10.7.14 Removing an instance from a state class

The `remove` property of `sClass` removes an instance from a state class. As an example,

```
#(dogView:dog) has remove:#buck
```

The message to remove an existing instance to the class named `Cn` is of the form

```
#Cn has remove:I
```

where I is the oid of the instance to be removed.

Removing an instance from a class requires removing it from all subclasses and all total cluster superclasses. The realization for remove

a) Sends a `remove` to each subclass.
b) Tells `I` to stop being a participant in any relationship.
c) if `Self` is a lowclass of `I`, removes it and adds any superclass of `Self` that is not a class of `I`.
d) `forall highCluster` is total, sends a `remove` to superclass

The result is that `Cn` is not a superclass of any lowclass of `I`.

10.7.15 Coercions

No automatic coercions are done. Coercions can be done like any other operation. For example, the message

```
    V has asReal: X
```

obtains the real X for the integer V.

10.7.16 is

The base case for the empty list with `foreach` is common.

is`(CI,S,[],foreach(K,X),S)`

is`(CI,S,[],foreach(K,X,Acc),S)`

10.7.17 Inheritance

A subclass *inherits* the responsibilities of its superclasses. An instance inherits the responsibilities of its class. An inheritor can *override* one or more of its inherited responsibilities. The inheritance order is shown in Figure 106. In Figure 106, the arrows are labeled with the predicate symbols used in the formalization.

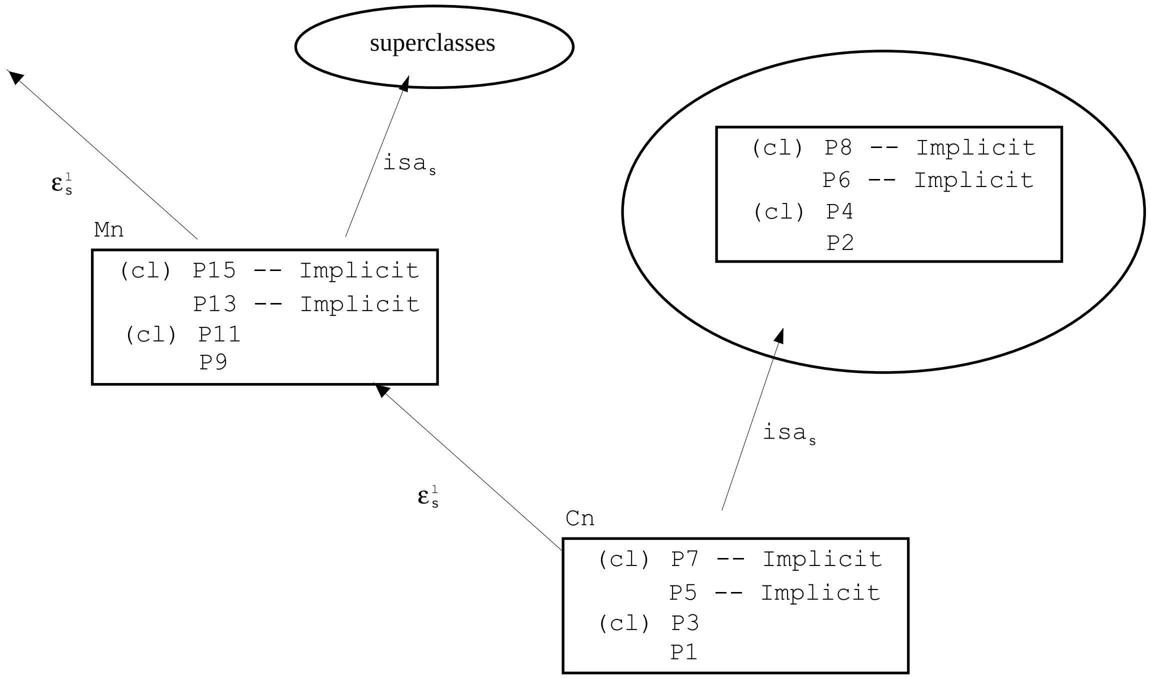


Figure 106—Inheritance order

A message to an instance of `Cn` will search for a match in this order:

- `Cn` to superclasses to object for explicit instance methods `P1, P2`…
- `Cn` to superclasses to object for explicit class methods `P3, P4`…
- `Cn` to superclasses to object for implicit instance methods `P5, P6`…
- `Cn` to superclasses to object for implicit class methods `P7, P8`…
- `Mn` to superclasses to object for explicit instance methods `P9,` …
- `Mn` to superclasses to object for explicit class methods `P11,` …
- `Mn` to superclasses to object for implicit instance methods `P13,` …
- `Mn` to superclasses to object for implicit class methods `P15,` …
- Etc.

A message to `#Cn` searches in the same order, but starts at explicit class methods, `P3`.

A message to `Self super` issued within a realization starts the search at the next point beyond that realization, as though the realization had not been there for the search that found it. For example, if `P2` is initially found for the message to an instance of `Cn`, and `P2` issues a `Self super` message, the search begins at `P3`.

The search ignores private or protected responsibilities that are hidden to the message sender.

10.7.18 Message

A request to an object to carry out one of its responsibilities is called a *message*. A message consists of the identity of the receiver, the name of a responsibility, the values of the input arguments, and the (typically unknown) values (as variables) of the output arguments for the responsibility. Syntactically, a message is

```
Object has Responsibility { PropertyOperator Object }
```

In the formalization, if there is no `PropertyOperator`, `: true`$_t$ is assumed. In the messages

```
I has Pn: V
I has R: V
```

`I` is the receiver, `Pn` is the name of the responsibility, `R` is a responsibility, and `V` is the argument value or list of argument values. The receiver or an argument value can be a constant, an instance of a state class, an instance of a value class, or a variable denoting any of them. In the formalization, the messages are mapped to

```
has(CI,Si,I,Pn:V,So)
has(CI,Si,I,R:V,So)
```

respectively, where `CI` is the sender, `Si` is the input state, and `So` the output state.

```
has(CI,Si,I,Pn:V,So)
```

means that evaluating the message `I has Pn: V` sent by `CI` in state `Si` with the values of the input arguments given by `V` produces the value of the output arguments given by `V` and the new state `So`.

```
has(CI,Si,I,R:V,So)
```

means that evaluating the message `I has R: V` sent by `CI` in state `Si` with the values of the input arguments given by V produces the value of the output arguments given by V and the new state So.

`Si` is mapped to `So` only if the proposition is true. In other words, a state change is possible only if the message succeeds. The new state will differ from the previous if the message was dynamically bound to one of the implicit (i.e., built-in) realizations for updating a fact property (e.g., `I has Pn:=V`), or was bound to an explicit (i.e., user written) responsibility that included a message that changed the state. If a message fails, no updates are done, including those of successful nested messages.

The first kind of message, `I has Pn: V`, must be bound to a responsibility `R`. The second kind of message, `I has R: V`, needs no binding, but the set `IRs` of overridden responsibilities is needed for checking the pre-conditions and post-conditions. Allowing for the property operators, the message axiom is

```
has(CI, Si, I, RPnV, So)←
     bind(CI, Si, I, RPnV, POn, IRs, I', R, V, T) ∧
```

```
          ~(R $\varepsilon_S$ #constraint) ∧
          ~preOk(Si, IRs, V, T) → exception('pre condition', IRs) ∧
          cardinalityOk(CI, Si, I', POn, R, V, T) ∧
          issue(CI, Si, I', POn, R, V, So) ∧
          ~postOk(Si, IRs, V, So) → exception('post condition', IRs)

     has(CI, Si, I, RPnV, So)←
          bind(CI, Si, I, RPnV, POn, IRs, I', R, V, T) ∧
          R $\varepsilon_S$ #constraint ∧
          ∀(I',R')(I':R' ∈ IRs → issue(CI, Si, I', POn, R', V, So)
```

If a message is bound to an implicit fact, a message is issued for the property operator by name.

```
     issue(CI, Si, I', POn, #(-QPnT), V, So) ← -- implicit fact
          fact(S,R,#(responsibility:isFact:boolean),true$_t$) ∧
          has(CI,Si,R,POn:[I',V],So)         -- send message for POn

     issue(CI, Si, I', POn, R, V, So) ←    -- not implicit fact
          ~∃(QPnT)(R = #(-QPnT) ∧
          fact(S,R,#(responsibility:isFact:boolean),true$_t$))
          has(_:I', Si, I', R, V, So)-- realization
```

10.7.19 Implicit realizations

An implicit realization is "built in" and not written by the modeler. An explicit realization is one written by the modeler.

Anthropomorphically, an object realizes a responsibility by derivation or memory. A responsibility realized by memory is called a *fact*.

Attributes and participant properties are facts unless declared a derivation by the suffix keyword `derived`. (The `derived` keyword is intended for use on views for class producers, not on views for class clients.) Constraints and operations are always realized by derivation.

An attribute or participant property can have a responsibility declaration for any or all the property operators. If any is declared derived, then all must be. If either participant is derived, then the other must be.

For any fact (i.e., nonderived) property, the graphics must include a get (:) interface specification for the property. The interface specifications for any other property operator shall appear in the same class or a subclass.

An explicit override of a fact should not be declared `derived`.

All facts and certain derivations have implicit realizations. An implicit realization can be overridden by an explicit realization. Within an override, a message to `super` can be issued to the implicit realization.

10.7.19.1 Facts

Table 26 describes the use of the `isFact` and plicity properties for a fact responsibility.

`Set` and `unset` are defined in terms of `get`, `add`, and `remove`.

Table 26—isFact and plicity properties for a fact responsibility

IsFact	Plicity	Oid	Comment
`true`	`implicit`	`#(-Cn:Pn:T)`	As a participant, this instance knows the inverse. As a characteristic, the `get`, `set`, `unset`, `add`, and `remove` operations provide the implicit realizations for the property operators.
`true`	`explicit`	`#(Cn:Pn:T)`	Override
`true`	`explicit`	`#(Cn:Pn:=T)`	Override state class only
`true`	`explicit`	`#(Cn:Pn:!=T)`	Override state class only
`true`	`explicit`	`#(Cn:Pn:+=T)`	Override state class only
`true`	`explicit`	`#(Cn:Pn:-=T)`	Override state class only
`true`	`explicit`	`#(Cn:Pn:T)`	Override state class only

For a property `Pn` of type `T` which is not a single-valued collection, `get` gets a value for `Pn`, `add` adds a new value for `Pn`, and `remove` removes a value for `Pn`, where the value has type `T`.

The `get`, `set`, `unset`, `add`, and `remove` properties are declared as a part of the metamodel like any other, and their RCL realizations are mapped to clausal form like any other.

In 10.7.19.1.1 through 10.7.19.1.5, `I` is a receiver and `V` a value.

10.7.19.1.1 `(op) characteristic: get`

If the property has value `V`, the request succeeds with the state unchanged. If the property does not have value `V`, the request fails with the state unchanged.

```
characteristic: Self has get:[I,V] if_def
     I has recall: (Self:V).
```

10.7.19.1.2 `(op) characteristic: set`

If a `set` and, at the time the request is issued, the property has the value `V’`, the value, if not `V`, is removed, and the new value is added.

```
characteristic: Self has set:[I,V] if_def
     forall ( I has recall: (Self:V’), not (V = V’) ):
           (I has forget: (Self:V’),
if         not I has recall: (Self:V)
then
           I has remember: (Self:V)
endif.
```

10.7.19.1.3 `(op) characteristic: unset`

If an `unset` and the property has that value, the value is removed. If the property does not have that value, the request fails.

```
characteristic: Self has unset:[I,V] if_def
```

```
          I has recall: (Self:V),
          I has forget: (Self:V).
```

10.7.19.1.4 **`(op) characteristic: add`**

If an `add` and the property already has value `V`, the request succeeds with the state unchanged.

If an `add` to a single-valued collection and the collection already has member `V`, the request succeeds with the state unchanged. If there is no collection, the request fails. If there is a collection and it does not contain the added member, it is added by an `insertLast`.

```
     characteristic: Self has add:[I,V] if_def
          if   Self has isFunction,-- single valued
               Self has type..superStar: collection(T),
          then
               I has recall: (Self:Collection),
               Collection has insertLast:[V,NewCollection],
               if not Collection = NewCollection
                    then
                           I has forget: (Self:Collection),
                           I has remember: (Self:NewCollection)
                    endif
          else
               if not I has recall: (Self:V)
               then
                    I has remember: (Self:V)
               endif
          endif.
```

10.7.19.1.5 **`(op) characteristic: remove`**

If a `remove` and the property does not have value `V`, the request succeeds with the state unchanged.

If a `remove` from a single-valued collection and the collection has no member `V`, the request succeeds with the state unchanged. If there is no collection, the request fails. If there is a collection and it contains the member, it is removed.

```
     characteristic: Self has remove:[I,V] if_def
          if   Self has isFunction,-- single valued
               Self has type..superStar: collection(T),
          then
               I has recall: (Self:Collection),
               Collection has remove:[V,NewCollection],
               if not Collection = NewCollection
               then
                    I has forget: (Self:Collection)
                    I has remember: (Self:NewCollection)
               endif,
          else
               if I has recall: (Self:V)
               then
                    I has forget: (Self:V)
               endif
     endif.
```

10.7.19.2 Derivations for basic types

Table 27 describes the use of the `isFact` and plicity properties for a derived responsibility.

Table 27—isFact and plicity properties of a derived responsibility

IsFact	Plicity	Oid	Comment
`false`	`implicit`	`#(-Cn:Pn:T)`	Represents built-in operations that are directly realized by axioms.
`false`	`explicit`	`#(Cn:Pn:T)`	Derived
`false`	`explicit`	`#(Cn:Pn:=T)`	Derived
`false`	`explicit`	`#(Cn:Pn:!=T)`	Derived
`false`	`explicit`	`#(Cn:Pn:+=T)`	Derived
`false`	`explicit`	`#(Cn:Pn:-=T)`	Derived
`false`	`explicit`	`#(Cn:Pn:T)`	Derived

The implicitly realized derived properties are declared as a part of the metamodel like any other, but no RCL is specified for the realization. The implicit realization is overridden by supplying RCL for an explicit realization.

Every theory includes

— The declarations for these responsibilities, rewritten to definition clausal form.
— The realizations for these responsibilities.

The declarations are done in the same way as any other responsibility, except that the `OID = #(-X)` instead of `#X`. (The declarations of the implicit responsibilities are in addition to the declarations for the explicit responsibilities that result from declaring instances of the metamodel for the metamodel view.) The declarations are then rewritten to definition clausal form like any other declaration RCL.

The definition clausal form for the axiom of an implicit realization is directly included as a part of every theory (instead of being produced by a mapping from RCL). The axioms for the implicit realizations for derivations are given below.

The implicit realizations are used to avoid direct access in RCL to the state `ADT`, value `ADT`, the form of `Self`, or the representation of the `integer`, `real`, `identifier`, `character`, `string`, `pair`, or `list` types assumed by their base theories.

For each of the types for which there is a base theory, the implicit realizations provide mapping between the representation for value classes and the representation used in the theory (called `Rep` in the axioms). None of these axioms need the sender, state, or type parts of the arguments, so a simplified *has*`(_,_,Self,#(-Cn:Pn:_),V,_)` pattern is used for the arguments.

For each base type, the theory's representation, `Rep`, is either the oid of the instance, `Self`, or the value of a property named for the class, `Cn`. The `theoryRep` predicate relates `Cn` and `Self` to `Rep`.

```
theoryRep(list,Rep,Rep) ← isList(Rep)
theoryRep(pair,Rep,Rep) ← isPair(Rep)
theoryRep(character,Rep,Rep) ← isCharacter(Rep)
theoryRep(identifier,Rep,Rep) ← isIdentifier(Rep)
theoryRep(string,Rep,Rep) ← isString(Rep)
theoryRep(integer,Rep,Rep) ← isInteger(Rep)
theoryRep(real,Rep,Rep) ← isReal(Rep)
theoryRep(Cn,(#Cn:Value),Rep) ← fact(Value,#(Cn:Cn:#Cn),Rep)
```

10.7.19.2.1 (op) list: isEmpty

RCL Head:	`list: Self has isEmpty`
Mapping:	{#list..instance} → {#boolean..instance}
Variables	`Self`

```
has(_,_,Self,#(-Cn:isEmpty:_),true_t),_) ←
    Cn = list ∧
    theoryRep(Cn,Self,Self') ∧
    Self' = []
```

10.7.19.2.2 (op) list(T): prefix

RCL Head:	`list(T): Self has prefix:[X:T,List:list(T)]`
Mapping:	{#(list:[T])..instance} × {#T..instance} → {#(list:[T])..instance}
Variables	`Self × X → List`

```
has(_,_,Self,#(-Cn:prefix:_),[X,List]),_) ←
    Cn = list ∧
    theoryRep(Cn,Self,Self') ∧
    theoryRep(T,X,X') ∧
    List = prefixList(X',Self')
```

10.7.19.2.3 (at) list(T): first

RCL Head:	`list(T): Self has first:(X:T)`
Mapping:	{#(list:[T])..instance} → {#T..instance}
Variables	`Self → X`

```
has(_,_,Self,#(-Cn:first:_),X),_) ←
    Cn = list ∧
    theoryRep(Cn,Self,Self') ∧
    Self' = prefixList(X,_)
```

10.7.19.2.4 (at) list(T): rest

RCL Head:	`list(T): Self has rest:(List:list)`
Mapping:	{#(list:[T])..instance} → {#(list:[T])..instance}
Variables	`Self → List`

```
has(_,_,Self,#(-Cn:rest:_),List),_) ←
    Cn = list ∧
    theoryRep(Cn,Self,Self') ∧
    Self' = prefixList(_,List)
```

10.7.19.2.5 (at) pair(T1,T2): left

RCL Head:	`pair(T1,T2): Self has left:(X:T)`
Mapping:	{#(pair:[T1,T2])..instance} → {#T1..instance}
Variables	`Self → X`

```
has(_,_,Self,#(-Cn:left:_),X),_) ←
```

```
            Cn = pair ∧
            theoryRep(Cn,Self,Self') ∧
            Self' = X:_
```

10.7.19.2.6 (at) pair(T1,T2): right

```
RCL Head:   pair(T1,T2): Self has right:(Y:T2)
Mapping:    {#(pair:[T1,T2])..instance} → {#T2..instance}
Variables   Self → Y
        has(_,_,Self,#(-Cn:right:_),Y),_) ←
            Cn = pair ∧
            theoryRep(Cn,Self,Self') ∧
            Self' = _:Y)
```

10.7.19.2.7 (op) identifier: isEmpty

```
RCL Head:   identifier: Self has isEmpty
Mapping:    {#identifier..instance} → {#boolean..instance}
Variables   Self
        has(_,_,Self,#(-Cn:isEmpty:_),true_t),_) ←
            Cn = identifier ∧
            theoryRep(Cn,Self,Self') ∧
            Self' = ''
```

10.7.19.2.8 (op) identifier: prefix

```
RCL Head:   identifier: Self has prefix:[C:character,Ident:identifier]
Mapping:    {#identifier..instance} × {#character..instance} → {#identifier..instance}
Variables   Self × C → Ident
        has(_,_,Self,#(-Cn:prefix:_),[C,Ident]),_) ←
            Cn = identifier ∧
            theoryRep(Cn,Self,Self') ∧
            theoryRep(character,C,C') ∧
            Ident = prefixIdentifier(C',Self')
```

10.7.19.2.9 (at) identifier: first

```
RCL Head:   identifier: Self has first:(C:character)
Mapping:    {#identifier..instance} → {#character..instance}
Variables   Self → C
        has(_,_,Self,#(-Cn:first:_),C),_) ←
            Cn = identifier ∧
            theoryRep(Cn,Self,Self') ∧
            Self' = prefixIdentifier(C,_)
```

10.7.19.2.10 (at) identifier: rest

```
RCL Head:   identifier: Self has rest:(Ident:identifier)
Mapping:    {#identifier..instance} → {#identifier..instance}
Variables   Self → Ident
        has(_,_,Self,#(-Cn:rest:_),Ident),_) ←
            Cn = identifier ∧
            theoryRep(Cn,Self,Self') ∧
            Self' = prefixIdentifier(_,Ident)
```

10.7.19.2.11 `(op) string: isEmpty`

RCL Head: `string: Self has isEmpty`
Mapping: {#string..instance} → {#boolean..instance}
Variables `Self`

```
has(_,_,Self,#(-Cn:isEmpty:_),true_t),_) ←
    Cn = string ∧
    theoryRep(Cn,Self,Self') ∧
    Self' = ""
```

10.7.19.2.12 `(op) string: prefix`

RCL Head: `string: Self has prefix:[C:character,Str:string]`
Mapping: {#string..instance} × {#character..instance} → {#string..instance}
Variables `Self × C → Str`

```
has(_,_,Self,#(-Cn:prefix:_),[C,Str]),_) ←
    Cn = string ∧
    theoryRep(Cn,Self,Self') ∧
    theoryRep(character,C,C') ∧
    Str = prefixString(C',Self')
```

10.7.19.2.13 `(at) string: first`

RCL Head: `string: Self has first:(C:character)`
Mapping: {#string..instance} → {#character..instance}
Variables `Self → C`

```
has(_,_,Self,#(-Cn:first:_),C),_) ←
    Cn = string ∧
    theoryRep(Cn,Self,Self') ∧
    Self' = prefixString(C,_)
```

10.7.19.2.14 `(at) string: rest`

RCL Head: `string: Self has rest:(Str:string)`
Mapping: {#string..instance} → {#string..instance}
Variables `Self → Str`

```
has(_,_,Self,#(-Cn:rest:_),Str),_) ←
    Cn = string ∧
    theoryRep(Cn,Self,Self') ∧
    Self' = prefixString(_,Str)
```

10.7.19.2.15 `(op) integer: '+'`

RCL Head: `integer: Self has '+':(Int:integer,Result:integer)`
Mapping: {#integer..instance} × {#integer..instance} → {#integer..instance}
Variables `Self → Int × Result`

```
has(_,_,Self,#(-Cn: '+':_),[Int,Result]),_) ←
    Cn = integer ∧
    theoryRep(Cn,Self,Self') ∧
    theoryRep(Cn,Int,Int') ∧
    iPlus(Self',Int',Result)
```

10.7.19.2.16 `(op) integer: '-'`

RCL Head: `integer: Self has '-':(Int:integer,Result:integer)`
Mapping: {#integer..instance} × {#integer..instance} → {#integer..instance}
Variables `Self → Int × Result`

```
has(_,_,Self,#(-Cn: '-' :_),[Int,Result]),_) ←
        Cn = integer ∧
        theoryRep(Cn,Self,Self') ∧
        theoryRep(Cn,Int,Int') ∧
        iMinus(Self',Int',Result)
```

10.7.19.2.17 (op) integer: '*'

RCL Head: `integer: Self has '*':(Int:integer,Result:integer)`
Mapping: {#integer..instance} × {#integer..instance} → {#integer..instance}
Variables `Self → Int × Result`

```
has(_,_,Self,#(-Cn:'*':_),[Int,Result]),_) ←
        Cn = integer ∧
        theoryRep(Cn,Self,Self') ∧
        theoryRep(Cn,Int,Int') ∧
        iTimes(Self',Int',Result)
```

10.7.19.2.18 (op) integer: '/'

RCL Head: `integer: Self has '/':(Int:integer,Result:integer)`
Mapping: {#integer..instance} × {#integer..instance} → {#integer..instance}
Variables `Self → Int × Result`

```
has(_,_,Self,#(-Cn:'/':_),[Int,Result]),_) ←
        Cn = integer ∧
        theoryRep(Cn,Self,Self') ∧
        theoryRep(Cn,Int,Int') ∧
        iDivideby(Self',Int',Result)
```

10.7.19.2.19 (op) integer: '^'

RCL Head: `integer: Self has '^':(Int:integer,Result:integer)`
Mapping: {#integer..instance} × {#integer..instance} → {#integer..instance}
Variables `Self → Int × Result`

```
has(_,_,Self,#(-Cn:'^':_),[Int,Result]),_) ←
        Cn = integer ∧
        theoryRep(Cn,Self,Self') ∧
        theoryRep(Cn,Int,Int') ∧
        iExp(Self',Int',Result)
```

10.7.19.2.20 (op) real: '+'

RCL Head: `real: Self has '+':(Real:real,Result:real)`
Mapping: {#real..instance} × {#real..instance} → {#real..instance}
Variables `Self → Real × Result`

```
has(_,_,Self,#(-Cn: '+':_),[Real,Result]),_) ←
      Cn = real ∧
      theoryRep(Cn,Self,Self') ∧
      theoryRep(Cn,Real,Real') ∧
      rPlus(Self',Real',Result)
```

10.7.19.2.21 (op) real: '-'

RCL Head: `real: Self has '-':(Real:real,Result:real)`
Mapping: {#real..instance} × {#real..instance} → {#real..instance}
Variables `Self → Real × Result`

```
    has(_,_,Self,#(-Cn: '-' :_),[Real,Result]),_) ←
         Cn = real ∧
         theoryRep(Cn,Self,Self') ∧
         theoryRep(Cn,Real,Real') ∧
         rMinus(Self',Real',Result)
```

10.7.19.2.22 (op) real: '*'

```
RCL Head:    real: Self has '*':(Real:real,Result:real)
Mapping:     {#real..instance} × {#real..instance} → {#real..instance}
Variables    Self → Real × Result
    has(_,_,Self,#(-Cn:'*':_),[Real,Result]),_) ←
         Cn = real ∧
         theoryRep(Cn,Self,Self') ∧
         theoryRep(Cn,Real,Real') ∧
         rTimes(Self',Real',Result)
```

10.7.19.2.23 (op) real: '/'

```
RCL Head:    real: Self has '/':(Real:real,Result:real)
Mapping:     {#real..instance} × {#real..instance} → {#real..instance}
Variables    Self → Real × Result
    has(_,_,Self,#(-Cn:'/':_),[Real,Result]),_) ←
         Cn = real ∧
         theoryRep(Cn,Self,Self') ∧
         theoryRep(Cn,Real,Real') ∧
         rDivideby(Self',Real',Result)
```

10.7.19.2.24 (op) real: '^'

```
RCL Head:    real: Self has '^':(Real:real,Result:real)
Mapping:     {#real..instance} × {#real..instance} → {#real..instance}
Variables    Self → Real × Result
    has(_,_,Self,#(-Cn:'^':_),[Real,Result]),_) ←
         Cn = real ∧
         theoryRep(Cn,Self,Self') ∧
         theoryRep(Cn,Real,Real') ∧
         rExp(Self',Real',Result)
```

10.7.19.3 Derivations for metamodel classes

10.7.19.3.1 (op) object: exception

```
    has(#object:I,Si,I,Exception,[R,X], So) ←
         Exception = #(-object:exception:[+object,+object]) ∧
         exception(R,X)
```

10.7.19.3.2 (op) object: remember

```
    has(#object:I,Si,I,Remember, (P:V), So) ←
         Remember = #(-object:remember:
         pair(+#characteristic,#object))∧
         So = remember(I,P,V,Si)
```

10.7.19.3.3 (op) object: forget

```
has(#object:I,Si,I,Forget, (P:V), So) ←
     Forget = #(-object:forget: pair(+#characteristic,#object))∧
     So = forget(I,P,V,Si)
```

10.7.19.3.4 (op) object: recall

```
has(#object:I,S,I,Recall, (P:V), S) ←
     Recall = #(-object:recall: pair(+#characteristic,#object))∧
     fact(S,I,P,V)
```

10.7.19.3.5 (op) parametricVClass: instance

```
has(#parametricVClass:I,S,I,Instance, #Cn':value(type,Type',Value),
S) ←
     Instance = #(-parametricVClass:instance: I),
     fact(S,I,#(parametricVClass:nameCn:identifier),NameCn),
     fact(S,I,#(parametricVClass:type:list(class)),Type),
     build(Type,Type'),
     Cn' = NameCn:Type',

build([],[])
build([T|Ts],[T'|Ts']) ← build(Ts,Ts')
```

10.7.19.3.6 (op) value: recall

```
has(#value:I,S,I,Recall, (P:V), S) ←
     I = #Cn:value(P',V',Value) ∧
     Recall = #(-value:recall: pair(+#characteristic,#object))∧
     fact(value(P',V',Value), P, V )
has(#value:I,S,I,Recall, (#(Cn:Cn:#Cn):I), S) ←
     Recall = #(-value:recall: pair(+#characteristic,#object)) ∧
     ~(I = #Cn:value(P',V',Value)) ∧
     I εSl #Cn
```

10.7.19.3.7 (op) value: lowClass

```
has(#value:I,S,I,LowClass, #Cn, S) ←
     I = #Cn:value(P',V',Value) ∧
     LowClass = #(-value:lowClass: #vClass)
```

10.7.19.3.8 (op) value: type

```
has(#value:I,S,I,P, Type, S) ←
     I = #(Cn:Type):value(P',V',Value) ∧
     P = #(-value:type: list(class))
```

10.7.19.3.9 (op) vClass: instance

```
has(#vClass:I,S,I,Instance, I:value(P,V,Value), S) ←
     Instance = #(-vClass:instance: I)
```

10.7.20 Visibility

A private responsibility is visible only to an instance of the class within a responsibility of the class. A protected responsibility is visible only to an instance of the class within a responsibility of the class or a subclass of the class.

To determine for which classes `Cn:PnT` is visible for a message `I has Pn:V` requires knowing the sender class and instance. For a message issued within a responsibility with the head `Cn: Self has PV`, the sender is `#Cn: Self`. Let `Cs:Is` be the sender. Then for a message `I has PnV` bound to a responsibility `R`, `R` is visible if one of three conditions are satisfied.

a) The property is public.

```
visible(Cs:Is,S,I,R) ←
     fact(S,R,#(responsibility:visibility:object),public)
```

b) The property is private and
the sender class is the same, `Cs = #Cn`, and the sender instance is the same, `Is = I`.

```
visible(#Cn:I,S,I,R) ←
     fact(S,R,#(responsibility:visibility:object),private)∧
     (R = #(Cn:PnT) ∨ R = #(-Cn:PnT))
```

The sender class is the same, `Cs = #Cn`, and the sender instance is the class, `Is = #Cn`.

```
visible(#Cn:#Cn,S,I,R) ←
     fact(S,R,#(responsibility:visibility:object),private)∧
     (R = #(Cn:PnT) ∨ R = #(-Cn:PnT))
```

The sender class is the metaclass of the class, `#Cn` ε_S`l Cs`, and the sender instance is the class or a subclass of the class, `Is isa* #Cn`.

```
visible(Cs:Is,S,I,R) ←
     fact(S,R,#(responsibility:visibility:object),private)∧
     (R = #(Cn:PnT) ∨ R = #(-Cn:PnT)) ∧
     #Cn εSl Cs
     Is isa* #Cn
```

c) The property is protected and
the sender instance is the same, `Is = I`.

```
visible(Cs:I,S,I,R) ←
     fact(S,R,#(responsibility:visibility:object),protected)
```

The sender class is the same or a subclass, `Cs isa* #Cn`.

```
visible(Cs:Is,S,I,R) ←
     fact(S,R,#(responsibility:visibility:object),protected)∧
     (R = #(Cn:PnT) ∨ R = #(-Cn:PnT)) ∧
     Cs isa* #Cn
```

The sender class is the metaclass of the class, `#Cn` ε_S`l Cs`.

```
visible(Cs:Is,S,I,R) ←
     fact(S,R,#(responsibility:visibility:object),protected)∧
     (R = #(Cn:PnT) ∨ R = #(-Cn:PnT)) ∧
     #Cn εSl Cs
```

10.7.21 Dynamic binding

Dynamic binding uses the inheritance search order to determine the realization `R` to be used for a message. Properties not visible by the visibility rules are ignored as though they were not present.

The axioms for binding ensure that `R` matches in name, level, property operator, and types; is reachable from `I`; is above any floor set by a `send to super`; is not lower than the responsibility `R` for a direct message; and is the lowest such responsibility in the inheritance order. Constraints in the metamodel ensure that there is at most one such responsibility. The vocabulary consists of the following:

`Cs`	is the sender class
`Is`	is the sender instance
`S`	is the input state
`I`	is the nominal receiver
`PnV`	is `Pn:V, Pn:=V, Pn:!=V, Pn:+=V, or Pn:-=V`
`RV`	is `R:V`, where `R` is a responsibility
`IRs`	is the set of reachable, matching responsibilities, including `R`, with their receiver, `I′`
`I′`	is the actual receiver.
`R`	is the selected responsibility

The axioms for dynamic binding are given in 10.7.21.1 through 10.7.21.7.

10.7.21.1 Bind

For the message `I` has `PnV`, bind finds

— The responsibility `R`,
— The overridden instance responsibility pairs `IRs`,
— The instance `I′`, and
— The value `V`

so that `I′` has `R:V`.

```
bind(Cs:Is,S,I,PnV,POn,IRs,I′,R,V,T) ←
     parsePnV(PnV,Pn,POn,V,QPnT) ∧ ~(Pn = #_) ∧
     IRs = {I′:[R′,I′,LC,Pl,L,#Cn,T] where
          match(Cs:Is,S,I,QPnT,V,R′,Pn,T,L,Pl) ∧
          reach(S,I,LC,#Cn,I′) ∧
          if floor(S,I,LCf,Plf,Lf,Cf)
          then
               lessThan(S,[LCf,Plf,Lf,Cf,object],
               [LC,Pl,L,#Cn,object])
          endif } ∧
     if IRs = {} then exception(‘property not found ’,QPnT) endif ∧
     minimum(S,IRs,[R,I′,LC,Pl,L,#Cn,T])

parsePnV(Pn: V, Pn, get,             V, Cn:Pn: T)
parsePnV(Pn:= V, Pn, set,            V, Cn:Pn:= T)
parsePnV(Pn:!=V, Pn, unset,          V, Cn:Pn:!=T)
parsePnV(Pn:+=V, Pn, add,            V, Cn:Pn:+=T)
parsePnV(Pn:-=V, Pn, remove,         V, Cn:Pn:-=T)

bind(Cs:Is,S,I,RV,POn,IRs,I,R,V,T) ←
     parseRV(RV, R,Pn,POn,T,V,QPnT,Cf) ∧
reach(S, I,LCf, Cf, I) ∧
fact(S,R,#(responsibility:plicity:object),Plf) ∧
fact(S,R,#(responsibility:level:object),Lf) ∧
IRs = {I′:[R′,I′,LC,Pl,L,#Cn,T] where
     match(Cs:Is,S,I,QPnT,V,R′,Pn,T,L,Pl) ∧
```

```
          reach(S,I,LC,#Cn,I') ∧
          ~lessThan(S, [LC,Pl,L,#Cn,object] , [LCf,Plf,Lf,Cf,object]) }

     parseRV(#(Cn:Pn:T) : V, Pn, get,      T, V, Cn':Pn:  T', #Cn)
     parseRV(#(Cn:Pn:=T) : V, Pn, set,     T, V, Cn':Pn:= T', #Cn)
     parseRV(#(Cn:Pn:!=T): V, Pn, unset,   T, V, Cn':Pn:!= T', #Cn)
     parseRV(#(Cn:Pn:+=T): V, Pn, add,     T, V, Cn':Pn:+= T', #Cn)
     parseRV(#(Cn:Pn:-=T): V, Pn, remove,  T, V, Cn':Pn:-=T', #Cn)
```

10.7.21.2 Match

Match R in name, level, property operator, and types.

```
     R=#(Cn:PnT) for explicit responsibilities
     R=#(-Cn:PnT) for implicit responsibilities
     PnT = Pn PO T
     R has class: #Cn
     R has name: Pn
     R has level: L
     R has type: T
     forall +Ti in T and corresponding Vi in V, Vi ιs Ti.
```

Match explicit responsibilities.

```
     match(Cs:Is, S, I, Cn:PnT, V, R, Pn, L, explicit, T) ←
          R εS #responsibility ∧
          R = #(Cn:PnT) ∧
          visible(Cs:Is,S,I,R) ∧
          fact(S,R,#(responsibility:isRealized:boolean),truet) ∧
          fact(S,R,#(responsibility:name:     object),Pn) ∧
          fact(S,R,#(responsibility:level:    object),L) ∧
          fact(S,R,#(responsibility:plicity:  object),explicit) ∧
          fact(S,R,#(responsibility:type:     object),T) ∧
          accept(S,V,T)
```

Match implicit responsibilities on `Pn` and `T`, excluding `add` and `remove` for collections.

```
     match(Cs:Is, S, I, Cn:PnT, V, R, Pn, L, implicit, T) ←
          R εS #responsibility ∧
          parsePnV(PnT,Pn,POn,T,_) ∧
          R = #(-Cn:Pn:T) ∧
          ~(   (POn = add ∨ POn = remove)∧
               (T = T' ∨ T = +T') ∧
               T' isa* collection(T) ∧
          visible(Cs:Is,S,I,R) ∧
          fact(S,R,#(responsibility:level:object),L) ∧
          fact(S,R,#(responsibility:plicity:object),implicit) ∧
          accept(S,V,T)
```

Match implicit responsibilities on `Pn:Cn(T)`, add and remove for collections.

```
     match(Cs:Is, S, I, Cn:PnT, V, R, Pn, L, implicit, T) ←
          R εS #responsibility ∧
```

```
parsePnV(PnT,Pn,POn,T,_) ∧
(POn = add ∨ POn = remove)∧
(R = #(-Cn:Pn:T’) ∨ R = #(-Cn:Pn:+T’)) ∧
T’ isa* collection(T) ∧
visible(Cs:Is,S,I,R) ∧
fact(S,R,#(responsibility:level:object),L) ∧
fact(S,R,#(responsibility:plicity:object),implicit) ∧
accept(S,V,T)
```

10.7.21.3 Accept

Accept value V as type T.

```
accept(S,V,T) ← (T=+T’ ∨ T=T’) ∧ V $\iota_S$ T’
accept(S,[],[])
accept(S,[V|Vs],[T|Ts]) ← accept(S,V,T) ∧ accept(S,Vs,Ts)
```

10.7.21.4 Reach

Class C must be reachable from the nominal receiver along ε_Sl and isa$_S$ relations. If an ε_Sl relation is taken to a lowclass, the lowclass becomes the actual receiver (see Table 28).

Table 28—Reachability

Nominal receiver	Reachable class C	Actual receiver
I	I ε_Sl+ LC ∧ LC isa$_S$* C	if I ε_Sl LC then I else LC
super(I:#Cn)	I ε_Sl+ LC ∧ LC isa$_S$* C	if I ε_Sl LC then I else LC
C’	(C’ = LC ∨ C’ ε_Sl+ LC) ∧ LC isa$_S$* C	LC
super(C’:#Cn)	(C’ = LC ∨ C’ ε_Sl+ LC) ∧ LC isa$_S$* C	LC

```
reach(S, I, LC, C, I) ← ~(I $\varepsilon_S$ #class) ∧ I $\varepsilon_S$l LC ∧ LC isa$_S$* C
reach(S, I, LC, C, LC) ← ~(I $\varepsilon_S$ #class) ∧ I $\varepsilon_S$l C ∧ C $\varepsilon_S$l+ LC ∧ LC isa$_S$* C
reach(S, C’,LC, C, LC) ← (C’ $\varepsilon_S$ #class) ∧ (C’ = LC ∨ C’ $\varepsilon_S$l+ LC) ∧ LC
isa$_S$* C
reach(S, super(X:#Cn), LC, C, X’) ← reach(S, X, LC, C, X’)
```

10.7.21.5 Floor

For a `send to super`, the property bound must be above the property that issued the `send to super`. The lowclass, `LC`, is the one with which reach starts. The sender is always explicit. The level is the sender's level. The class is `Cn`, the class from which the `send to super` was issued (see Table 29).

Table 29—Floor

Receiver	Floor lowclass, level, class
super(I:#Cn)	I ε_Sl LC, instance, #Cn
super(C′:#Cn)	C′, class, #Cn

```
floor(S, super(I:#Cn), LC, explicit, instance, #Cn) ←
     ~(I ε_S #class) ∧ I ε_Sl LC

floor(S, super(C′:#Cn), C′, explicit, class, #Cn) ← C′ ε_S #class
```

10.7.21.6 lessthan

The inheritance order is ascending on lowclass `LC`, plicity, level, class, and type, where

```
LC < LC′ if LC ε_Sl+ LC′
explicit < implicit
instance < class
C < C′ if C isa_S+ C′
T < T′ if T <: T′ and not(T = T′)
```

The same order determines which responsibilities are above the floor.

```
lessThan(S, [LC,Pl,L,C,T], [LC′,Pl′,L′,C′,T′])←
     -- R is less than R′…
     LC ε_Sl+ LC′ -- lower lowClass
     ∨ LC=LC′ ∧ -- or same super classes and
          (Pl < Pl′ -- lower plicity
          ∨ Pl = Pl′ ∧ -- or same plicity and
               (L < L′ -- lower level
               ∨ L = L′ ∧-- or same level and
               (C isa_S+ C′ -- lower class
               ∨ C=C′ ∧ -- or same class and
                      T <: T′ ∧ ~(T=T′))))--lower type
```

10.7.21.7 Minimum

For any responsibility `R′` that matches, is reachable, and is above the floor, `R = R′` or `R` is less than `R′`

```
minimum(S, IRs, [R,I,LC,Pl,L,C,T])←
     I:[R,I,LC,Pl,L,C,T,] ∈ IRs ∧
     ∀(R′,I′,LC′,Pl′,L′,C′,T′)(
          if I′:[R′I′,LC′,Pl′,L′,C′,T′] ∈ Rs ∧ not ( R = R′)
```

```
            then        -- R is less than R'…
                 lessThan(S, [LC,Pl,L,C,T], [LC',Pl',L',C',T'])
            endif)
```

10.7.22 Pre- and post-conditions

A property can have any number of pre-conditions or post-conditions. During the rewrite to definition clausal form, separate clauses are produced for the property and the pre-conditions and post-conditions. The pre-condition is a disjunction of the specified `pre Sentences`. The post-condition is a conjunction of the specified `post Sentences`.

```
preOk(Si, IRs, V, T) ←
     build(V,T,Vi) ∧ I:R ∈ IRs, pre(R,Rpre) ∧ has(_:I, Si, I, Rpre,
     Vi, Si)

pre( #(Cn:Pn:T),  #(Cn:(-Pn):T) )
pre( #(Cn:Pn:=T), #(Cn:(-Pn):=T) )
pre( #(Cn:Pn:!=T), #(Cn:(-Pn):!=T) )
pre( #(Cn:Pn:+=T), #(Cn:(-Pn):+=T) )
pre( #(Cn:Pn:-=T), #(Cn:(-Pn):-=T) )

postOk(Si, IRs, V, So) ←
     ~∃(I,R,Rpost)(I:R ∈ IRs ∧ post(R,Rpost) ∧ ~has(_:I, Si, I,
     Rpost, V, S))

post( #(Cn:Pn:T),  #(Cn:(+Pn):T) )
post( #(Cn:Pn:=T), #(Cn:(+Pn):=T) )
post( #(Cn:Pn:!=T), #(Cn:(+Pn):!=T) )
post( #(Cn:Pn:+=T), #(Cn:(+Pn):+=T) )
post( #(Cn:Pn:-=T), #(Cn:(+Pn):-=T) )
```

10.7.23 Constraints

It is up to the modeler to construct the model, i.e., theory, so that for all public messages within the intended interpretation, the message terminates, does not raise an exception, and produces the correct answer. Ultimately, only the modeler knows the intended interpretation. In order to construct a theory for the intended interpretation, the modeler must know the effect the constraints have on the theory. The purpose of formalizing the constraints is to state their effect on the theory.

A constraint holds only for specific states and messages. Some constraints, such as a read-only constraint, are expected to hold in all states. Other constraints, such as total, are not expected to hold in temporary, intermediate states. A constraint in the graphics and RCL gives the constraint condition and indicates the states and messages for which it must hold. Table 30 gives the intended meaning of the constraints.

Checking a named constraint that is not constant and has no effective pre-condition requires issuing a message in the appropriate pre-conditions, post-conditions, or assertions. For example, the post-condition on the outermost updating operation should check all constraints dependent on any property that may have been updated.

Table 30—Constraints and their meanings

Constraint	Quantification	Condition that must be true	Comment
Read-only responsibility	All states, all argument values	Input state equals output state.	All named constraints and uniqueness constraints are read-only.
Nonupdateable argument	All states, all argument values	The facts about the argument are the same in the input state and the output state.	—
Constant responsibility	All states, all argument values	Precondition, if any. Cardinality constraint (total, function, cardinality `N`). Post-condition, if any. Same output values for given input values.	—
Responsibility with an effective pre-condition	All states, all argument values for which the effective pre-condition is true	If the responsibility is true, then the effective post-condition, if any, must be true.	—
Responsibility with an effective pre-condition and total	All states, all argument values for which the effective pre-condition is true	Responsibility. Effective post-condition, if any.	A named constraint with an effective recondition is covered by this case.
Uniqueness constraint	All states	No two instances agree on all properties in the uniqueness constraint.	—

Table 31 describes the constraints that are checked as part of the dynamic binding axiom for each message sent.

Table 31—Constraints checked by dynamic binding axiom

Constraint on responsibility	Property operator	Condition that must be true	Comment
Total and no effective pre-condition	`All`	Message must be true.	This case includes named constraints.
Responsibility with an effective precondition	`All`	Effective pre-condition. If the responsibility is true, then the effective post-condition, if any, must be true.	—
Responsibility with an effective precondition and total	`All`	Effective pre-condition. Responsibility. Effective post-condition, if any.	—
Function, i.e., single valued	`All`	At most one solution.	—
Single-valued collection property, collection constrained to cardinality `N`	`Get`	If property has a value, the collection count is `N`	—
Multi-valued property, cardinality `N`	`Get`	Exactly `N` solutions	—

The constraints checked as a part of the dynamic binding axiom are entirely redundant if pre-conditions and post-conditions are fully utilized.

10.7.23.1 CardinalityOk

The cardinality predicate raises an exception if a total, function, or cardinality `N` constraint is not met (see Table 32).

Table 32—Cardianlity constraints

Plicity	`POn`	Check total	Check function	Check cardinality `N`	Comment
Implicit	`get`	Yes	Yes	If read-only	Override the explicit to whom delegated (which is unconstrained).
Explicit	`get`	Yes	Yes	If read-only	—
Implicit	Other	No	No	No	Defer to the explicit to whom delegated.
Explicit	Other	Yes	Yes	No	—

```
cardinalityOk(CI, S, I, POn, R, V, T) ←
     (R = #(Cn:PnT) ∨ R = #(-Cn:PnT) ∧ POn = get )∧
     cardinalitySolutions(CI, S, I, POn, R, V, T, Solutions, Cnt) ∧
     if fact(S,R,#(responsibility:isTotal:boolean), true_t)
     then
          if Cnt = 0 then exception('not total ',R) endif
     endif ∧
     if fact(S,R,#(responsibility:isFunction:boolean), true_t)
     then
          if Cnt > 1 then exception('not function ',R) endif
     endif ∧
     if   POn = get ∧
          fact(S,R,#(responsibility:isReadOnly:boolean), true_t) ∧
          fact(S,R,#(responsibility:cardinalityN:integer),CardN)
     then
          if   ~(∃(T')(T isa* collection(T')) ∧
               ~(Cnt = CardN)
               ∨
          (T isa* collection(T')) ∧
          Cnt = 1 ∧
          Solutions = [Collection:So] ∧
          fact(Collection,#(collection:list:list(object)),List) ∧
          ~count(List,CardN)
     then
          exception('not cardinality N', R)
     endif
endif
```

10.7.23.2 Cardinality solutions

```
cardinalitySolutions(CI,S, I, POn, R, V, T, Solutions, Cnt) ←
```

```
        isList(Solutions) ∧
        ∀(V’,So)
            (if build(V,T,V’) ∧ issue(CI,S,I,POn,R,V’,So)
            then V’:So ∈ Solutions
            endif) ∧
        ∀(V’,So)
            (if build(V,T,V’) ∧ V’:So ∈ Solutions
            then issue(CI,S,I,POn,R,V’,So))
            endif) ∧
        noDup(Solutions) ∧
        count(Solutions,Cnt)
```

10.7.23.3 Build

Build up a new value for the arguments, keeping the input values and using new, existentially quantified variables for the output values.

```
    build(V,+T,V) ← ~isList(T)
    build(V,T,V’) ← ~isList(T)
    build([],[],[])
    build([V|Vs],[+T|Ts],[V|Vs’]) ← build(Vs,Ts,Vs’)
    build([V|Vs],[T|Ts],[V’|Vs’]) ← build(Vs,Ts,Vs’)
```

10.7.24 Exceptions

```
    false ← exception(R,X)
```

due to the closed world assumption. The result is that every *model* has an empty relation assigned to `exception`.

10.8 Summary of the formal meaning of a view

The formal meaning of a view is defined as a first order theory with the vocabulary specified above and the following axioms.

a) The axioms for equality and the base theories.
b) The axioms for the modeling constructs.
c) The clauses resulting from the mapping of the RCL declarations and realizations for the metamodel.
d) The clauses resulting from the mapping of the RCL declarations and realizations for the view.
e) The completion of all the clauses.

Annex A

(informative)

Bibliography

This annex contains references that are directly cited in this standard or that directly influenced it.

[B1] Abadi, M. and Cardelli, L., *A Theory of Objects*. Springer-Verlag, 1996.

[B2] Apt, K. R., *From Logic Programming to Prolog*. Prentice Hall, 1997.

[B3] Brown, R. G., *Logical Database Design Techniques*. Mountain View, CA: The Database Design Group, Inc., 1982.

[B4] Brown, R. G., "The Evolution of Domains," *Database Newsletter*. Nov/Dec 1993.

[B5] Brown, R. G., *From IDEF1X to IDEF$_{object}$*. Prepared for the National Institute of Standards and Technology under contract 871-4052, 1995.

[B6] Brown, R. G., *IDEF1X$_{97}$ Rule and Constraint Language (RCL)*. 1997.

[B7] Brown, R. G. and Berzins, V., *IDEF1X$_{97}$ Formalization*. 1998.

[B8] Bruce, T. A., *Designing Quality Databases with IDEF1X Information Models*. Dorset House, 1992.

[B9] Castagna, G., "Covariance and Contravariance: Conflict without a Cause," *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 3. May 1995, pp. 431-447.

[B10] Castagna, G., *Object Oriented Programming, A Unified Foundation*. Birkhauser, 1997.

[B11] Cattell, R. G. G., ed., Atwood, T., ed., *The Object Database Standard: ODMG-93: Release 1.2*. Morgan Kaufmann, 1996.

[B12] Cleaveland, J. C., *An Introduction to Data Types*. Addison-Wesley, 1986.

[B13] FIPS PUB 184, *Integration Definition for Information Modeling (IDEF1X)*. Gaithersburg, MD: US Department of Commerce, National Institute of Standards and Technology (NIST), December 1993.[94]

[B14] Gamma, E., *et al*, *Design Patterns*. Addison-Wesley Publishing Company, 1995.

[B15] General Electric Company, *Integrated Information Support System (IISS), Vol. V-Common Data Model Subsystem, Part IV-Information Modeling Manual-IDEF1 Extended*. AFWALTR864006, Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, WrightPatterson Air Force Base, Ohio 45433, November 1985.

[B16] Hogger, C. J., *Introduction to Logic Programming*. Academic Press, 1984.

[94] FIPS publications are available from the National Technical Information Service (NTIS), US Department of Commerce, 5282 Port Royal Rd., Springfield, VA 22161 USA.

[B17] ISO Technical Report 9007, *Information Processing Systems—Concepts and Terminology for the Conceptual Schema and the Information Base*. 1987.

[B18] Khoshafian, S., and Abnous, R., *Object Orientation: Concepts, Languages, Databases, User Interfaces*. John Wiley & Sons, Inc., 1990.

[B19] Liskov, B., "Data Abstraction and Hierarchy," *OOPSLA 87 Addendum to the Proceedings*. 1987.

[B20] Lloyd, J. W., *Foundations of Logic Programming*. Springer-Verlag, 1984.

[B21] Maier, D., and Zdonik, S. B., "Fundamentals of Object-Oriented Databases," *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.

[B22] Manna, Z., and Waldinger, R., *The Logical Basis for Computer Programming, Volume 1: Deductive Reasoning*. Addison-Wesley, 1985.

[B23] SofTech, *ICAM Architecture Part II—Volume V—Information Modeling Manual (IDEF1)*. AFWALTR814023, Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, WrightPatterson Air Force Base, Ohio 45433, June 1981.

[B24] Spivey, M., *An Introduction to Logic Programming through Prolog*. Prentice Hall, 1996.

[B25] Stonebraker, M., *Object-Relational DBMSs*. Morgan Kaufmann, 1996.

[B26] *Merriam-Webster's Collegiate Dictionary, 10th Edition.* Springfield, MA: Merriam-Webster, Inc., 1995.

Annex B

(informative)

Comparison of IDEF1X$_{93}$ and IDEF1X$_{97}$ constructs

B.1 Migration from data models to object models

IDEF1X$_{97}$ is a new modeling style, not simply an extension of IDEF1X$_{93}$. Clause 9 has shown how the full identity-style language can be restricted to provide a key style compatible with earlier versions. While it is intended that the new language be used by IDEF1X practitioners, it is recognized that a transition period will occur, with some modelers practicing solely in the old style, some practicing solely in the new, and others experimenting with the new capabilities but not yet ready to adopt them completely.

Many of the features of the identity style can be used informally in conjunction with key-style models as a way to learn how to use them. For example, adopting the notion of intrinsic instance identity, while retaining the use of primary and foreign keys, allows the modeler to describe constraints precisely, as an alternative to the imprecise notes used with the key style. In other words,, notes may be written in "informal" RCL as well as natural language. Thinking of domains as value classes, or emulating them with entities, opens up additional possibilities for sharing and reuse of data.

A view is to be clearly of one type, identity style or key style. This standard does not sanction blending of the styles, or specify the meanings of constructs for "hybrid" models. Use of such models as learning tools during the period of transition to full identity style may, however, be helpful to some.

B.2 Comparison of modeling constructs

The fundamental semantic constructs of IDEF1X$_{93}$ FIPS PUB 184 [B13] are part of IDEF1X$_{97}$. The earlier concepts of IDEF1X have been incorporated into IDEF1X$_{97}$ by

a) Relaxing some of the restrictions in IDEF1X$_{93}$.
b) Exploiting the fundamental concepts more fully.
c) Adding some important new concepts.

The identity-style version of IDEF1X$_{97}$ described in Clauses 5 through 8 of this standard is the result of carrying out these steps. A summary of the correspondence of the IDEF1X$_{93}$ and identity-style IDEF1X$_{97}$ constructs and terminology is provided in Tables B.1 through B.3.

Table B.1—Corresponding constructs

IDEF1X$_{93}$ construct name	Identity-style IDEF1X$_{97}$ subsuming construct	Comments
alternate key	uniqueness constraint	Essentially the same concept.
attribute	attribute	Essentially the same concept. In IDEF1X$_{93}$, attributes typed to domains rather than value classes.
category cluster	subclass cluster	Essentially the same concept. There are subtle differences in the treatment of a “total cluster” as an “abstract class.”
domain	value class	A value class is a class in its own right. Its representation is hidden, allowing attributes typed to it to have their representations hidden.
entity (dependent entity)	dependent state class	A dependent state class is one that is by its very nature *intrinsically* related to certain other state class(es).
entity (independent entity)	state class	Classes have both knowledge and behavior properties. Classes can also have class-level properties specified.
entity instance	instance (object)	An instance is a member of the like things represented by the class.
generalization hierarchy	generalization taxonomy	Generalization allows for inheritance of properties. The identity style of IDEF1X$_{97}$ includes multiple inheritance.
glossary	glossary	Essentially the same concept.
model	model	Essentially the same concept.
note	note	Notes remain as a flexible way of writing comments or informally stating constraints.
relationship	relationship	Essentially the same concept. In IDEF1X$_{93}$, a relationship is expressed in terms of foreign keys where the foreign key identifies an instance in a particular entity class. In IDEF1X$_{97}$, relationships are specified in terms of identity and participant properties; the relationship participant property identifies an instance without regard to class.
view	view	Essentially the same concept.
view level	view level	Essentially the same concept.

Table B.2—IDEF1X$_{93}$ constructs new in IDEF1X$_{97}$

IDEF1X$_{93}$ construct name	Identity-style IDEF1X$_{97}$ subsuming construct	Comments
(no corresponding construct)	class-level property	In IDEF1X$_{93}$, entities did not have attributes; only their instances did. In identity-style IDEF1X$_{97}$ classes, as well as instances, may have properties.
(no corresponding construct)	constraint/specification language (RCL)	Many constraints could not be stated directly in IDEF1X$_{93}$ and were, therefore, recorded as notes. RCL is available to precisely state constraints in identity-style IDEF1X$_{97}$.
(no corresponding construct)	environment	There was no corresponding concept described in IDEF1X$_{93}$.
(no corresponding construct)	instance identity	An instance in identity-style IDEF1X$_{97}$ possesses its own intrinsic identity, distinct from all other instances. Primary keys were used to represent identity in IDEF1X$_{93}$, but this identity was explicitly declared, not intrinsic.
(no corresponding construct)	operation	There was no corresponding concept described in IDEF1X$_{93}$.
(no corresponding construct)	participant property	In IDEF1X$_{97}$, the value of a participant property is the identity of the related instance. IDEF1X$_{93}$ did not support the notion of identity and thus could not support this construct.
(no corresponding construct)	subject domain	There was no corresponding concept described in IDEF1X$_{93}$.

Table B.3—IDEF1X$_{93}$ constructs not in IDEF1X$_{97}$ (identity style)

IDEF1X$_{93}$ construct name	Identity-style IDEF1X$_{97}$ subsuming construct	Comments
foreign key	(no directly corresponding construct)	Foreign keys are not used in the identity style of IDEF1X$_{97}$.
primary key	(no directly corresponding construct)	In IDEF1X$_{97}$ instances have intrinsic identity. The explicit uniqueness constraint inherent in an IDEF1X$_{93}$ primary key can be stated with a uniqueness constraint in identity-style IDEF1X$_{97}$.

Annex C

(informative)

Examples

C.1 Pattern example: composite

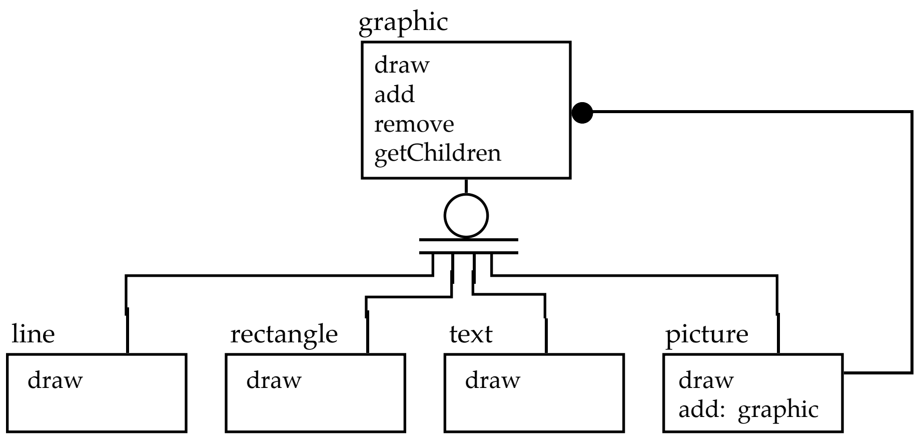
This set of examples illustrates how IDEF1X can be mapped to design patterns or templates. A *pattern*, in general, has been defined as something that "describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over...."[95] A *design pattern* has been defined as a description of "communicating objects and classes ... customized to solve a general design problem." [B14]

Design patterns can be useful constructs for building reusable objects or for building new solutions that did not have to start from scratch. Obviously, to accomplish this feat, one must find or create an appropriate design pattern. When using a design pattern, it is important to think in abstract terms in order to determine applicability. To this end, the next two examples use design patterns from [B14], as they would be represented in the IDEF1X language. A third example (in C.3) combines the first two patterns and applies that aggregate pattern to represent the classes and properties of a familiar business context.

The first pattern is called the "composite" design pattern.[96] It uses a tree structure representing a part-whole hierarchy. In IDEF1X terms, Figure C.1 uses a cluster (here, a total cluster) to depict the various subclasses of `graphic`. The distinguishing aspect of this pattern is the use of recursion in the `picture` class. This class represents a more complex class than the other subclasses in the cluster (`line`, `rectangle`, and `text`,).

The purpose of this pattern is to specify an abstract `graphic` class that is capable of treating simple graphics (such as lines and text) with the same ease as a more complex picture, which may be composed of many lines, shapes, text, and even other pictures. Conceptually, the depth of the part-whole hierarchy is unlimited. Therefore, a `picture` could be made up of other pictures that, in turn, could include various shapes and possibly more pictures.

[95]Christopher Alexander, speaking of patterns in architecture, as quoted in *Design Patterns* [B14], p. 2.
[96]See *Design Patterns* [B14], p. 163-173, for a full discussion of the "composite" pattern.



```
picture: Self has draw ifdef
     forall (Self has graphic: G): (G has draw).

picture: Self has add: G ifdef -- add instance of graphic
     Self has graphic:+= G.   -- to a picture
```

Figure C.1—Composite pattern example

C.2 Pattern example: state

The second pattern illustrated is called the "state" design pattern.[97] It allows an instance to appear to change its class (subclass) when it needs to alter its behavior as the result of a state change. This was introduced in the discussions of "changing specialization" in 5.4.1.11 and 7.3.6.[98]

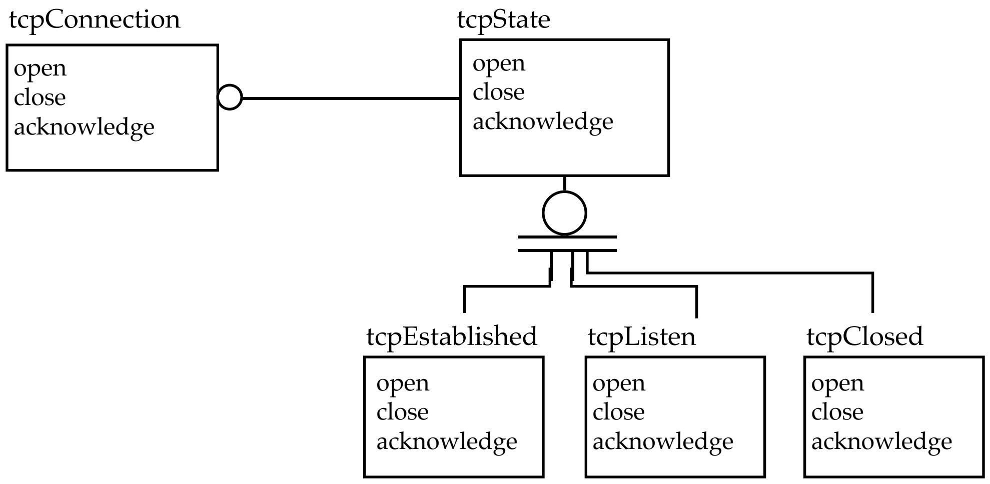
The [B14] discussion of this pattern uses a network (TCP) connection. In this example, an instance of a connection (`tcpConnection`) can be in one of three states: established, listening, or closed. When the connection instance receives a request to `open`, `close`, or `acknowledge`, it needs to respond differently, depending on its current state. The "state" pattern describes how an instance can appear to change its specialization (to have differing behaviors) without actually altering its identity.

The key idea in this pattern is the introduction of another, abstract class (here called `tcpState`) to represent the various states of the instance. This class declares an interface for each of the requests calling for specialization-dependent behavior; its subclasses then implement the behaviors.

An instance of original class (here, `tcpConnection`) is always related to the one instance of the abstract class that represents its current state. When an instance of `tcpConnection` receives a request for one of the state-specific behaviors, it simply delegates the request to its current state instance, as shown in the RCL in Figure C.2. Whenever the instance changes state, it does so by changing its relationship to the appropriate state instance.

[97]See *Design Patterns* [B14], p. 305-313, for a full discussion of the "state" pattern.

[98]IDEF1X directly supports respecialization. An alternative conceptual level solution would be simply to make the states subclasses of `tcpConnection` and eliminate `tcpState` as a separate class. However, at the implementation level, many programming languages do not allow an instance to "change its class" (respecialize) so a solution of this nature is needed. The pattern is also applicable in cases where a programming language does not allow a class to have multiple clusters of subclasses.



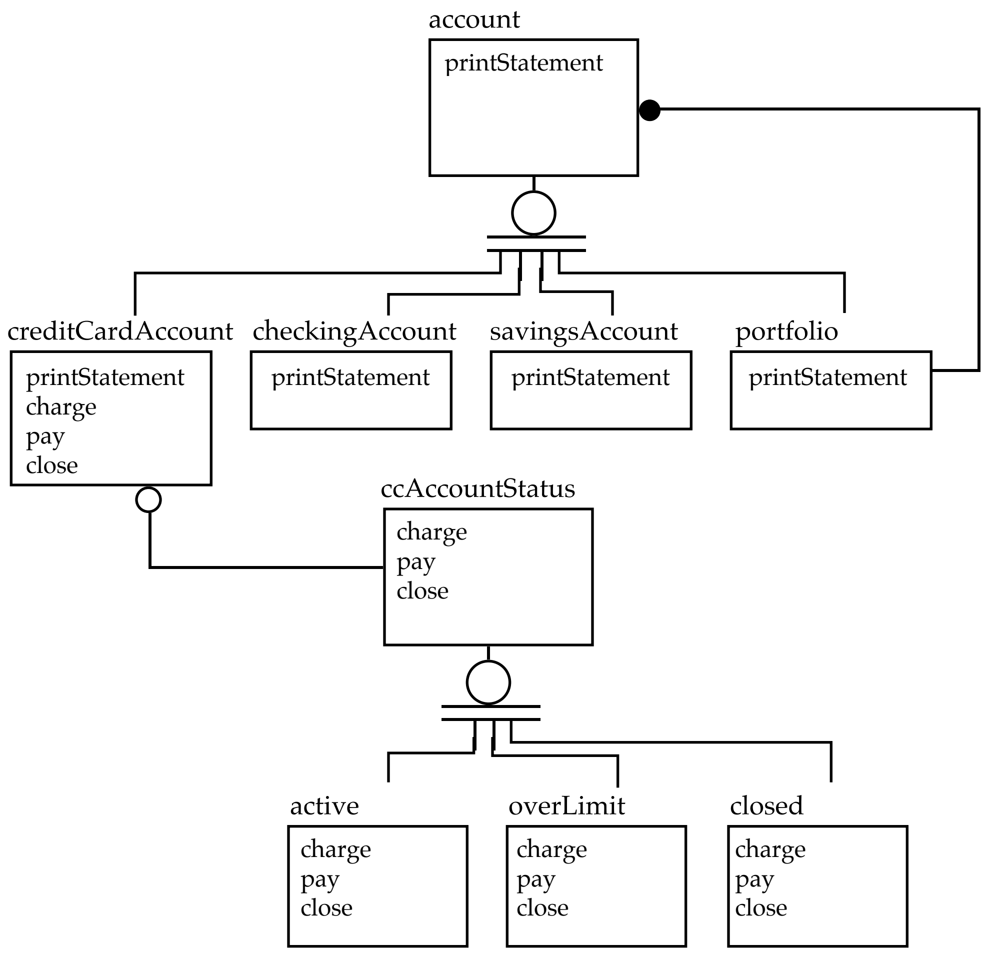
```
tcpConnection: Self has open if_def
     Self has tcpState..open.

tcpConnection: Self has close if_def
     Self has tcpState..close.

tcpConnection: Self has acknowledge if_def
     Self has tcpState..acknowledge.
```

Figure C.2—State pattern example

C.3 Business pattern example (combining two patterns)

This pattern combines the two patterns introduced in C.1 and C.2. It applies that resulting aggregate pattern to represent the classes and properties of a familiar business context, a bank account. In Figure C.3, the "composite" pattern is used to represent the part-whole hierarchy of a portfolio and its component accounts of various types (credit card account, checking account, savings account, and possibly other portfolios). Each basic account can print its own individual statement, and (as shown in the RCL) the portfolio's `printStatement` results in a combined statement for its constituent accounts.

The `creditCardAccount` portion of the model illustrates the application of the "state" pattern. In this example, a credit card account can be in one of three states: active, over its limit, or closed. How an instance responds to a request to `charge`, `pay`, or `close` depends on its current state. For example, a request to `charge` when the `creditCardAccount` is (related to) `active` will respond with the expected behavior (and apply the charge to the account balance). However, if the credit card account is (related to) `overLimit`, the request to `charge` will be denied (if that is the business rule).



```
portfolio: Self has printStatement if_def
      forall (Self has account: A): (A has printStatement).

creditCardAccount: Self has charge if_def
      Self has ccAccountStatus..charge.
```

Figure C.3—Business pattern example, combining composite and state patterns

C.4 Model view level examples

Figure C.4 illustrates how a set of complementary views can depict the major concepts of interest to an area of a (hypothetical) business. In this way, the defined concepts presented in the views serve as a graphic representation of the business's vocabulary. Because this diagram is intended as illustrative of the concepts, only simple names for views and classes are shown.

Here a survey level view is shown for the subject domain, Sales Campaign. Responsibilities are displayed for one of its contained subject domains, Promotion. Also illustrated are the initial integration-level views for the four subject domains of a sales campaign. Each of these integration-level views depicts the classes most significant to that subject domain.

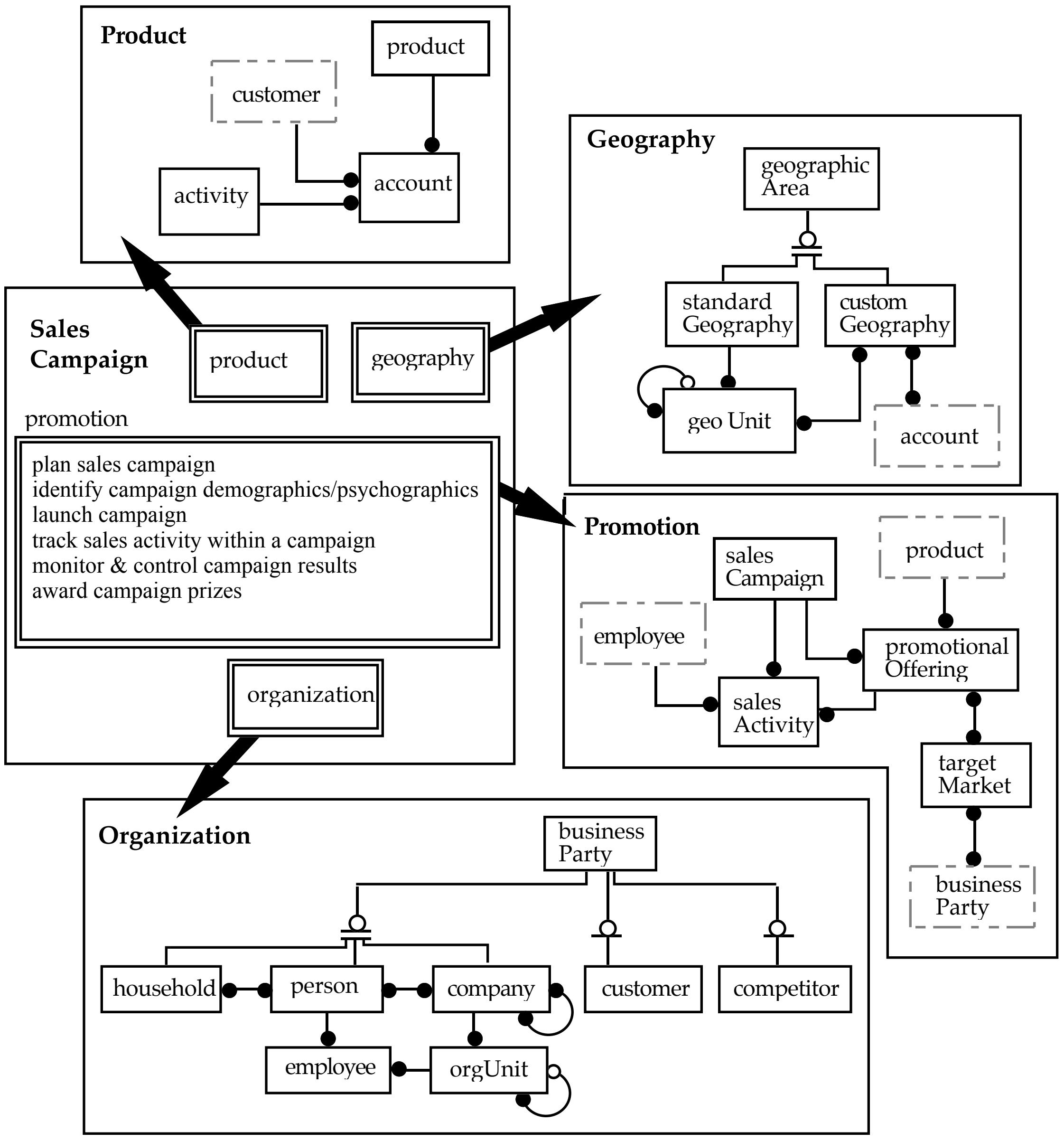


Figure C.4—Example set of model view levels

C.5 Behavior patterns

This set of examples illustrates how IDEF$_{object}$ can specify typical patterns of update behavior. This material uses a set of classes and properties drawn from the case study for HiHo, a fictitious recruitment agency. Knowledge of the full case study is not required for understanding of the patterns presented here since the focus is on typical patterns of behavior found in many application domains.

C.5.1 Coordinated change to attribute values

The first behavior pattern illustrates how the value of one attribute can be tied automatically to the change in another attribute's value. For example, HiHo specifies that when the placement consultant assigned to an applicant changes, that applicant's review date must be set to "30 days from now." The attribute properties involved in this requirement are shown in Figure C.5.

HiHo's requirement statement: "When an applicant's placement consultant is changed, the review date must be set to 30 days from today."

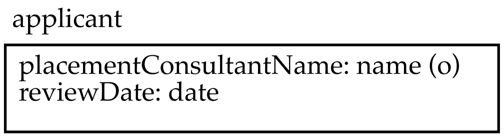


Figure C.5—Applicant's attributes

The placement consultant alone can be changed by making the simple request:

```
APP has placementConsultantName:= NewName
```

where

`APP` is the applicant in question, and
`NewName` is the name of the newly assigned placement consultant.

However, to enforce the requirement that the date be reset every time the placement consultant changes, the default "set value" for `placementConsultantName` needs to be overridden, with two alternative forms of the realization shown graphically in Figure C.6. Figure C.7 depicts the value class used in both forms of the realization.

```
  applicant: Self has placementConsultantName:= NewName if_def
    Self has placementConsultantName: CurrentName,
    if NewName == CurrentName
    then
           false
    else
           #date has today: TD,       -- request to the date value class
           TD has add: [ 30, ND ],    -- calculate today+30
           Self has reviewDate:= ND,  -- set to calculated value
           Self super has placementConsultantName:= NewName
    endif.
```

Alternative:

```
  applicant: Self has placementConsultantName:= NewName if_def
    not Self has placementConsultantName: NewName,
    #date has today: TD,
    TD has add: [ 30, ND ],
    Self has reviewDate:= ND,
    Self super has placementConsultantName:= NewName
```

Figure C.6—Realization of override to `placementConsultantName` assignment

In this case, the realization checks that a change to the value of the placement consultant is being made since there are side effects (here, resetting the review date). If a nonchange is permitted to succeed (i.e., making a "change" of the placement consultant to its current value), the review date would, in effect, be reset for no apparent reason.

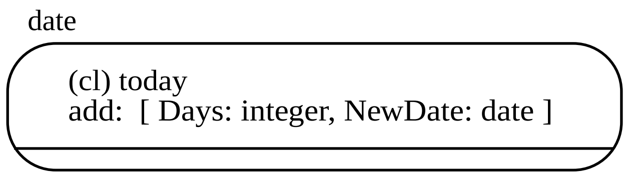


Figure C.7—Value class properties used in `assignPlacementConsultant` operation

C.5.2 Create an instance with a set of related instances

This behavior pattern is used when the existence of one instance requires "at least one" related other instance, e.g., when a relationship has a "positive" cardinality constraint, as shown by the "`P`" annotation in Figure C.8. Here an applicant who registers for job placement with HiHo is required to register at least one qualifying applicant skill.

HiHo's requirement statement: "When an applicant registers with us, the applicant must provide a set of their qualifications, i.e., the skill areas for which the applicant wishes to be presented for employment."

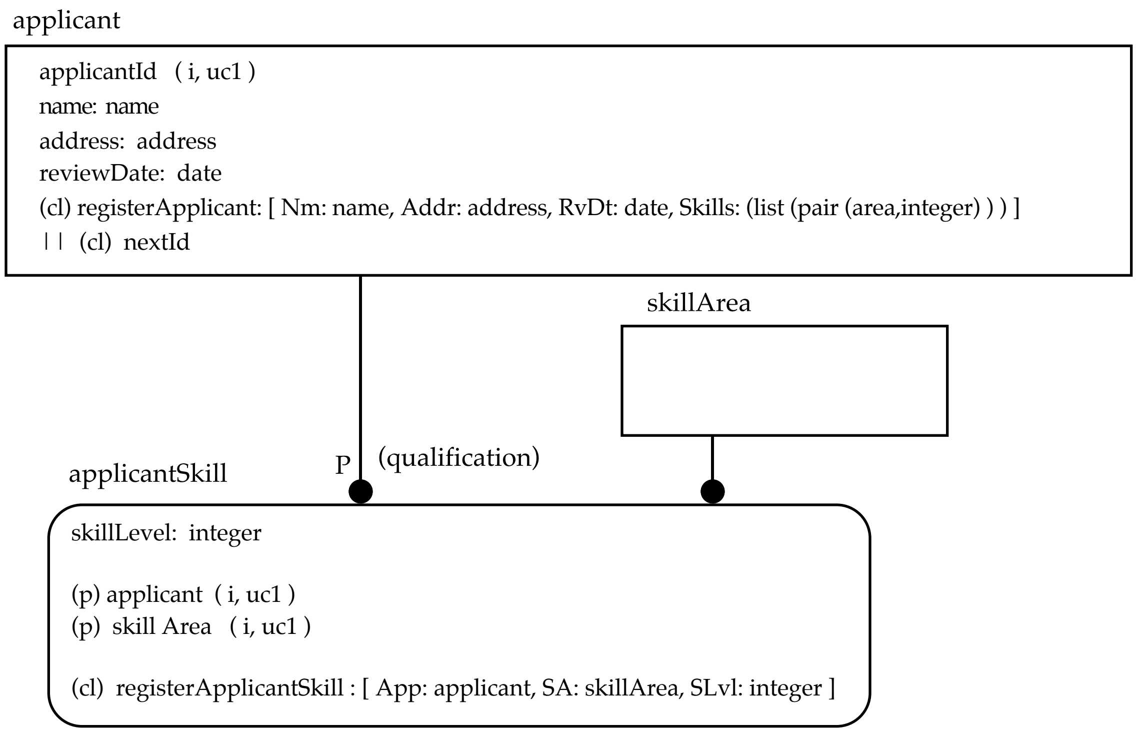

applicant
applicantId (i, uc1)
name: name
address: address
reviewDate: date
(cl) registerApplicant: [Nm: name, Addr: address, RvDt: date, Skills: (list (pair (area,integer)))]
| | (cl) nextId
skillArea
P
(qualification)
applicantSkill
skillLevel: integer
(p) applicant (i, uc1)
(p) skill Area (i, uc1)
(cl) registerApplicantSkill : [App: applicant, SA: skillArea, SLvl: integer]

Figure C.8—Value class properties used in `registerApplicant` operation

The applicant class has a class-level property `registerApplicant` that accepts the user's input values, requests and assigns an internally generated identifier (via another class-level property, `nextId`), and creates a new instance of applicant using the built-in `create` operation. The realization of `registerApplicant` is given in Figure C.9. This operation, in turn, invokes the class-level operation `registerApplicantSkill` in `applicantSkill`, shown in Figure C.10. This step will invoke the :+= override in `skillArea` (see C.5.4).

```
  applicant: Self has registerApplicant:
              [ Nm: name
              , Addr: address
              , RvDt: date
              , Skills: list ( pair ( skillArea, integer ) )
              ]                    if_def
           Self has nextId: Id,
           Self has create:
              [  applicantId: Id
              ,  name: Nm
              ,  address: Addr
              ,  reviewDate: RvDt
              , instance: App
              ],
           forall (Skills has member: SkillPair):
              (  SkillPair has left: SA,
                 SkillPair has right: SLvl,
                 #applicantSkill has registerApplicantSkill:
                    [ App: applicant, SA: skillArea, SLvl: integer ]
              ).
```

Figure C.9—Realization of `registerApplicant` operation

```
  applicantSkill: Self has registerApplicantSkill:
              [ App: applicant
              , SA: skillArea
              , SLvl: integer
              ]                    if_def
           Self has create:
              [  applicant: App
              ,  skillArea:SA
              ,  skillLevel: SLvl
              ],
           ....          -- anything else needed for registration
```

Figure C.10—Realization of `registerApplicantSkill` operation

C.5.3 Specify a value class as an enumerated value list

This behavior pattern is used to specify the valid list of values for a value class. In the case of HiHo, when an applicant is made a job offer and the offer is refused, the applicant must provide one of HiHo's prescribed refusal reasons.

HiHo's requirement statement: "The only reasons we record for an applicant's refusal of a job offer are: salary, job duties, hours, benefits package, commute, other."

When an interview has been completed, it has one of three outcomes: the applicant is rejected by the employer, the employer makes an offer and the applicant accepts, or the employer makes an offer and the applicant refuses the offer. In the last case, one of the valid reasons must be recorded. The realization of the value class ensures that only one of the reasons that HiHo has specified may be recorded as a refusal reason. Any request that attempts to provide another value will fail. The value class graphic for `refusalReason` and its use in a property of the `interview` state class is shown in Figure C.11, and the value class property realizations are given in Figure C.12.

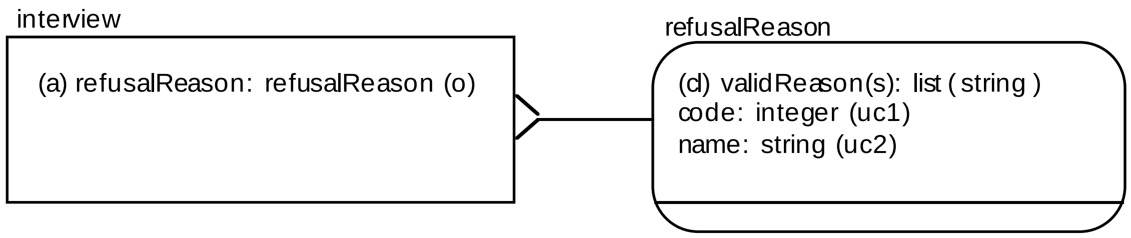

interview
(a) refusalReason: refusalReason (o)
refusalReason
(d) validReason(s): list (string)
code: integer (uc1)
name: string (uc2)

Figure C.11—`refusalReason` value class

```
refusalReason: Self has uc1: Code if_def
         not Code < 1,
         not Code > 6,
         Self has validReason(s): VRs,
         VRs has at: [Code, Name]
         Self has code: Code,
         Self has name: Name.

refusalReason: Self has uc2: Name if_def
         Self has validReason(s): VRs,
         VRs has at: [Code, Name],
         Self has code: Code,
         Self has name: Name.

refusalReason: Self has validReason(s): Vs if_def
         Vs is
               [  "salary"
               ,  "job duties"
               ,  "hours"
               ,  "benefits package"
               ,  "commute"
               ,  "other"
               ].
```

Figure C.12—Realization of valid "refusal reasons"

C.5.4 Pendent deletion

This behavior pattern is used to specify a deletion constraint that permits the deletion of an instance that has no dependents while placing a special constraint on the deletion of an instance that does have dependents. In this case, when the instance to be deleted has dependents, that instance is not deleted immediately but rather placed in a state that prohibits the establishment of any further dependent relationships; when the last dependent is gone, the instance is deleted.

HiHo's requirement statement: "We may decide to discontinue (delete) one of our skill areas, but only if it is not actively being used, i.e., associated with any applicant skills. If a skill area that we wish to discontinue does have active applicant skills registered, we will continue to support that skill area until it is no longer used, but we do want to discontinue allowing new applicant skills to be accepted against the skill area."

Figure C.13 shows a graphic supporting this HiHo requirement. Figure C.14 provides the realizations. The built-in operation establishing new relationship instances between skill area and applicant skill must be overridden to check the value of a `skillArea`'s `myState` property and fail if its value indicates a value that prohibits new relationships. Finally, to make this solution bullet-proof, the deletion of `skillArea` is overridden to prohibit the direct deletion of a skill area while it still has related applicant skills.

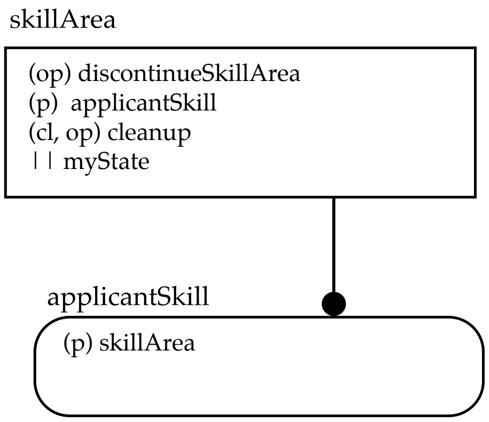


Figure C.13—Skill area and its dependent

```
skillArea: Self has discontinueSkillArea ifdef
     if
          not Self has applicantSkill: Any    -- if no participants
     then
          Self has delete                     -- delete the Skill Area
     else
          Self has myState:= 'discontinued'   -- 'discontinued' state
     endif.

skillArea: Self has cleanup ifdef
     forall (skillArea has instance: SA):
          (    if
                    not SA has applicantSkill: Any,
                    SA has myState: 'discontinued'
               then
                    SA has delete
               endif
          ).

skillArea: Self has applicantSkill:+= AS ifdef -- override the builtin
     if
          Self has myState: 'discontinued'
     then
          false
     else
          Self super has applicantSkill:= AS
     endif.

skillArea: Self has delete ifdef                -- override the builtin
     not Self has applicantSkill: Any,
     Self super has delete.
```

Figure C.14—Realization of pendent deletion operations for `skillArea`

C.5.5 Generalized change request

This behavior pattern is used to specify an operation that can respond to a request that may supply (optionally) one or more attribute values, i.e., it illustrates how to handle an input argument list in which one or more arguments may have no value.

HiHo's requirement statement: "Applicants may contact us and ask us to update one or more of the details we have recorded in their files, e.g., their names and/or their addresses and/or their phone numbers."

An applicant's `correctApplicantDetails` operation may find a new name and/or a new address and/or a new phone number when a request is made, i.e., any or all of the input argument values may be supplied with a request. Figure C.15 shows the graphic for `applicant`, and Figure C.16 provides the realization of the `correctApplicantDetails` operation.

applicant

```
name: name
address: address
phone: integer (o)

correctApplicantDetails: [ Nm: name, Adr: address, Phn: integer ]
```

Figure C.15—Applicant's generalized change operation

```
applicant: Self has correctApplicantDetails:
  [ Nm: name, Adr: address, Phn: integer ] if_def
         if Self has name: Nm
            then
               true
            else
               Self has name:= Nm
            endif,
         if not Self has address: Adr
            then Self has address:= Adr
            endif,
         if Self has phone: Phn
            then
               true
            else
               Self has phone:= Phn
            endif.
```

Figure C.16—Realization of applicant's `correctApplicantDetails` operation

This handles all variations of request argument values and current instance values. For example, if `Nm` has a value when the request is made and that value is not the value currently assigned to `name,` then `Self has name: Nm` is false; and the new value is assigned to `name` (following the `else` leg). If, on the other hand, `Nm` has no value when the request is made, `Nm` is treated as an output argument, its value is set to the current value of `name`, and `Self has name: Nm` is true (having no effect). Finally, if `phone` (an optional attribute) has no value and there is no value of `Phn` supplied at the time of the request, the comparison is false and `phone` is (re)set to "no value." The test and change to `address` illustrates an alternative form of the `if/then/else` syntax, using the conjunction.

C.6 Value class examples

C.6.1 The sunset query

IDEF1X model constructs and query language provide a clean mapping to various "extended relational" implementation languages. This material presents the solution to Stonebraker's "sunset query" problem [B25] as an IDEF1X model, with the query expressed in RCL. It then illustrates how the model specification could be translated into the syntax of one object-relational DBMS.

The query problem, as described by Stonebraker,[99] comes from the State of California Department of Water Resources (DWR). As a part of its documentation library, DWR maintains a sizable collection of 35-mm slides. Each slide has an identifier, the date it was taken, a caption, and the digitized image in Kodak Photo-CD format. The collection is accessed daily by DWR employees and clients, often on a "query by content" basis, for example, "I am looking for a sunset picture taken within 20 miles of Sacramento." This kind of query cannot be readily satisfied by a search of the file of textual captions that have been associated with the slides. DWR is developing a system that will allow their images to be classified and searched electronically as complex data.

C.6.1.1 The sunset query in IDEF1X

Figure C.17 shows an IDEF1X representation of their design. The realization of the two operations used in the query problem is given in Figure C.18.

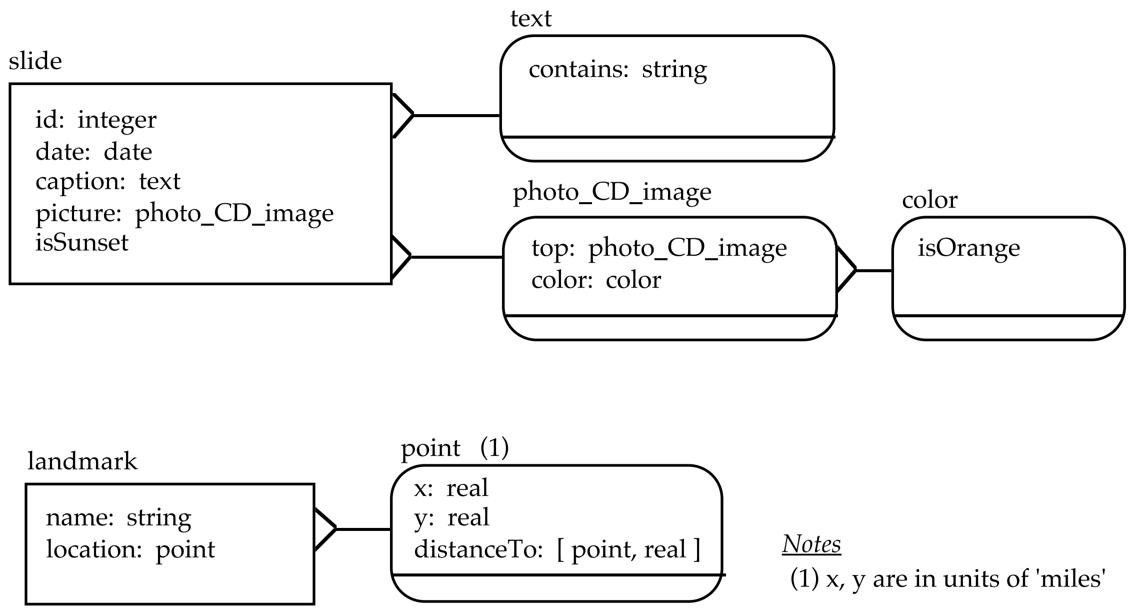


Figure C.17—State and value classes for the "sunset problem"

The query to "find a sunset within 20 miles of Sacramento" is shown in Figure C.19. It involves the following components:

a) Find the geographic `location` of "Sacramento" in the `landmark` table.
b) Find other landmarks that have a location within 20 miles of the Sacramento location.

[99]A discussion of this problem is presented in *Object-Relational DBMSs* [B25] on pp. 13-17.

```
slide: Self has isSunset if_def
  Self has picture..top..color..isOrange.

point: Self has distanceTo: [ Point, D ] if_def
  D is (( Self..x - Point..x )^2 + ( Self..y - Point..y )^2)^0.5.
```

Figure C.18—Realizations used in `isSunset` operation

c) `contains` determines whether a word (passed as an argument) appears in the `caption` text.

d) `isSunset` examines the bits in an image to determine if the image has the color orange at its top. Figure C.19 presents the RCL in a form that parallels the structure of an SQL query. Figure C.20 provides a variation on the same query.

```
Slides is [ P where (                                -- SQL "Select"

  slides has instance: P,                            -- SQL "from"
  landmark has instance: S,
  landmark has instance: L,

  S..name == 'Sacramento',                           -- SQL "where"
  L..location has distanceTo: [ S..location, D ],
  D =< 20,
  Lname is L..name,
  P has caption..contains: Lname,
  P has picture..isSunset

  ].
```

Figure C.19—RCL query to "find a sunset picture within 20 miles of Sacramento"

```
Slides is [ P where (
  S is landmark with name: 'Sacramento',
  L is #landmark..instance,
  L..location has distanceTo: [ S..location, D ],
  D =< 20,
  Lname is L..name,
  P is slide with (caption..contains: Lname, picture..isSunset)
  ].
```

Figure C.20—Alternative RCL for sunset query

C.7 The Table and Chair Company (TcCo)

The Table and Chair Company, TcCo, provides an example of an identity-style model in the form of a view diagram and example instance tables and definitions. TcCo buys parts from vendors and assembles them into tables and chairs. Concepts such as part, vendor, quantity-on-hand, and so on are of concern to TcCo. This UOD has an existence and reality in TcCo independent of any model of it. That UOD is described in Figure C.21 using a view diagram, a depiction of the value classes, a set of sample instance tables, and glossary entries for some of the defined concepts, concept pairs, and relationship descriptions. The example is not intended to be complete; it is simply illustrative.

C.7.1 TcCo production view diagram

Figure C.21 is a view named `production` that reflects the state classes, attributes, and relationships of concern to TcCo. Other views of TcCo could be drawn, sharing some of the same concepts but emphasizing other concerns. For example, a `marketing` view might include `part` with a different set of attributes, omit the `vendor` state class, and add a state class for `customer`.

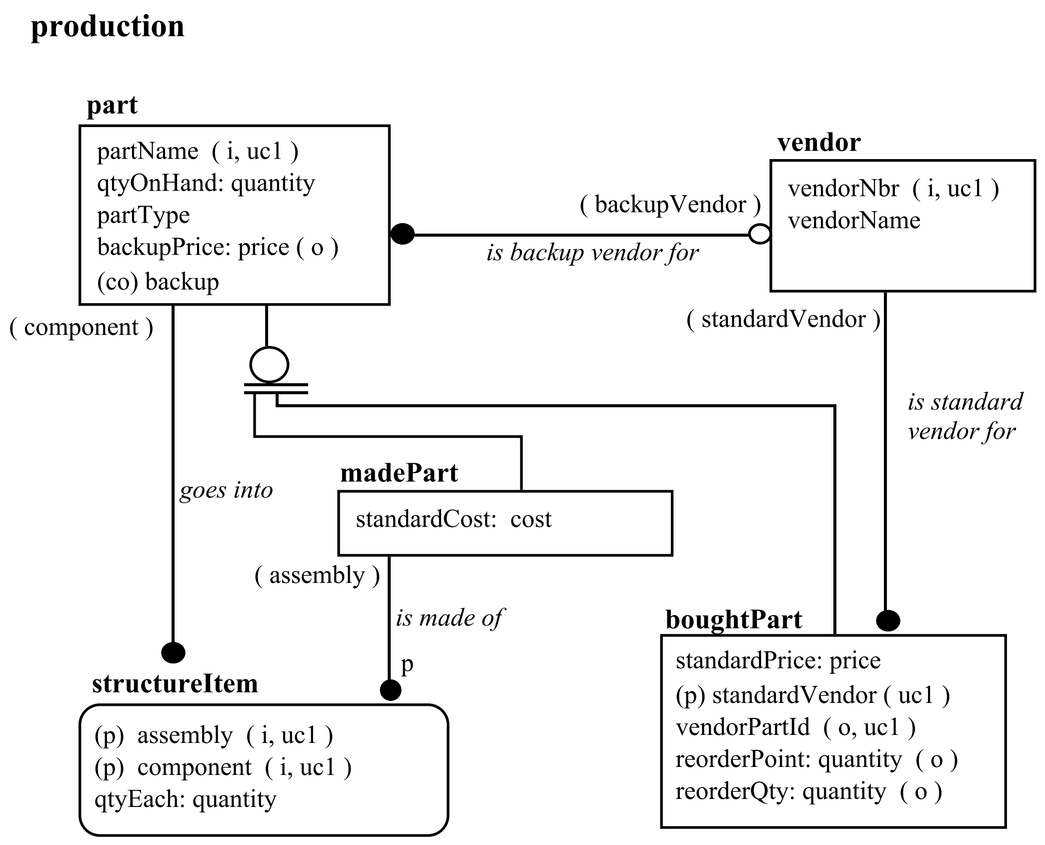


Figure C.21—The `Production` view of The Table and Chair Company

The state classes labeled `part` in the `production` and `marketing` views are distinct state classes. It is the state class in a view (*view state class*) that specifies the state class. For that reason, the terms *view state class* and *state class* will be used interchangeably. Relationships exist between state classes in the context of a view. In Figure C.21, `production part` is related to `production vendor`.[100]

It is a view state class that has properties. In Figure C.21, the view state class `production part` has six properties—four of these are attributes (`partName`, `qtyOnHand`, `partType`, and `backupPrice`). This means that each instance of `production part` potentially has a `partName` value, a `qtyOnHand` value, and so on for each of its attributes. Because `partName` is declared intrinsic, it is constrained to have a single, unchanging value for every instance of `part`; because it is has a uniqueness constraint, these values are required to be distinct. The value of the `partName` attribute of `production part` is an instance

[100]The state class `production part` is denoted formally by `production: part`. That is, the function application `production: part` denotes (evaluates to) the state class.

of the `partName` value class that is a specialization (subclass) of the `name` value class. The value of the `qtyOnHand` attribute of `production part` is an instance of the `quantity` value class.

An attribute may be named by the value class (as in `partName`) or may have its own role name (as in `qty-OnHand`). Saying that a view state class has an attribute `X` (or has a role named attribute `Y` with a type of `X`) is just a way of saying that the view state class is related to (maps to) the value class `X`.

The participant properties reflect the relationships between the view state classes. Each relationship results in two participant properties, one in each of the state classes participating in the relationship. In Figure C.21, the state class `production part` has two participant properties (`backupVendor` and `structure-Item`), although they are not explicitly shown on the diagram. The participant property `backupVendor` reflects the relationship between `production part` and `production vendor`. The property carries the role name, `backupVendor`, which has been given to `vendor`'s participation in the relationship. If displayed, it would be marked `optional` corresponding to the specification stated by the relationship. The participant property `structureItem` corresponds to the relationship between `production part` and `production structureItem`. This property carries the name of the related state class, `structure-Item`, since no role name has been given to `structureItem`'s participation in the relationship. If displayed, it would be marked `optional` and `multiple` (multi-valued), corresponding to the specification stated by the relationship.

Participant properties are normally displayed inside the class rectangle only if necessary to state a constraint, like uniqueness, that could not otherwise be stated. For this reason, in `structureItem` the two participant properties are shown.

C.7.2 TcCo value classes

The value classes for TcCo are illustrated in Figure C.22. Note that in a completed model the inclusion of each value class would need to be justified on the basis of having useful responsibilities. Further analysis would screen out those with no useful responsibilities and identify the useful responsibilities of surviving value classes.

Figure C.22 shows value classes, such as `vendorPartId` and `quantity`, shows representation datatypes such as `character` and `number` (where specified), and shows generalization hierarchies, such as `price` and `cost,` which are each specializations of `money`.

C.7.3 TcCo sample instance tables

The UOD for TcCo includes instances of parts and vendors, attribute values for the instances, and relationships between the instances. These are shown in Figure C.23 as sample instance tables.

The `#K` at the left of the tables in Figure C.23 represent the oid of a "thing" that, in the TcCo UOD, is classified as being a member of the state class corresponding to the table. In this case, the individual identified as `#1` is classified as both a `production part` and a `production madePart`.

An individual considered as an instance of a view state class is called a *view state class instance* (simply, *instance*). The instance tables in Figure C.23 show view state class instances, such as `production part #1` and `production madePart #1`.

The values of the attributes of a view state class instance are aligned in rows and columns, one row for each view state class instance and one column for each attribute. The rows in these examples are shown in the sequence of the objects' oids. However, there is no requirement to sequence the rows or any implied meaning in the sequence.

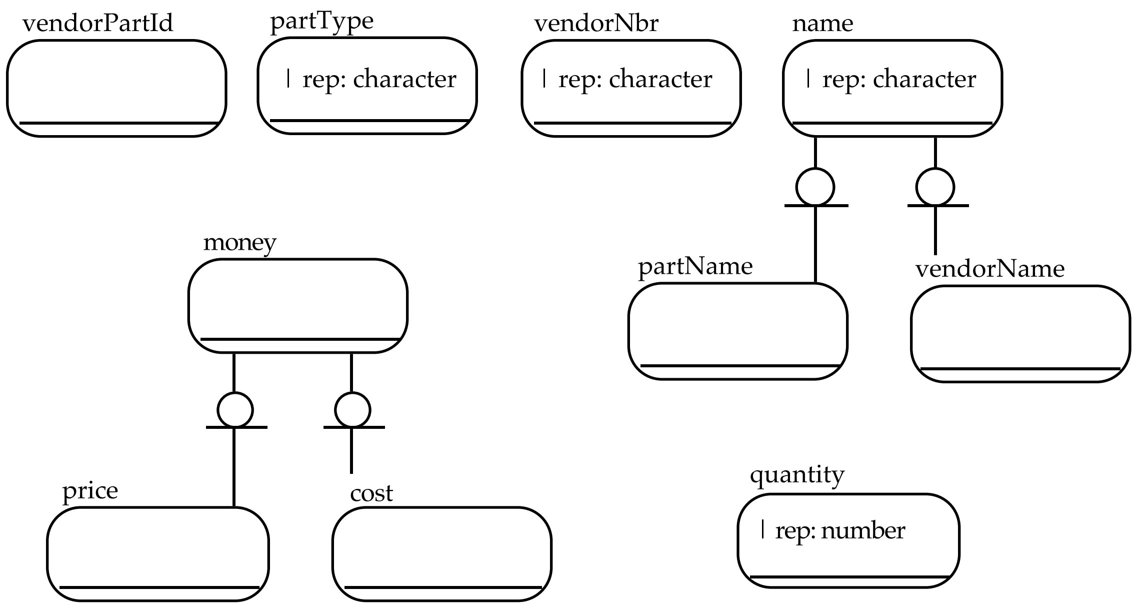


Figure C.22—TcCo value classes

The value of an attribute is an instance of a value class, so each cell in fact specifies a value class instance, such as `vendorPartId 57`, denoted formally as `vendorPartId: 57`. The value shown, such as `57`, is the representation portion; the value class name is given by the column heading. In this case the column heading is both the name of the attribute and the name of the value class.

The special symbol "−−" indicates that a view state class instance has no value for a property. Note that "−−" is not a value; "−−" is not an instance of any value class.

C.7.4 TcCo sample relationship instances

One form of presenting the sample instances of relationships for the TcCo `production` view is shown in Figure C.24. This style of displaying relationship instances uses one table for each relationship, with each table named for the participating state classes and using the role names (if any). Each instance table has two columns, each named for the participating state classes. Each row of a table is an ordered pair consisting of the identities of the participating instances. For example, the "standardVendor boughtPart" table indicates that `production boughtPart #4` has `standardVendor #201`.

An alternative method of presenting relationship instances is as a sample instance diagram of state classes. The same relationships shown in Figure C.24 for `standardVendor/boughtPart` are depicted in Figure C.25.[101]

Both forms of presentation demonstrate that *"one vendor may be the standard vendor for many bought parts"* by showing `vendor #301` as the standard vendor for `boughtPart #6` and `boughtPart #7`.

C.7.5 TcCo sample definitions

Table C.1 provides examples of definitions for TcCo. Both "Single Concepts" and "Concept Pairs" are provided (see the formalization metamodel for the structure of this section of the glossary).

[101]This illustration omits the display of the other properties of TcCo's `vendor` and `boughtPart` state classes.

`part`

	partName	qtyOnHand	partType	backupPrice
#1	table	2000	m	--
#2	top	17	m	800
#3	leg	3	m	10
#4	shaft	7	b	8
#5	foot	3	b	6
#6	screw	100	b	2
#7	plank	0	b	--
#8	seat	2	m	--

`madePart`

	standardCost
#1	300
#3	5
#2	100
#8	50

`vendor`

	vendorNbr	vendorName
#101	10	Acme
#201	17	Babcock
#301	30	Cockburn
#401	40	Dow
#501	50	Eisenhower
#601	60	Fum

`boughtPart`

	standardPrice	vendorPartId	reorderPoint	reorderQty
#4	6	57	10	15
#5	3	--	--	7
#6	1	10ab-33	30	40
#7	4	--	--	--

`structureItem`

	qtyEach
#17	4
#27	1
#37	4
#47	4
#57	1
#67	1
#77	1
#97	1

Figure C.23—Sample instances of TcCo

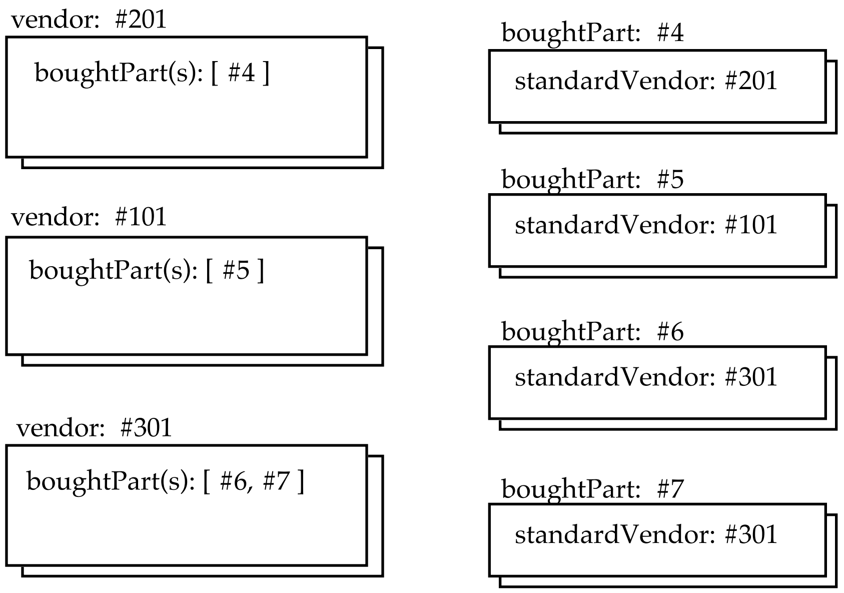


Figure C.24—Alternative presentation of TcCo relationship instances

`standardVendor` `boughtPart`

vendor	boughtPart
#201	#4
#101	#5
#301	#6
#301	#7

`backupVendor` `part`

vendor	part
#201	#2
#101	#3
#101	#4
#401	#5
#401	#6

`component` `structureItem`

part	structureItem
#2	#27
#3	#37
#4	#57
#5	#67
#6	#47
#6	#17
#7	#77
#7	#97

`assembly` `structureItem`

madePart	structureItem
#1	#17
#1	#27
#1	#37
#3	#47
#3	#57
#3	#67
#2	#77
#8	#97

Figure C.25—Sample relationship instances of TcCo

Table C.1—Sample glossary entries for TcCo

Single concepts

oid	Name	Description
#sC1	part	What TcCo makes or uses to make what it makes.
#sC2	vendor	A party who sells parts to TcCo.
#sC3	quantity	A count of the number of something.
#sC4	name	A distinctive designation by which some thing is known.
#sC5	partName	A name that may be given to a part.
#sC6	vendorName	A name that may be given to a vendor.
#sC7	backup	TcCo policy states that there must be a backup vendor for any part that TcCo does not make.

Concept pairs

oid	p-props	Name	Description
#cP1	#sC1, #sC5	part + partName	A name that may be given to a part. In this case, the name given to this part.
#cP2	#sC1, #sC3	part + qtyOnHand	A count of the number of something. In this case, the quantity of this part that TcCo has in stock.
#cP3	#sC1, #sC2	part + backupVendor	A party who sells parts to TcCo. In this case, the vendor who is “backup” for the “primary” vendor of this part. Not all parts have such a backup vendor.
#cP4	#sC1, #sC7	part + backup	TcCo policy states that there must be a backup vendor for any part that TcCo does not make.
#cP5	#sC2, #sC6	vendor + vendorName	A name that may be given to a vendor. In this case, the name given to this vendor.
#cP6	#sC2, #sC1	vendor + part	What TcCo makes or uses to make what it makes. In this case, a part that may be provided to TcCo by this vendor in emergency (“backup”) situations.

Relationship descriptions

oid	p-props	Name	Description
#rD1	#cP3, #cP6	part + backupVendor + vendor + part	TcCo policy states that there must be a backup vendor for any part that TcCo does not make. The *backup* relationship is used whenever the primary vendor of the part is unable to supply sufficient quantities.

Annex D

(informative)

Built-in classes

The built-in state and value classes are extensible. Properties can be added, subclasses can be added, and properties can be overridden. Parametric value classes can be added. Subclasses can be added to the anonymous top view shown in Figure D.1, or in a named view.

The built-in classes shall include at least those shown here, with the properties shown. The realization RCL determines the semantics of the properties. Any equivalent RCL that satisfies the semantics is acceptable.

D.1 Built-in state classes

The built-in state classes are metamodel classes. They provide properties such as instance creation, instance deletion, and access to the instances of a class. See 10.2 for the specification of the built-in state classes.

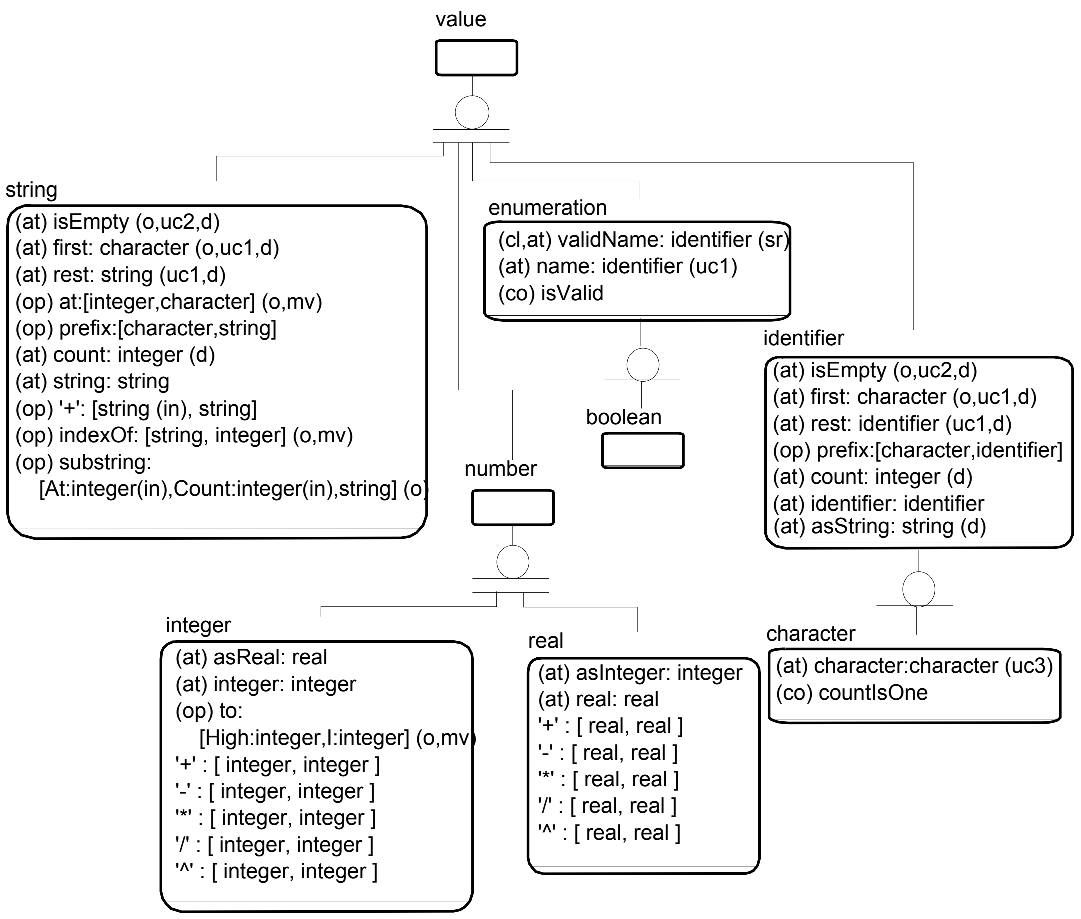

value
string
(at) isEmpty (o,uc2,d)
(at) first: character (o,uc1,d)
(at) rest: string (uc1,d)
(op) at:[integer,character] (o,mv)
(op) prefix:[character,string]
(at) count: integer (d)
(at) string: string
(op) '+': [string (in), string]
(op) indexOf: [string, integer] (o,mv)
(op) substring:
[At:integer(in),Count:integer(in),string] (o)
enumeration
(cl,at) validName: identifier (sr)
(at) name: identifier (uc1)
(co) isValid
boolean
number
identifier
(at) isEmpty (o,uc2,d)
(at) first: character (o,uc1,d)
(at) rest: identifier (uc1,d)
(op) prefix:[character,identifier]
(at) count: integer (d)
(at) identifier: identifier
(at) asString: string (d)
integer
(at) asReal: real
(at) integer: integer
(op) to:
[High:integer,l:integer] (o,mv)
'+' : [integer, integer]
'-' : [integer, integer]
'*' : [integer, integer]
'/' : [integer, integer]
'^' : [integer, integer]
real
(at) asInteger: integer
(at) real: real
'+' : [real, real]
'-' : [real, real]
'*' : [real, real]
'/' : [real, real]
'^' : [real, real]
character
(at) character:character (uc3)
(co) countIsOne

Figure D.1—Built-in value classes

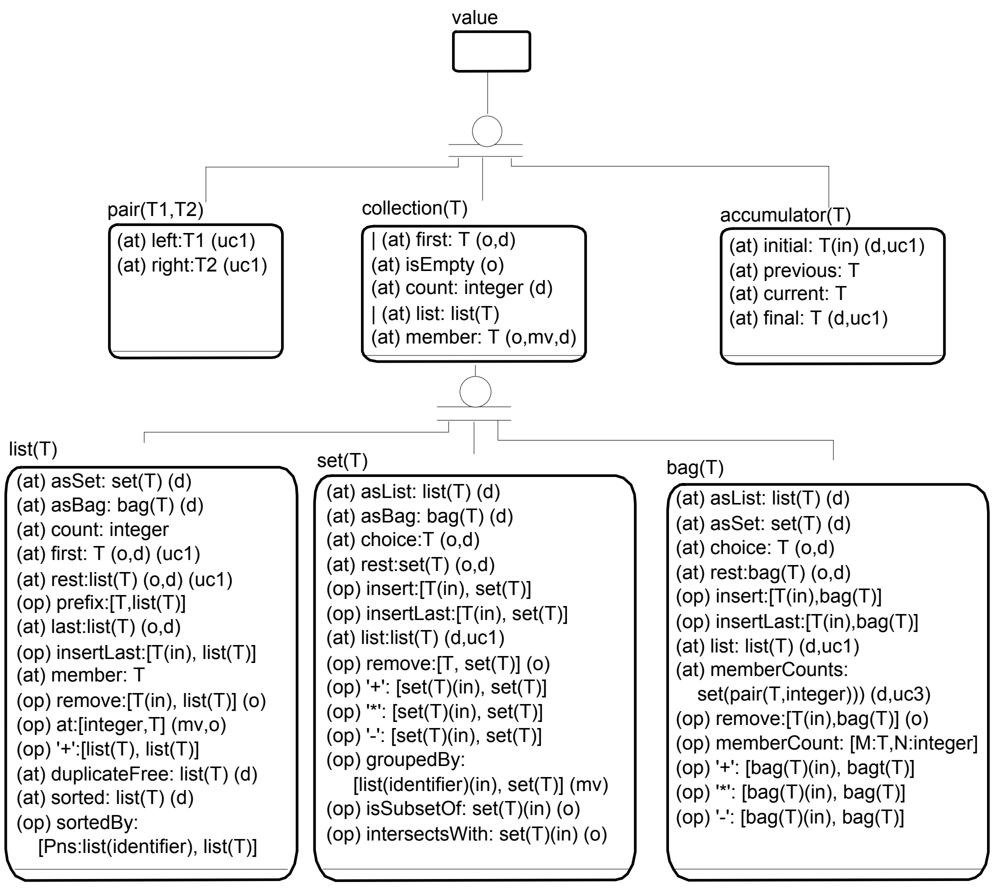

value
pair(T1,T2)
(at) left:T1 (uc1)
(at) right:T2 (uc1)
collection(T)
| (at) first: T (o,d)
(at) isEmpty (o)
(at) count: integer (d)
| (at) list: list(T)
(at) member: T (o,mv,d)
accumulator(T)
(at) initial: T(in) (d,uc1)
(at) previous: T
(at) current: T
(at) final: T (d,uc1)
list(T)
(at) asSet: set(T) (d)
(at) asBag: bag(T) (d)
(at) count: integer
(at) first: T (o,d) (uc1)
(at) rest:list(T) (o,d) (uc1)
(op) prefix:[T,list(T)]
(at) last:list(T) (o,d)
(op) insertLast:[T(in), list(T)]
(at) member: T
(op) remove:[T(in), list(T)] (o)
(op) at:[integer,T] (mv,o)
(op) '+':[list(T), list(T)]
(at) duplicateFree: list(T) (d)
(at) sorted: list(T) (d)
(op) sortedBy:
[Pns:list(identifier), list(T)]
set(T)
(at) asList: list(T) (d)
(at) asBag: bag(T) (d)
(at) choice:T (o,d)
(at) rest:set(T) (o,d)
(op) insert:[T(in), set(T)]
(op) insertLast:[T(in), set(T)]
(at) list:list(T) (d,uc1)
(op) remove:[T, set(T)] (o)
(op) '+': [set(T)(in), set(T)]
(op) '*': [set(T)(in), set(T)]
(op) '-': [set(T)(in), set(T)]
(op) groupedBy:
[list(identifier)(in), set(T)] (mv)
(op) isSubsetOf: set(T)(in) (o)
(op) intersectsWith: set(T)(in) (o)
bag(T)
(at) asList: list(T) (d)
(at) asSet: set(T) (d)
(at) choice: T (o,d)
(at) rest:bag(T) (o,d)
(op) insert:[T(in),bag(T)]
(op) insertLast:[T(in),bag(T)]
(at) list: list(T) (d,uc1)
(at) memberCounts:
set(pair(T,integer))) (d,uc3)
(op) remove:[T(in),bag(T)] (o)
(op) memberCount: [M:T,N:integer]
(op) '+': [bag(T)(in), bagt(T)]
(op) '*': [bag(T)(in), bag(T)]
(op) '-': [bag(T)(in), bag(T)]

Figure D.1—Built-in value classes *(continued)*

D.2 Built-in value classes

D.2.1 Interfaces

D.2.2 Realizations

(The properties shown in the graphics for which no realization is given are directly realized by axioms, as described in Clause 10.)

D.2.2.1 Accumulator(T)

D.2.2.1.1 `(at) accumulator(T): current`

The current value of the accumulator, or if no value yet exists, the previous value.

```
accumulator(T): Self has current:(V:T) if_def
     V is (Self super)..current,
     if V is Self..previous then true endif
```

D.2.2.1.2 (at) accumulator(T): final

The current value of the accumulator.

```
accumulator(T): Self has final:(V:T) if_def
     V is Self..current
```

D.2.2.1.3 (at) accumulator(T): initial

The initial value of the accumulator.

```
accumulator(T): Self has initial:(V:T)(in) if_def
     V is Self..previous.
```

D.2.2.1.4 (co) accumulator(T): uc1

Uniqueness constraint 1 for the accumulator. The initial and final values determine the values of the representation properties, previous and current. The uniqueness constraint is responsible for determining T.

```
accumulator(T): Self has uc1:[Initial:T,Final:T] if_def
     Self has previous:Initial,
     Self has current: Final,
     Initial has lub:[Final,T].
```

D.2.2.1.5 Example

Using the model in Figure C.17, an accumulator can be used in a query to find the `Landmark` closest to a given `Point` and its `Distance` to that point.

```
Variable Acc: accumulator(pair(landmark,real)),
L is landmark..instance,              -- an arbitrary instance to
     use as
D is L..location..distanceTo(Point),  -- the initial closest
Acc is accumulator(initial(L:D), final(Landmark:Distance)),
for Acc all (Landmark is landmark..instance): (
     Distance is Landmark..location..distanceTo(Point),
     if Distance < Acc..previous..right
     then
          Acc..current is Landmark:Distance
     endif)
```

D.2.2.1 Bag

D.2.2.2.1 (at) bag(T): asList

`List` is the receiver bag as a list.

```
bag(T): Self has asList:(List:list(T)) if_def
     List is Self..list
```

D.2.2.2.2 (at) bag(T): asSet

`Set` is the receiver bag as a set.

```
bag(T): Self has asSet:(Set:set(T)) if_def
     Set is set with list(Self..list)
```

D.2.2.2.3 (at) bag(T): choice

One of the members.

```
bag(T): Self has choice: (X:T) if_def
     X is Self..list..first
```

D.2.2.2.4 (op) bag(T): insert

The bag with X added.

```
bag(T): Self has insert:[X:T,Result:bag(T)] if_def
     Result is bag with list(Self..list..prefix(X))
```

D.2.2.2.5 (op) bag(T): insertLast

The bag with X added.

```
bag(T): Self has insertLast:[X:T,Result:bag(T)] if_def
     Result is Self..insert(X)
```

D.2.2.2.6 (op) bag(T): intersectsWith

True if the receiver has at least one element in common with the argument.

```
bag(T): Self has intersectsWith:[Bag:bag(T)] if_def
     Self has member:X,
     Bag has member:X
```

D.2.2.2.7 (op) bag(T): memberCount

A count N for a member M.

```
bag(T): Self has memberCount: [M:T,N:integer] if_def
     Self..asSet has member:M,
     Acc is accumulator(initial:0,final:N),
     for Acc all (Self has member:M):
     (acc..current is acc..previous + 1)
```

D.2.2.2.8 (at) bag(T): memberCounts

A set of pairs M:N giving the count N for a member M.

```
bag(T): Self has memberCounts: (MCs:set(pair(T,integer))) if_def
     MCs is { M:N where Self has memberCount:[M,N] }
```

D.2.2.2.9 (op) bag(T): remove

The bag with X removed.

```
bag(T): Self has remove:[X:T(in),Result:bag(T)] if_def
     Result is bag with list(Self..list..remove(X))
```

D.2.2.2.10 **(at) bag(T): rest**

The rest of the members, all but the choice.

```
bag(T): Self has rest: (Xs:bag(T)) ifdef
     Xs is bag with list(Self..list..rest)
```

D.2.2.2.11 **(at) bag(T): uc1**

The uniqueness constraint is on `list`.

```
bag(T): Self has uc1: (List:list(T)) ifdef
     Self..list is List..sorted
```

D.2.2.2.12 **(at) bag(T): uc3**

The uniqueness constraint is on `memberCounts`.

```
bag(T): Self has uc3: (MCs:set(pair(T:integer))) ifdef
     List is [M where MCs has member:(M:N),I is 0..to(N-1)],
     Self has uc1: List
```

D.2.2.2.13 **(op) bag(T): '+'**

`Result` is the union of `Self` and `Bag`.

```
bag(T): Self has '+':[Bag:bag(T),Result:bag(T)] ifdef
     Result is bag(X where Self has member:X or Bag has member:X)
```

D.2.2.2.14 **(op) bag(T): '*'**

`Result` is the intersection of `Self` and `Bag`.

```
bag(T): Self has '*':[Bag:bag(T),Result:bag(T)] ifdef
     Set is (Self + Bag)..asSet,
     MCs is {M:N where
          M is Set..member,
          if Self..memberCount(M)< Bag..memberCount(M)
          then
               N is Self..memberCount(M)
          else
               N is Bag..memberCount(M)
          endif},
Result is bag with memberCounts: MCs
```

D.2.2.2.15 **(op) bag(T): '-'**

`Result` is the bag difference, `Self` minus `Bag`.

```
bag(T): Self has '-':[Bag:bag(T),Result:bag(T)] ifdef
     Set is (Self + Bag)..asSet,
     MCs is {M:N where
          M is Set..member,
          N is Self..memberCount(M) - Bag..memberCount(M),
          N > 0},
     Result is bag with memberCounts: MCs
```

D.2.2.3 Character

`Character` is a subclass of `identifier`, but does not have the same representation. `Identifier` is represented by a composition of `prefixIdentifier` function applications. `Character` is represented in any way so that `character`'s value satisfies the `isCharacter` predicate. The uniqueness constraints on identifier is overridden to establish both representation values.

D.2.2.3.1 (co) character: countIsOne

The count must be one. This constraint is checked at the conclusion of any literal for a character. Use of the inherited uniqueness constraint on `isEmpty` will cause the constraint to fail.

```
character: Self has countIsOne if_def
     Self super has count: 1
```

D.2.2.3.2 (op) character: uc1

The uniqueness constraint is on `first` and `rest` overrides `uc1` in `identifier`. If `Rest` is not `''`, then `uc3` will be false. Therefore, `uc1` will be false, and the `isTotal` constraint will be violated. Such a violation will cause an exception to be raised.

```
character: Self has uc1: [First:character,Rest:identifier] if_def
     Self super has uc1:[First,Rest],
     Self has uc3: First
```

D.2.2.3.3 (op) character: uc3

The uniqueness constraint is on `character`.

```
character: Self has uc3: (Char:character) if_def
     Self super has uc1:[Char,''],
     Self..character is Char
```

D.2.2.4 Collection

D.2.2.4.1 (at) collection(T): count

The number of members of the collection.

```
collection(T): Self has count: (N:integer) if_def
     N is Self..list..count
```

D.2.2.4.2 (at) collection(T): first

The first member, if any.

```
collection(T): Self has first (X:T) if_def
     X is Self..list..first
```

D.2.2.4.3 (at) collection(T): isEmpty

True if the collection is empty.

```
collection(T): Self has isEmpty if_def
     Self..list..isEmpty
```

D.2.2.4.4 **(at) collection(T): member**

`Result` is a member of the collection.

```
collection(T): Self has member: (X:T) ifdef
     X is Self..list..member
```

D.2.2.5 Enumeration

The `validName` class-level property is a multi-valued property for each of the valid names for the enumeration. Every instance has one of the names in the list. This property is intended to be overriden by every subclass. It needs no explicit realization in the enumeration class. The name property is the name of an instance. It needs no explicit realization. The uniqueness constraint is on `name`. The `isValid` constraint checks that the name of the instance is valid. There is typically no need to override the uniqueness constraint or the `isValid` constraint. The constraint is checked as a part of the specification of an instance by a literal. If the constraint fails, the literal fails.

D.2.2.5.1 **(co) enumeration: isValid**

```
enumeration: Self has isValid ifdef
     Self..name is Self..validName.
```

D.2.2.5.2 **(at) enumeration: uc1**

```
enumeration: Self has uc1: Name ifdef
     Self..name is Name.
```

D.2.2.5.3 Example

The enumeration color contains three valid names: red, green, and blue. The only realization that is needed is for the `validNames`.

```
color: Self has validName:Name ifdef
     Name is [red,green,blue]..member.
```

An instance of the enumeration `color` is specified by, for example,

```
Red is color(name:red).
```

A literal such as

```
Huh is color(name:who)
```

will fail; in other words, the proposition is false. An exception is raised because the `isValid` constraint is not met.

A less than operator can be defined for an ordered enumeration. For example,

```
color: Self has '<': (Color:color) ifdef
     Self..name is Self..validName(s)..at I,
     Color..name is Self..validName(s)..at J,
     I < J.
```

D.2.2.6 Identifier

`Identifier` is a primitive class with its `isEmpty`, `first`, `rest`, and `prefix` properties supplied by axioms.

D.2.2.6.1 (at) identifier: count

`Count` is the number of characters in the `identifier`.

```
identifier: Self has count: N if_def
     if Self..isEmpty
     then
          N is 0
     else
          N is 1 + Self..rest..count
     endif
```

D.2.2.6.2 (op) identifier: uc1

The uniqueness constraint is on `first` and `rest`.

```
identifier: Self has uc1: [First:character,Rest:identifier] if_def
     Self..identifier is Rest..prefix(First)
```

D.2.2.6.3 (op) identifier: uc2

The uniqueness constraint is on is`Empty`.

```
identifier: Self has uc2 if_def
     Self..identifier is ‘’
```

D.2.2.6.4 (at) identifier: asString

The identifier as a string.

```
identifier: Self has asString:String if_def
     if Self..isEmpty
     then
          String is String(isEmpty)
     else
          String is string(first(Self..first),rest(Self..rest..
          asString))
     endif
```

D.2.2.7 integer

D.2.2.7.1 (op) integer: to

The integers `Self` to `N`, `N >= I`, in order.

```
integer: Self has to:[N,I] if_def
     N >= Self,
     I is Self or I is (Self+1)..to(N)
```

D.2.2.7.2 (op) integer: ‘/’

Integer division, B is Self/A.

```
integer: Self has ‘/’:[A,B] if_def
     pre A != 0,
     Self super has ‘/’:[A,B]
```

D.2.2.8 List

D.2.2.8.1 (at) list(T): asBag

Bag is the receiver list as a bag.

```
list(T): Self has asBag:(Bag:bag(T)) if_def
     Bag is bag with list: Self
```

D.2.2.8.2 (at) list(T): asSet

Set is the receiver list as a set.

```
list(T): Self has asSet:(Set:bag(T)) if_def
     Set is set with list: Self
```

D.2.2.8.3 (op) list(T): at

The list has X at location N, where the first location is 0.

```
list(T): Self has at:[N:integer,X:T] if_def
     (Self has first:X, N is 0)
or
     (Self has rest..at:[M,X], N is M+1)
```

D.2.2.8.4 (at) list(T): count

The number of members of the list.

```
list(T): Self has count: (N:integer) if_def
     if Self..isEmpty
     then
          N is 0
     else
          N is 1 + Self..rest..count
     endif
```

D.2.2.8.5 (op) list(T): duplicateFree

The Result is Self with all distinct members.

```
list(T): Self has duplicateFree: ( Result:list(T)) if_def
   if Self == []
   then
      Result is Self
   else
      if Self..rest has member(Self..first)
```

```
            then
               Result is Self..rest..duplicateFree
            else
               Result is Self..rest..duplicateFree..prefix(Self..first)
            endif
        endif
```

D.2.2.8.6 (op) `list(T): insertLast`

The `Result is Self` with X added as the last member.

```
    list(T): Self has insertLast: [X:T, Result:list(T)] if_def
         if Self == []
         then
              Result is [X]
         else
              Result is Self..rest..insertLast(X)..prefix(Self..first)
         endif
```

D.2.2.8.7 (op) `list(T): last`

The list has X as the last member.

```
    list(T): Self has last: (X:T) if_def
         (Self == [X])
         or
         (Self != [], Self has rest..last:X)
```

D.2.2.8.8 (at) `list(T): member`

Result is a member of the list.

```
    list(T): Self has member: (X:T) if_def
     X is Self..first
     or
     X is Self..rest..member
```

D.2.2.8.9 (op) `list(T): remove`

The `Result is Self` with the first X, if any, removed.

```
    list(T): Self has remove: [X:T(in), Result:list(T)] if_def
         if Self == []
         then
              Result is Self
         else
              if Self..first == X
              then
                   Result is Self..rest
              else
                   Result is Self..rest..remove(X)..prefix(Self..first)
              endif
         endif
```

D.2.2.8.10 (op) `list(T): sorted`

`Result is Self` sorted.

```
list(T): Self has sorted: (Result:list(T)) if_def
     if Self..isEmpty
     then
          Result is Self
     else
          U is Self..first,
          Acc is accumulator(initial:([]:[]),final:(LE:GT)),
          for Acc all (X is Self..rest..member): (
               if X <= U
               then
                     L is Acc..previous..left..prefix(X)
                     R is Acc..previous..right
               else
                     L is Acc..previous..left
                     R is Acc..previous..right..prefix(X)
               endif,
               Acc..current is L:R
          ),
          Result is LE..sorted + [U] + GT..sorted
endif
```

D.2.2.8.11 (op) `list(T): sortedBy`

`Result is Self` sorted on the property values, `Pns`. Each property must be a single-argument property of the receiver or a superclass of the receiver.

```
list(T): Self has sortedBy: (Pns:list(identifier),Result:list(T)) if_def
     pre (forall (Pn is Pns..member)):
          (Self..lowClass..superStar has responsibility: R,
          R has name: Pn,
          R has type..count:1),
     Vs is [ MVs:M where Self has member:M,
          MVs is [V where Pns has member:Pn, M has Pn:V]]
Result is [M where Vs..sorted has member: (_:M)]
```

D.2.2.8.12 (op) `list(T): uc1`

The uniqueness constraint is on `first` and `rest`.

```
list(T): Self has uc1: [First:T,Rest:list(T)] if_def
     Self..list is Rest..prefix(First)
     Acc is accumulator(initial:bot,final:T),
     for Acc all (Self has member:M):
          (Acc..current is M..lowClass..lub(acc..previous))
```

D.2.2.8.13 (op) `list(T): '+'`

Result is the concatenation of `Self` and `Cat`.

```
list(T): Self has '+': [Cat:list(T),Result:list(T)] if_def
```

```
          if Self..isEmpty
          then
               Result is Cat
          else
               Result is (Self..rest + Cat)..prefix(Self..first)
          endif
```

D.2.2.9 Real

D.2.2.9.1 (op) real: '/'

Real division, `B is Self/A`.

```
     integer: Self has '/':[A,B] if_def
          pre A != 0.0,
          Self super has '/':[A,B]
```

D.2.2.10 Set

D.2.2.10.1 (at) set(T): asBag

`Bag` is the receiver set as a bag.

```
     set(T): Self has asBag:(Bag:bag(T)) if_def
          Bag is bag with list(Self..list)
```

D.2.2.10.2 (at) set(T): asList

`List` is the receiver set as a list.

```
     set(T): Self has asList:(List:list(T)) if_def
          List is Self..list
```

D.2.2.10.3 (at) set(T): choice

One of the members.

```
     set(T): Self has choice: (X:T) if_def
          X is Self..list..first
```

D.2.2.10.4 (op) set(T): groupedBy

`Result` is a subset of the receiver set. Each member of the subset has the same values for the properties, `Pns`. Each property must be a single-argument property of the receiver or a superclass of the receiver.

```
set(T): Self has groupedBy:[Pns:list(identifier)(in),Result:set(T)]
if_def
     pre (forall (Pn is Pns..member)):
          (Self..lowClass..superStar has responsibility: R,
          R has name: Pn,
          R has type..count:1),
     Divisors is { PnVs where
          Self has member: M,
          PnVs is [ Pn:V where Pns has member: Pn, M has Pn:V]},
     Divisors has choice: D,
     Result is { M where
          Self has member: M,
          forall (D has member: (Pn:V)): (M has Pn:V)}
```

D.2.2.10.5 (op) set(T): insert

The set with X added.

```
     set(T): Self has insert:[X:T,Result:set(T)] ifdef
          Result is set with list(Self..list..prefix(X))
```

D.2.2.10.6 (op) set(T): insertLast

The set with X added.

```
     set(T): Self has insertLast:[X:T,Result:set(T)] ifdef
          Result is Self..insert(X)
```

D.2.2.10.7 (op) set(T): intersectsWith

True if the receiver has at least one element in common with the argument.

```
     set(T): Self has intersectsWith:[Set:set(T)] ifdef
          Self has member:X,
          Set has member:X
```

D.2.2.10.8 (op) set(T): isSubsetOf

True if the receiver is a subset of the argument.

```
     set(T): Self has isSubsetOf:[Set:set(T)] ifdef
          forall (Self has member:X, Set has member:X)
```

D.2.2.10.9 (op) set(T): remove

The set with X removed.

```
     set(T): Self has remove:[X:T,Result:set(T)] ifdef
          Result is set with list(Self..list..remove(X))
```

D.2.2.10.10 (at) set(T): list

The list property is overridden in order to indicate it is a uniqueness constraint.

```
     set(T): Self has list:( List:list(T) ) ifdef
          Self super has list: List
```

D.2.2.10.11 (at) set(T): rest

The rest of the members, all but the choice.

```
     set(T): Self has rest: (Xs:set(T)) ifdef
          Xs is set with list(Self..list..rest)
```

D.2.2.10.12 (at) set(T): uc1

The uniqueness constraint is on list.

```
set(T): Self has uc1: (List:list(T)) if_def
     Self..list is List..duplicateFree..sorted
```

D.2.2.10.13 (op) set(T): '+'

`Result` is the union of `Self` and `Set`.

```
set(T): Self has '+':[Set:set(T),Result:set(T)] if_def
     Result is set(X where Self has member:X or Set has member:X)
```

D.2.2.10.14 (op) set(T): '*'

`Result` is the intersection of `Self` and `Set`.

```
set(T): Self has '*':[Set:set(T),Result:set(T)] if_def
     Result is set(X where Self has member:X, Set has member:X)
```

D.2.2.10.15 (op) set(T): '-'

`Result` is the set difference, `Self minus Set`.

```
set(T): Self has '*':[Set:set(T),Result:set(T)] if_def
     Result is set(X where Self has member:X, not Set has member:X)
```

D.2.2.11 String

`String` is a primitive class with its `isEmpty`, `first`, `rest`, and `prefix` properties supplied by axioms.

D.2.2.11.1 (op) string: at

The string has character `C` at location `N`, where the first location is 0.

```
string: Self has at:[N:integer,C:character] if_def
     (Self has first:C, N is 0)
     or
     (Self has rest..at:[M,C], N is M+1)
```

D.2.2.11.2 (op) string: count

`Count` is the number of characters in the string.

```
string: Self has count: N if_def
     if Self..isEmpty
     then
          N is 0
     else
          N is 1 + Self..rest..count
     endif
```

D.2.2.11.3 (op) string: indexOf

Finds the index of a substring.

```
     string: Self has indexOf: [String:string,At:integer] if_def
          At is 0..to(Self..count – String..count),
          forall (I is 0..to(String..count - 1)):(Self..at(At+I) ==
               String..at(I))
```

D.2.2.11.4 (op) string: substring

Extracts a substring.

```
string: Self has substring: [At:integer,Count:integer, SubString:string]
     if_def
     pre (At >= 0),
     pre (Count >= 0),
     pre (At + Count <= Self..count),
     Acc is accumulator(initial:””,final:Subtring),
     for Acc all (I is 1..to(Count)):
          (acc..current is acc..previous..prefix(Self..at(At + Count - I))
```

D.2.2.11.5 (op) string: uc1

The uniqueness constraint is on `first` and `rest`.

```
     string: Self has uc1: [First:character,Rest:string] if_def
          Self..string is Rest..prefix(First)
```

D.2.2.11.6 (op) string: uc2

The uniqueness constraint is on `isEmpty`.

```
     string: Self has uc2 if_def
          Self..string is “”
```

D.2.2.11.7 (op) string: ‘+’

`Result` is the concatenation of `Self` and `Cat`.

```
     string: Self has ‘+’: [Cat:string,Result:string] if_def
          if Self..isEmpty
          then
               Result is Cat
          else
               Result is (Self..rest + Cat)..prefix(Self..first)
          endif
```