6-1-1996

Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications

Scott A. DeLoach

AFIT/DS/ENG/96-05

Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications

DISSERTATION

Scott Allan DeLoach
Major, USAF

AFIT/DS/ENG/96-05

AFIT/DS/ENG/96-05

Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications

DISSERTATION

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Scott Allan DeLoach, B.S., M.S.

Major, USAF

June, 1996

AFIT/DS/ENG/96-05

Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications

Scott Allan DeLoach, B.S., M.S.

Major, USAF

Approved:

Thomas C. Hartrum — May 16, 1996

Thomas C. Hartrum, Chairman

Paul D. Bailor — May 16, 1996

Paul D. Bailor

Mark E. Oxley — May 16, 1996

Mark E. Oxley

W. Jerry Bowman — May 16, 1996

Dean's Representative

[illegible]

Robert A. Calico, Jr

Dean, Graduate School of Engineering

Acknowledgements

I would like to thank my advisor, Dr. Thomas Hartrum, for his guidance and assistance during this research effort. I would also like to thank my committee members, Lieutenant Colonel Paul Bailor and Dr. Mark Oxley, for their advice and suggestions.

I would also like to thank my fellow graduate students, Captain Frank Young and Captain Robert Graham, for their discussions concerning both my research, that helped focus my efforts, as well as life in general, that kept me from getting "tunnel vision".

I also wish to thank my wife, Amy. Mere words are not enough to express the debt of gratitude and respect I have for her. She took on much more than her share of the burden to enable me to complete this effort. To my children, Vanessa, Lauren, Zachary, and Jordan, you are a constant reminder that my title of "daddy" will always be more important than "Major" or "Doctor". I love you all!

Finally, and most importantly, I thank the one who gave me the vision to pursue this goal and the strength to achieve it, my Lord and Saviour Jesus Christ.

Scott Allan DeLoach

Table of Contents

Page

Page

Page

List of Figures

Figure Page

List of Tables

List of Abbreviations

AFIT/DS/ENG/96-05

Abstract

Formal software specification has long been touted as a way to increase the quality and reliability of software; however, it remains an intricate, manually intensive activity. An alternative to using formal specifications is to use graphically-based, semi-formal specifications such as those used in many object-oriented specification methodologies. While semi-formal specifications are generally easier to develop and understand, they lack the rigor and precision of formal specification techniques. The basic premise of this investigation is that formal software specifications can be constructed using correctness preserving transformations from graphically-based object-oriented representations. In this investigation, object-oriented specifications defined using Rumbaugh's Object Modeling Technique (OMT) were translated into algebraic specifications. To ensure the correct translation of graphically-based OMT specifications into their algebraic counterparts, a formal semantics for interpreting OMT specifications was derived and an algebraic model of object-orientation was developed. This model defines how object-oriented concepts are represented algebraically using an object-oriented algebraic specification language O-SLANG. O-SLANG combines basic algebraic specification constructs with category theory operations to capture internal object class structure as well as relationships between classes. Next, formal transformations from OMT specifications to O-SLANG specifications were defined and the feasibility of automating these transformations was demonstrated by the development of a proof-of-concept system.

Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications

I. Introduction

1.1 Purpose

The insertion of traditional engineering methods has been suggested as the only way to transform software development from an art into an engineering discipline (26, 8). One approach to inserting this engineering discipline is the *transformational programming* paradigm where software is developed and maintained at the formal specification level and provably correct code is automatically derived from the specification (42). This paradigm is inherently knowledge-based and requires two types of software engineering knowledge: design knowledge and domain knowledge. Design knowledge incorporates the domain independent knowledge required to produce software including knowledge of architectures, algorithms, and data structures. Domain knowledge, on the other hand, captures knowledge about objects in the problem domain. In the transformational programming paradigm, software engineers use domain knowledge to derive system specifications from *domain models* and then use design knowledge to produce the code. Thus, software engineering evolves from the art of programming to the development of the domain models and design knowledge necessary to derive provably correct software (68:634).

Use of algebraic theories to represent software engineering knowledge has gained momentum during the last decade. Some of the most promising work is using theory-based specifications to drive software synthesis systems. A notable example of such a synthesis system, the Kestrel Interactive Development System (KIDS) (86), has yielded some exciting results. KIDS has been used to derive dozens of algorithms including a transportation scheduling algorithm for over 15,000 movements that was 78 percent faster and had 75 percent fewer delays than the best previously known algorithms (87). A follow-on effort to KIDS, Specware (55), supports a systematic approach to the composition of theory-based specifications followed by their stepwise refinement into code. The basic synthesis steps are to 1) develop a domain theory for the problem to be solved, 2) create a specification describing the problem in the language of its domain theory, 3) apply specification refinements to construct a program-based model of the problem specification, 4) apply program optimizations, and 5) compile the program (86).

While systems such as KIDS and Specware have been making progress in software synthesis research (steps 3 and 4 above), research in the acquisition of formal specifications (steps 1 and 2) has not been keeping pace. Formal software specification has long been touted as a way to increase the quality and reliability of software; however, it remains an intricate, manually intensive activity. Besides driving software synthesis systems as described above, justifications for using formal software specifications also include clarifying customer requirements, avoiding ambiguities and contradictions, and the ability to rigorously verify certain properties (29:74). In fact, experience in the use of formal methods suggests that the cost of using formal methods is no greater, and possibly even less, than using traditional software development methods (44:17). Yet, to date, formal methods are not widely accepted. Fraser suggests several reasons for this lack of acceptance and use (29:75,76):

1. Lack of research directed at developing practical methods and tools for incorporating formal methods into the software life-cycle. Most formal methods research is based on formal languages and inference rules.

2. Little expertise among practicing software developers in the mathematical and logical concepts and notations used in most formal specification languages.

3. Unsuitability of formal notation for communicating with end users since they are even less likely than the developer to be trained in formal mathematical and logical concepts.

4. Tendency of formal notations to inhibit creativity in a poorly defined problem areas.

5. Unwillingness of management to make a sizeable investment in what they consider to be an unproven technology.

So, what is the solution? Fraser suggests that a *Computer-Assisted Parallel Successive Refinement* methodology is the answer. In a computer-assisted parallel successive refinement method, the designers use both semi-formal (graphical) and formal representations to produce the specification by successively performing refinements to both representations in parallel (29:83). This methodology directly addresses items 1, 3, and 4 above and aids in resolving problems associated with 2 and 5. Both users and software personnel unaccustomed to formal specifications have access to the semi-formal, graphical representation while the formal representation, and its associated benefits, is maintained. The formal representation is used by knowledgeable software personnel and automated tools to check for ambiguities and contradictions and aid in transforming the specification into code. Unfortunately, Fraser reports that there has been no work in this area (29:83).

1.2 Overview

The basic premise of this investigation is that formal software specifications can be constructed using a computer-assisted parallel successive refinement approach incorporating correctness preserving transformations to automatically translate between graphically-based object-oriented representations and their corresponding formal representation. There are two obvious starting points for this investigation: object-oriented representations and formal representations.

Because a standard set of definitions does not yet exist for many terms and concepts in object-orientation, a specific object-oriented model was selected before continuing the investigation. There are several proposed object-oriented methodologies in use today (1, 4, 11, 13, 15, 19, 20, 21, 72, 73, 84, 98, 103). Rumbaugh's Object Modeling Technique (OMT) (83) was chosen for its breadth of coverage, availability of tools, and usefulness in domain analysis and modeling. Rumbaugh uses three distinct views to describe a domain: (1) the object model describes structural relationships between domain objects, (2) the dynamic model describes interactions between domain objects, and (3) the functional model describes how processes in the domain transform data. To enable automated translation of these models into a formal representation, a formal semantics for each model was developed. Then, an abstract syntax tree (AST) representation of a generic OMT specification was developed that captures the three OMT models in a single unifying structure.

Theory-based algebraic specification is concerned with (1) modeling system behavior using algebras (a collection of values and operations on those values) and axioms that characterize algebra behavior, and (2) composition of larger specifications from smaller specifications. Composition of specifications is accomplished via specification building operations defined by category theory constructs (88). In algebraic specifications, the structure of a specification is defined in terms of *sorts*, abstract collections of values, and *operations* over those sorts. This structure is called a *signature*. A signature describes the structure of a solution; however, a signature does not specify semantics. To specify semantics, the definition of a signature is extended with axioms defining the intended semantics of signature operations. A signature with associated axioms is called a *specification*.

Algebraic specifications are used in this investigation to define a theory-based model of object-orientation. Formal definitions for classes and objects were defined within this model. General class relationships were investigated resulting in the definition of formal techniques for the construction of valid inheritance, aggregation, and association relationships. An object-oriented algebraic specification language, O-SLANG, was developed that incorporates the category theory operations necessary to define relationships between object classes. Finally, formal translations were defined that map generic OMT specifications into O-SLANG algebraic specifications.

The feasibility of translating graphically-based OMT models into algebraic specifications was demonstrated by the development of an automated transformation system. The prototype system uses a commercial OMT modeling tool as the front end to a rule-based transformation system that generates O-SLANG based on the formal translations.

Related work is described in the next section while Section 1.4 defines the basic assumptions upon which this investigation is based, Section 1.5 describes the contributions of the investigation, and Section 1.6 outlines the sequence of presentation for the rest of this document.

1.3 Related Work

This section describes research related to my investigation. Section 1.3.1 describes proposed methods for incorporating formal specifications into current software development practices, Section 1.3.2 presents some formal specification languages used to describe object-oriented systems, and Section 1.3.3 describes existing transformation systems.

1.3.1 Formal Specification Incorporation Methodologies. Several authors have proposed techniques for incorporating formal methods into existing software development practices. Fraser et. al. created a framework for analyzing these methods (29). They categorize the methods by the *formalization process* and the *formalization support* of each of the methods. The formalization process can be defined as either being *direct*, where software developers move directly from informal (natural language) specifications to formal specifications without going through any semi-formal activity, and *transitional*, where the transformation from informal to formal specifications uses an intermediate semi-formal specification. Within the transitional process, Fraser defines two subtypes: sequential and parallel successive refinement. In a *sequential* transitional approach, the semi-formal specifications are fully defined and then transformed into a formal specification. In the *parallel successive refinement* approach, the semi-formal and formal specifications are produced simultaneously, going through equivalent successive refinements. Formalization support is also divided into two categories: *unassisted*, where all work is done manually, and *computer assisted* where computer-based heuristics or knowledge-based transformation tools assist the developer. I briefly discuss a number of methodologies and then show how they fit into Fraser's framework.

Andrews and Gibbons describe a methodology where Structured Analysis is used to build a hierarchical system structure chart. This structure chart is then translated manually into VDM. Once in VDM, the structure chart is used to guide the decomposition of operations and data refinements (29).

Babin, Lustman, and Shoval propose a computer assisted method based on the ADISSA method, which is an extension of Structured System Analysis. In ADISSA, the system architecture is described via a set of transactions that model system events and user requests. Flow of control is modeled by a finite state machine. The method uses a ruled-based transformation system to help transform the semi-formal specification into a formal specification (10).

Conger et. al. developed a manual procedure for taking Structured Analysis data flow diagrams and transforming them into VDM. First, hierarchical data flow diagrams are developed using Structured Analysis heuristics. The data flow diagrams are then used to guide the developer in partitioning and stepwise refining of the VDM specifications. A VDM specification is produced for each data transformation process in the data flow diagram set (22).

Fraser, Kumar, and Vaishnavi propose an interactive, rule-based transformation system to translate Structured Analysis specifications into VDM specifications. The method is based on data flow diagrams and decision tables produced via Structured Analysis. The bottom level data flow transformation processes are defined by decision tables which are then transformed by the rule-based system (28).

Kemmerer describes a process which integrates traditional software development with development using formal methods. In this process, the formal methods are annotations to the semi-formal design and are developed directly from lower-level natural language descriptions (52).

Kung has developed Process Interface Modules which are formal descriptions of the communication and synchronization among processes. Basically, the system is initially developed using entity relationship diagrams to describe the static parts of the system and data flow diagrams to model the dynamic parts of the system. This set of diagrams is then manually transformed into Process Interface Modules to provide formal proof-based checking of the semi-formal design (57).

Miriyala and Harandi have developed an automated tool to help interactively create formal specifications directly from informal requirements specified in a natural language subset. The tool develops a problem structure tree directly from the informal specification and uses domain independent knowledge and analogy with past developments to guide the refinement of the tree (74).

Wing proposes a very vague method where informal requirements are transformed into a formal specification through a series of iterative interviews with the user. She proposes neither a semi-formal method to communicate with the user nor a specific formal specification language to work toward (102).

Table 1.1 classifies the methodologies discussed above according to Fraser's framework. As shown, and as reiterated by Fraser, the lower right hand corner, *computer assisted, parallel successive refinement* methodologies, has seen no research to date. According to Fraser, work in developing techniques and methodologies for this area is needed because of the promise of the approach (29:84).

1.3.2 Formal Specification Languages.

1.3.2.1 Z Extensions. There have been a number of Z extensions designed to make Z specifications easier to understand. Many of these extensions are object-oriented. Although most of the object-oriented Z extensions provide techniques for structuring the Z specification using the common object-oriented concepts, they do not attempt to provide an improvement in specification

			Formalization Support	
			Unassisted	Computer Assisted
Formalization Process	Direct		Kemmerer Wing	Miriyala and Harandi
	Transitional	Sequential	Andrews and Gibbins Kung	Babin, Lustman and Shoval Fraser, Kumar and Vaishnavi
		Parallel Successive Refinement	Conger et al.	

Table 1.1 Formal Methods Incorporation Strategies

development methodology. A number of Z extensions are discussed below. However, to avoid repetition, only the unique features of each language are discussed.

MooZ. MooZ is an object-oriented extension of Z. A MooZ specification is defined as a set of MooZ class specifications which are semantically described as records. These classes are then used to define data types or generic templates from which to instantiate objects. MooZ classes support parameterization and multiple inheritance. Attributes determine the state of an object and these attributes may be visible or hidden. Operations that manipulate these state attributes are handled at the individual object level; however, operations such as create or destroy, which do not reference state variables, are handled by the class. Operations and attributes are implemented via functions that are defined in the Z axiomatic style (62).

Object-Z. Object-Z is another object-oriented extension to Z. Like MooZ described above, Object-Z uses a class definition which includes attributes and operations which are defined using Z axiomatic definitions. One unique aspect of Object-Z is its use of a *history invariant.* The history invariant uses linear temporal logic to further restrict valid object behavior by defining legal sequences of operation calls (17). Object-Z also includes *referential semantics* which allows the declaration of references (pointers) to objects. These semantics are particularly useful in creating aggregates where one might define an object as a variable, or even define a set of objects. Object-Z also provides a *promotion* operation which allows the operation of an aggregate component to be promoted to an operation of the aggregate class (82:114).

Z++. Z++ is another object-oriented extension based on Z. However, Z++ is unique in that its semantics is based on algebra and category theory. In Z++, classes form a category with Z++ class refinements defined as the arrows. The formal algebraic and categorical basis makes it possible to prove specific properties about Z++ specifications (60:28). To keep Z++ consistent with the Z community, a model-based theory has been developed to enable reasoning using either algebraic or model-based semantics (58). While Z++ semantics are based on algebra and category theory operations, Z++ does not use category theory operations to build new specifications or compose new specifications from existing specifications. Z++ must be written manually. Category theory is only used to prove correctness of specification structure.

1.3.2.2 OBJ3, OOZE, and FOOPS. Despite its name and its use of the term "objects", OBJ3 is not an object-oriented language; however, it has been extended to include object-oriented concepts and does have many interesting aspects. Actually, OBJ3 is not simply a language, but a system which includes a *functional* programming language, environment, and interpreter for algebraic specifications (40). OBJ3 includes three types of components: objects, theories, and, views. Objects are not objects in the object-oriented sense, but are actually purely functional executable code modules. Theories, on the other hand, are similar to the concept of theories as defined in Section 3.2 in relation to algebraic specifications. Views are used in OBJ3 to relate how objects (modules) satisfy axioms of a particular theory and are defined as theory morphisms (38:436). Although OBJ3 uses theories, it was developed to perform code-level reuse by composing existing code and not for transformational derivation of code from specifications.

OBJ3 is based on order-sorted algebras which is an extension of many-sorted algebras (39). Basically, an order-sorted algebra is a many-sorted algebra with a partial order defined on the sorts. According to Goguen, order-sorted algebras are used to allow sorts to be of two different types (40:4). For example, a natural number is an integer, an integer is a rational number, and a rational number is a real number. This gives a subsort partial ordering of $natural \leq integer \leq rational \leq real$. This allows the definition of total functions on subsorts that otherwise would be defined as partial functions.

As a final note, and to make a clear distinction between OBJ3 and object-oriented languages, Goguen states that OBJ3 objects do not have states (40:37) which is a critical aspect of object-orientation. Although it provides many of the same operations necessary for class inheritance and parameterization, without providing state it can only produce functional specifications.

OOZE (Object-Oriented Z Extension) (6, 7) and FOOPS (Functional Object-Oriented Programming System) (37) are object-oriented specification languages based on OBJ3. Because OOZE is a syntactic variant of FOOPS and has the same semantics (5:181), I only discuss FOOPS here. FOOPS is an algebraic, object-oriented specification language based on OBJ3. It provides classes (possibly parameterized), objects, inheritance, attributes, and methods. Objects are the instances of a class, and each class may have a number of objects. Each object has a unique identifier, a set of observable attributes, and a set of methods which change the state of an object (38:441).

1.3.2.3 Larch. Larch is a two-tiered algebraic specification language based on multi-sorted first-order logic with equality (43:8). Each specification has two parts, a language specific specification written in one of a number of *Larch interface languages* (LIL), and a language independent part written in the *Larch Shared Language* (LSL). Larch is not intended to be object-oriented although object-oriented designs have been mapped into Larch (65). LSL specification entities, called traits, define a set of operations over a set of sorts.

LSL traits can be parameterized on sorts and sorts are defined by their use in defining operations. Larch is designed to be used to prove semantic properties about specifications and not to be transformable into code or to be executable; therefore, Larch provides an *implies* section of a trait to make explicit claims about theory containment (i.e., what theorems are logical consequences of the trait assertions). These claims must be proved by the author and are useful in error detection and reader understanding.

The Larch interface languages are used to define the mapping between an actual programming language and an LSL trait. Whereas the traits are used to define the functional aspects of the operations, an interface specification provides the programming language interface to the trait and models *state* as represented in the programming language. Usually this state modeling involves specifying the state before and after each operation.

1.3.2.4 Slang. Slang is an algebraic specification language where specifications are theory presentations using higher order logic extended with category operations such as products, coproducts, quotients, and subsorts (54). System-level specifications are developed by building diagrams which can be used to express parameterization, instantiation, importation, and refinement (50). Specification building operations include translation, colimit, and importation, each of which defines a morphism (or in the case of a colimit, morphisms) between a source specification and a target specification. Translation copies a specification while renaming some or all of the sorts or operations. The colimit operation takes the colimit of a number of specifications and morphisms between them, combining them over any "shared" sorts and operations, into a single specification. Importation involves including another specification – sorts, operations, and axioms – into a new specification where additional sorts, operations, and axioms can be defined.

Diagrams are critical to specification development in Slang. In Slang, diagrams are "a multi-directed graph whose nodes are labeled with specifications and whose arcs are labeled with morphisms" (54:18). In a diagram definition, the arcs define the morphisms between the specifications.

Cocone-morphisms are created by a colimit operation while *import-morphisms* and *translation-morphisms* are created via the import and translate operations respectively.

The colimit, import, and translate operations available in Slang are similar to the constructs used in OBJ3 to construct specifications while the ability to refine specifications, using *interpretations* built into Slang provides an even more powerful framework for deriving correct programs directly from the specifications. An interpretation from specification B to specification A provides a mechanism for constructing a model of A from models of B. Thus if we have a model for specification B and can construct an interpretation from B to a A, we can create a model for A as well.

1.3.3 Transformation Systems.

1.3.3.1 Bourdeau and Cheng.

Bourdeau and Cheng (14) have developed formal semantics for an extended version of the OMT object model notation using the Larch specification language (43) to describe modular algebraic specifications. The object model itself is a specification which simply "includes" the object classes and associations derived from the object diagram.

Classes and associations are defined using a set of semi-formal rules. According to these rules, each class defines a Larch specification called a *trait.* Within a class trait, sorts are introduced to represent objects in the class as well as the state of an object in the class. A special state evaluation function is added to map a given object to a value in the state sort. For each attribute in the class, a function is defined which takes an object value and returns the value associated with the object.

Relational aspects of the object model such as association, aggregation, and inheritance are defined as predicates. Each component in an aggregate relation defines a *has-part* predicate that, given two objects, determines if the two objects are in the relation. Likewise, each association defines a predicate R in an association trait that determines if two objects are related via the association. Aggregate and association multiplicities are defined as axioms over the appropriate predicates.

Inheritance is modeled by defining a *simulates* operation that takes an object of the subtype and produces an object of the supertype, thus implementing Bourdeau and Cheng's interpretation of the *substitution property.* For their definition of inheritance, Bourdeau and Cheng assume that a subclass object D must be substitutable for its superclass object C at any point in the object's lifetime. To satisfy this notion of inheritance, Bourdeau and Cheng define constraints on the *simulates* operation that requires all states in the subclass to map some state in the superclass and that for all attributes defined in the superclass, the values of those attributes in the subclass are allowable values in the superclass.

Once a specification for an object model is derived, Bourdeau and Cheng use OMT *instance models* (diagrams which show how a particular set of *objects* relate to each other) to define a set of algebras. They define the semantics of the object model as the complete set of instance diagrams that are consistent with the object model specification.

1.3.3.2 Rafsanjani and Colwill. Rafsanjani and Colwill have defined a mapping from Object-Z to C++ in the context of an abstract object model (81). They define their mapping informally without thought toward automation and their approach is based on empirical cases and is not theoretically well founded (i.e., there is no proof that the C++ implementation does in fact correctly represent the specification). Their mapping is purely structural. There is no attempt to transform the semantics of the specification into code. This is left as a "creative" exercise for the programmer.

Although the goals of their research differ from this investigation, their work has some interesting aspects. First is the use of an object model to capture features common to both languages. The object model allows them to relate concepts between the two languages and define a mapping. Although Object-Z operations are mapped to *virtual* C++ functions which allow the inheriting class to modify the functions defined in the parent class, Rafsanjani and Colwill view inheritance strictly, allowing only for extension or restriction. Unfortunately, this view of inheritance is only enforced by the good will of the programmer. Object-Z predicates are used to represent restrictions on attributes and state spaces as well to define operation pre- and post-conditions. Rafsanjani and Colwill map these predicates to C++ functions which are invoked before and after each operation to ensure the predicates remain true.

1.3.3.3 KIDS. The Kestrel Interactive Development System (KIDS) is a prototype system that provides semi-automatic derivation of programs from algebraic specifications. KIDS maps program specifications to algorithm theories to instantiate a functional program and then uses high-level optimization operations to produce an efficient and correct-by-construction program (with respect to the initial specification) (53). Operations provided by KIDS include algorithm design, deductive inference, context independent and context dependent simplification, partial evaluation, finite differencing, and compilation. KIDS is based on the use of algebraic theories and category theory operations. Theories are used in KIDS to encapsulate knowledge about problems in general, knowledge about the problem being solved, general knowledge about the application domain, and general programming knowledge. Category theory concepts and operations such as pushouts, colimits, and morphisms are used to combine and refine these theories into efficient programs.

The basic steps in deriving a program in KIDS are 1) to develop (or reuse) a domain theory for the problem to be solved, 2) create a specification that describes the problem to be solved in the language of its domain theory, 3) apply a design tactic which forms an interpretation of the problem specification in general algorithm theory and instantiates a program, 4) apply optimizations to the program, and 5) compile the program (86:1025). Application of this approach has yielded some impressive results. KIDS has been used to derive dozens of algorithms including real-world systems. KIDS has produced a transportation scheduling algorithm for over 15,000 individual movements that was 78 percent faster and produced 75 percent fewer delays than the best algorithms previously known (87:66). KIDS has shown that not only is transformational program derivation possible, but it can produce more efficient and more reliable software.

1.3.3.4 Specware. Specware (55) is a transformational program derivation system based on Slang (54) and KIDS. Specware extends the concepts used in KIDS by allowing the developer to build *diagrams* of specifications to build up a domain theory and eventually a system specification. Basically, Specware provides the automated tool support for developing specifications in the Slang specification language described above. When completed, Specware will incorporate facilities to provide algorithm design and optimization, data type refinement, integration of reactive system components, and code generation.

1.4 Assumptions

Because this research involves the use of specifications entered by an unknown user, two assumptions are made concerning the specifications entered.

Assumption I.1 Initial Specification Consistency. *All specifications, as entered by the user, are correct and consistent.*

If a specification is not internally consistent, then valid models of those specifications do not exist (32:3-13); therefore, internal consistency is a requirement for formal software synthesis. Unfortunately, Church and Turing independently showed that in general, proving that a set of first order axioms are inconsistent is not possible (18:45). Therefore, in this research, I assume that user provided specifications are consistent and only show that further specification composition operations (inheritance, aggregation, etc.) maintain that consistency.

Assumption I.2 Restricted Use. *All OMT models developed by the user are developed in accordance with the restricted models as defined in Chapter V.*

As defined by Rumbaugh, OMT has numerous ways to specify the same features using formal and informal techniques. In this investigation, specification using informal techniques are inadequate for automatic translation while the ability to specify the same functionality using multiple techniques ultimately leads to consistency questions. Therefore, Chapter V defines a restricted version of OMT's three models and limits how they are used.

1.5 Contributions

Based on these assumptions, the contributions of this research include:

1. Development of an algebraic, category theory based specification language with built-in constructs for object-oriented concepts such as classes, inheritance, aggregation, association, and global event communication.
2. Formalization of basic object-oriented concepts using algebraic and category theory constructs.
3. Formalization of a generally accepted notion of class inheritance and a sufficiency criteria for proving adherence to that formalization.
4. Formalization of the semantics of the object, dynamic, and functional OMT models.
5. Formalization of event-based communications paths within an OMT domain specification.
6. Formalization of translations from graphically-based object-oriented representations to algebraic specifications.
7. Elevation of the level of abstraction at which formal specifications are developed.
8. Development of techniques to ensure consistency of object-oriented specification composition.
9. Elevation of the acceptance of formal specifications and methods.

1.6 Summary

This chapter is an introduction to the goals and objectives of my investigation and a brief overview of related research. The remainder of this dissertation is organized as follows:

- Chapter II presents a framework for the parallel acquisition of theory-based specifications using graphically-based object-oriented concepts.
- Chapter III discusses basic algebraic specification construction techniques within a category theory setting.
- Chapter IV establishes the foundations for a theory-based model of object-orientation.
- Chapter V defines the formal semantics for the OMT object, dynamic, and functional models.
- Chapter VI introduces a theory-based model of object-orientation based on the OMT object, dynamic, and functional models.
- Chapter VII describes the formal translations from a generic OMT specification to a theory-based specification.
- Chapter VIII demonstrates the feasibility of automated specification translation by producing two theory-based domain specifications using an automated proof-of-concept system.
- Chapter IX contains the conclusions from this investigation and provides recommendations for future research.

II. Software Development and Specification Acquisition Framework

2.1 Overview

This chapter defines a theory-based Specification Acquisition Mechanism based on an object-oriented user interface and theory-based specifications. The basic concept is to allow system developers to graphically specify domain, architecture, and system-level details in an object-oriented fashion and automatically convert them into theory-based algebraic specifications. Section 2.1.1 describes the basic software development framework into which the theory-based Specification Acquisition Mechanism fits while Section 2.1.2 presents an overview of the Specification Acquisition Mechanism itself. Sections 2.2, 2.3, and 2.4 further define specific subsystems of the Specification Acquisition Mechanism.

2.1.1 Software Development Framework. A framework for the development of software using semi-automated software synthesis from theory-based system specifications is shown in Figure 2.1.

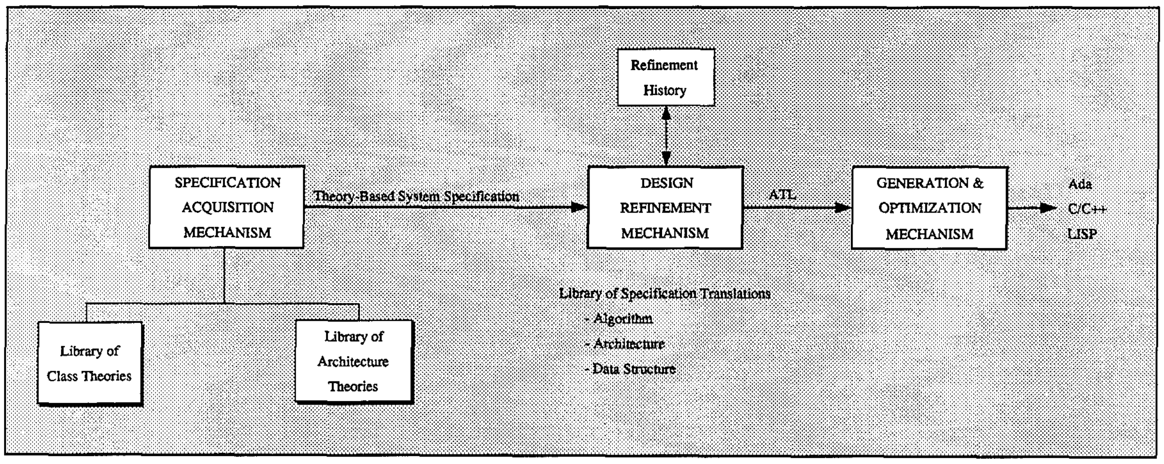
Refinement History
SPECIFICATION ACQUISITION MECHANISM
Theory-Based System Specification
DESIGN REFINEMENT MECHANISM
ATL
GENERATION & OPTIMIZATION MECHANISM
Ada
C/C++
LISP
Library of Specification Translations
- Algorithm
- Architecture
- Data Structure
Library of Class Theories
Library of Architecture Theories

Figure 2.1 Software Development Framework

The central theme behind the proposed software development framework is the synthesis of software from theory-based specifications. Functional specifications are developed and combined with architecture theories in the Specification Acquisition Mechanism to create theory-based system specifications. These specifications are then fed into the Design Refinement Mechanism where the sorts, operations, and architectures are refined and mapped to an intermediate abstract target language (ATL) representation. This ATL is then converted into compilable source code and optimized in the Generation & Optimization Mechanism. This research focuses mainly on the Specification Acquisition Mechanism and the Library of Class Theories. The Design Refinement and Generation & Optimization mechanisms are only discussed informally in this chapter and left for future research. This research focuses on one approach to the Specification Acquisition Mechanism.

2.1.2 Specification Acquisition Mechanism. The basic functions and data flows of the proposed Specification Acquisition Mechanism are shown in Figure 2.2. There are three phases to acquiring a theory-based system specification using this mechanism: Domain Engineering, Specification Generation, and Specification Structuring. Each phase of specification acquisition has an associated theory-based subsystem that accesses the Theory Library. However, to simplify specification acquisition, the domain engineer or system analyst uses a graphically-based object-oriented interface. For the purposes of researching and prototyping, I have chosen to base my object-oriented approach on the Rumbaugh's OMT. OMT was chosen due to its popularity, breadth of coverage, and availability of tools. I do not claim that OMT is the best, or even better than other object-oriented methods or techniques.

The Specification Acquisition Mechanism is designed to help produce consistent theory-based domain models which are then refined into functional system specifications. (I use the term "functional specification" here to describe the definition of the system functions. It does not imply a shift from an object-oriented to a functional view of the system.) A domain model is the "specific representation of appropriate aspects of an application domain" (48) and may take on various forms including domain taxonomies, generic architectures, and domain specific languages (80). In this mechanism, a domain model is captured via definition of the basic objects, operations, and communication paths in the domain. Object-oriented constructs such as inheritance and aggre-

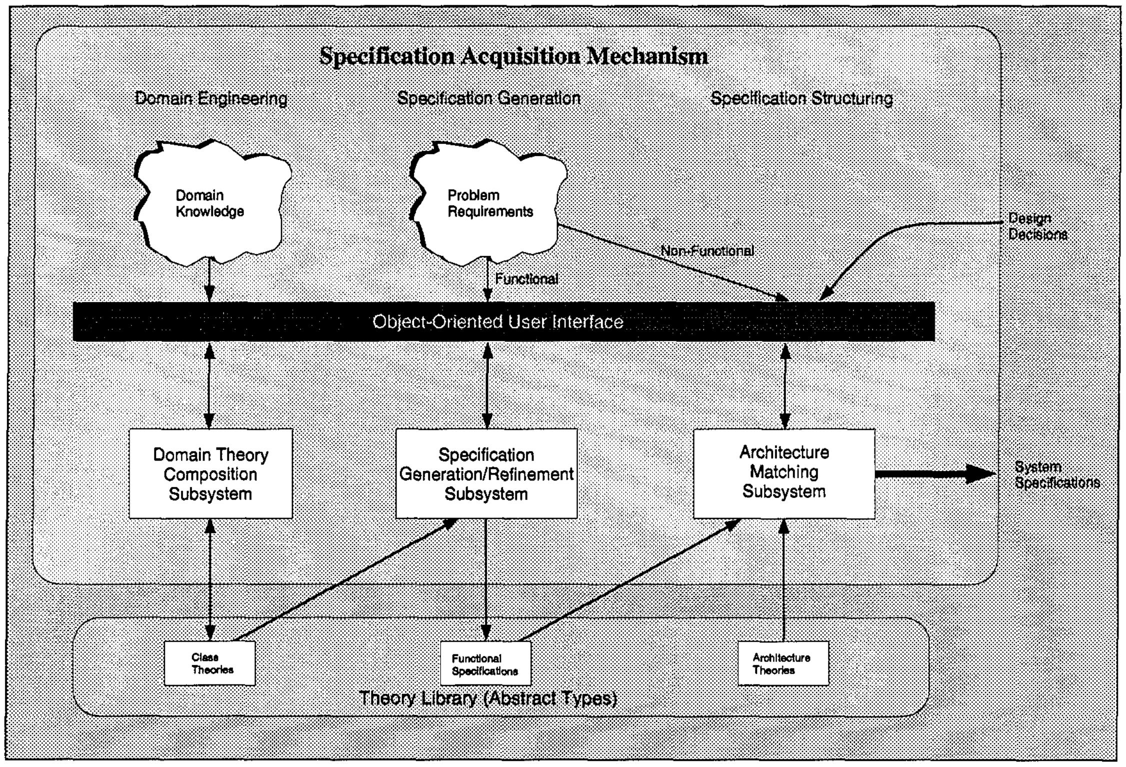
Specification Acquisition Mechanism
Domain Engineering
Specification Generation
Specification Structuring
Domain Knowledge
Problem Requirements
Non-Functional
Design Decisions
Functional
Object-Oriented User Interface
Domain Theory Composition Subsystem
Specification Generation/Refinement Subsystem
Architecture Matching Subsystem
System Specifications
Class Theories
Functional Specifications
Architecture Theories
Theory Library (Abstract Types)

Figure 2.2 Parallel Refinement Specification Acquisition Mechanism

gation are used to model generalization and specialization of domain objects as well as how they are composed into other domain objects. The domain model thus defines a language from which systems are specified. A functional specification is developed by refining a domain model through selection of applicable domain objects, instantiation of domain object parameters, and definition of specific communications paths between system objects. Once the system function is fully specified, the system specification is completed by determining how to decompose the system into separate processes and how those processes communicate.

Domain models are created by domain engineers using knowledge from domain experts and stored in the Theory Library. Domain models consist of a set of *class theories*, which describe the objects, operations, and communications paths of a domain. Individual class theories may describe the attributes, methods, states, and events of a group of similar objects in a domain or the relationships and communication paths between objects.

Class theories from a given domain are refined by users and system analysts to create *functional specifications* which are also stored in the Theory Library. Functional specifications have the same syntactic form as class theories but are tailored toward the requirements of a specific problem. Intuitively, functional specifications are models of a domain class theory.

Architecture theories describe the structure of systems in terms of processes and inter-process communications and are also stored in the Theory Library. Generally, while class theories are domain specific, architecture theories are domain independent–they are defined solely in terms of processes and inter-process communication.

Once a functional specification has been completed, the system analyst selects (or develops) a corresponding architecture theory from the theory library. The architecture theory's process parameters are then instantiated with class theories from the functional specification to form a *system specification.* For example, the system analyst may instantiate an architecture theory with three processes running in parallel with refined versions of a propulsion device class theory, airframe class theory, and a fuel-tank class theory from a rocket system domain to create a rocket system specification. In essence, a system specification is a model of a generic architecture theory instantiated with domain specific class theories.

Although the domain engineer or system analyst is producing theory-based models and specifications, interaction with the system is through a conceptually simpler object-oriented interface. Ideally, this interface consists of a graphical user interface with which the engineer or analyst specifies domain object classes, associations, and architectures to define the domain or architecture structure. Object attributes and operation semantics are then specified algebraically or graphically. Structures representing behavior, such as state charts and data flow diagrams, are automatically translated into equivalent algebraic definitions.

As the engineer or analyst interacts with the object-oriented user interface, commands and data are translated to the theory-based subsystems where the actual composition occurs and proofs of consistency and correctness are performed. For example, a domain engineer may create a new class that inherits from an existing class. As the domain engineer creates the new class, attributes, and methods, the *Specification Acquisition Mechanism* carries out proof obligations to show that the composition specified by the domain engineer satisfies appropriate composition rules defined by the theory-based object model in Chapter VI.

After the domain engineer creates a domain model and proves the correctness of its composition, a system analyst uses it to produce a functional specification. Again, the system analyst works with an object-oriented representation of the system while the operations and proofs are carried out on the theory-based specification. When complete, the functional specification is combined with an architecture theory to define a theory-based system specification which is fed into a correctness preserving *Design Refinement Mechanism* which derives code satisfying the specification. Since the domain and architecture models are stored in the Theory Library, maintenance is performed by modifying the class and architecture theories and re-deriving the system specification and software.

2.2 Domain Engineering

Domain engineering is the process of developing domain models for use in constructing applications within the domain. This section presents a basic overview of domain engineering in Section 2.2.1 followed by how the Specification Acquisition Mechanism proposed in this research implements domain engineering in Section 2.2.2.

2.2.1 Overview. The term *domain analysis* was first used by Neighbors to describe "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain" (78); however, since that time the definition of domain analysis has been expanded and applied to more than just identifying objects and operations in a given domain. The size of a domain may vary from very large and complex to very simple. The domain "avionics" is fairly large and complex while the domain "basic logic operators" is straight forward and comparatively small. Most domain analysis approaches are based on *application area domains.* An application area domain is a domain where the applications define the domain (e.g., stack packages, basic logic operators, etc.) (97). This is the definition used in this research. While many definitions abound ((51, 9, 77, 79, 97)), perhaps the simplest and most straight-forward definition is the process where "domain knowledge is studied and formalized" (96).

The goal of domain analysis is to capture knowledge in order to reuse it in developing new systems. After the domain knowledge is gathered, it is stored in a domain model for use in new software development efforts to increase the productivity and quality of new systems. As such, domain analysis is just part of an overall process called *domain engineering* which "includes domain analysis and subsequent construction of components, methods, and tools that address the problems of system/subsystem development through the application of domain analysis products" (51). Domain engineering has three steps: (1) domain analysis, (2) infrastructure specification, and (3) infrastructure implementation (9). (An infrastructure is domain knowledge along with information on how to find and use that knowledge).

A *domain model* is used to represent domain knowledge and is defined as the "specific representation of appropriate aspects of an application domain" (48). A domain model may take on various forms including domain taxonomies, generic architectures, and domain specific languages (80). Domain models are the end product of domain analysis and contain all the knowledge gathered during domain analysis including software architectures. *Software architectures* consist of "the components, connections, constraints, and configurations of components and constraints that specify the high-level design for a system" (93). Basically, the architecture is a blueprint for composing applications given well-defined domain-specific components. Specification of generic domain software architectures is critical to the automatic generation of domain-specific applications.

2.2.2 Implementation. As shown in Figure 2.2, domain engineering is the first phase in system specification acquisition and is the main focus of this research. Domain knowledge is input through an object-oriented user interface, translated to theory-based specifications, and stored in the Theory Library by the *Domain Theory Composition Subsystem.* This component translates graphically oriented OMT specifications, augmented with first-order logic axioms, into theory-based class theories. A particular domain analysis methodology is not specified for use with the subsystem; however, the chosen methodology should be compatible with capturing domain knowledge in an object-oriented setting.

The heart of the translation process from OMT to class theories is a Theory-Based Object Model as defined in Chapter VI. This model defines object-oriented concepts in a formal framework based on algebraic specifications and category theory, thus providing a powerful ability to reason about the resulting specifications. Due to the undecidability of first-order axiom consistency, axioms entered into the subsystem are assumed consistent; however, composition rules are used to ensure inconsistencies are not introduced during the composition process. It is also in this subsystem that completeness proofs are performed. These proofs show that the effect of each operation on a given object is completely defined by the domain engineer.

The output of the Domain Theory Composition Subsystem is a theory-based domain model. An example of the object-oriented view of a domain model for a simulated rocket is shown in Figure 2.3. The simulated rocket domain consists of three types of components: airframes, fuel tanks, and propulsion devices. Characteristics of all airframes and propulsion devices are defined in the Airframe and Propulsion Device classes. Characteristics of particular airframes and propulsion devices are defined in specializations of those object classes. The relationship between a fuel tank and a propulsion device is defined in the Feeds association. The basic structure of a simulated rocket object is described by the aggregation of, and multiplicities defined by the airframe, fuel tank, and propulsion device components; however, this structure does not define the process-based architecture. This domain model is refined and implemented in a number of ways. The system analyst may define the system as a single process, as multiple processes (one for each object class), or any combination in between. These decisions are made during the specification generation and structuring phases described below. Only the functionality of basic domain objects and their specializations are defined during the domain engineering phase.

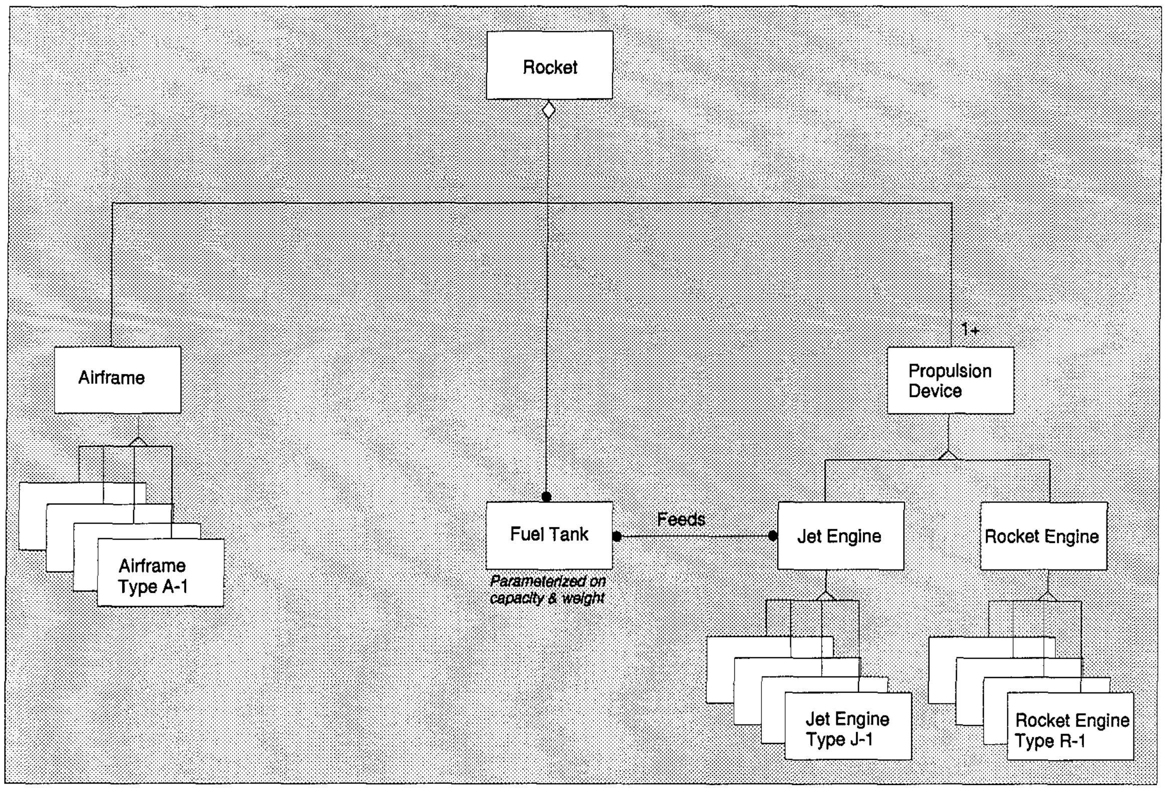

Figure 2.3 Rocket Object Domain Model

2.3 Specification Generation

Specification generation transforms domain models into specifications defining the functionality of a particular system in the domain. Functional specifications developed through the *Specification Generation/Refinement Subsystem* have the same theory-based syntax as the domain model's class theories. The basic result of specification generation is to create a system specification based on the domain model and then to restrict the number of models satisfying the specification. Specific generation/refinement operations include:

1. parameter instantiation

2. specialization selection
3. multiplicity restriction
4. initialization definition
5. communication path definition
6. constraint restriction

Parameter instantiation is the selection of values for predefined class parameters. These parameters can be single values or ranges of values. In either case, parameter instantiation results in the addition of axioms to the class definition that restrict object behavior. *Specialization Selection* allows the system analyst to select a particular class specialization. Because a domain theory models the entire application domain, it includes various class specializations in order to capture important variations in structure and behavior. Specialization selection simplifies and restricts the system specification by removing unwanted specializations. *Multiplicity Restriction* allows the system analyst to explicitly decide how many objects are allowed in aggregates and associations. Often, domain engineers define few restrictions on object class relationships (i.e., they allow "zero or more" objects in any given relationship). While this generalizes the domain model, actual systems within the domain are more restrictive. Multiplicity restriction simplifies the specification by placing additional constraints on aggregate and association multiplicities. *Initialization Definition* allows the system analyst to state specifically how objects are initialized. For instance, while a domain model may state that a one-to-one association exists between two classes, initialization defintion allows the system analyst to specify which objects from the two classes are associated. A similar situation exists for aggregates. Often, the question of whether an aggregate creates its components upon initialization or they are created separately is left unspecified in the domain model. Initialization defintion allows the system specifier to specify these requirements. *Communication Path Definition* allows the system developer to specify exactly which objects require communication. In the domain model, the domain engineer is concerned with specifying the classes of objects that may communicate, not which specific objects actually do communicate. In many cases, even though an object can communicate with all objects in a given class, it may only need to communicate with one specific object from that class. These constraints are added via communication path definition. The last generation operation, *constraint restriction* is a generic refinement operation. Constraint restriction allows the system analyst to place further constraints on class behavior through the introduction of axioms. These axioms are used to place restrictions on attributes or define restrictions across aggregate components or associations.

The specialization of the Rocket Domain Model, shown in Figure 2.3, to the Rocket Object Model, shown in Figure 2.4 is a simple example of the first four generation/refinement operations. First, the system analyst selects one Jet Engine class from the Jet Engines available in the domain model. After selecting a Jet Engine, the system analyst removes all other Propulsion Devices (jet and rocket engines) and restricts the 1+ multiplicity constraint between the Rocket and the Propulsion Device to be exactly two. Because the system being designed has exactly one Fuel Tank for each Jet Engine, the system analyst must also restrict the multiplicity constraint between the rocket and Fuel Tanks from zero-or-more (●) to exactly two and restricts the Feed association between the Fuel Tank and Jet Engine to one-to-one instead of many-to-many. The system analyst might then select the ◆ aggregation symbol (an extension to Rumbaugh's OMT notation) to specify that the rocket object creates its component objects upon creation. To complete the Fuel Tank specialization, the system analyst also provides Fuel Tank parameters. Finally, the system analyst selects a specialization from the Airframe class and completes any remaining Rocket-level constraints by providing bindings for any Rocket class parameters.

Further system-level functional constraints are captured in the system specification through constraint restriction. Communication path definition is not included in this example due to the complexity of inter-object communication (Section 6.6); however, if a Fuel Tank is required to communicate with its Jet Engine, the system analyst may specify that the Fuel Tank communicate only with the Jet Engine to which it is associated via the Feeds association.

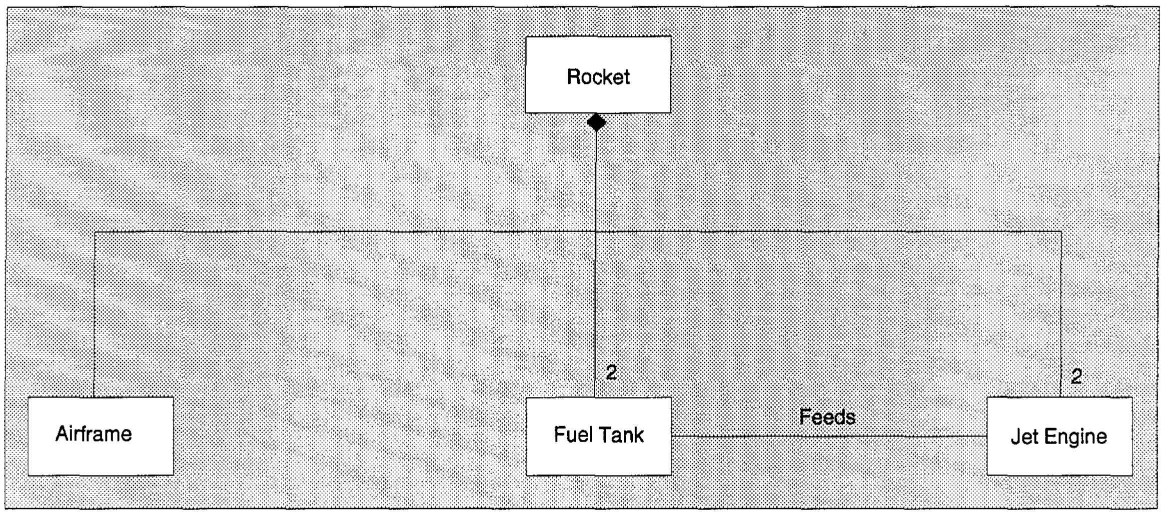

Figure 2.4 Specific Rocket Object Model

2.4 Specification Structuring

Specification structuring allows the system analyst to define the structure of the application in terms of process and inter-process communications. This phase is used to decompose specifications into simpler, less complex specifications, or, to build up larger specifications from smaller specifications.

An *Architecture* is a collection of objects (in this case, class theories) along with a relation over the objects defining object composition (e.g., parallel, sequential, etc.). Formally, an *Architecture Theory* defines a collection of objects, a set of relations over those objects that define the syntax of object composition, and a collection of axioms over the objects and relations that define the semantics of the architecture. An architecture theory consists of a *structuring specification*, which describes how the processes are composed (e.g., sequential, parallel, etc.) and a *diagram* that specifies how to construct the final system specification. Thus an architecture defines the objects of interest and the composition rules for these objects(32).

In the Specification Acquisition Mechanism, an architecture theory is a parameterized specification that defines the structure of an application in terms of processes and communication channels between the processes. The parameters of the architecture theory are class theories, each of which becomes a process in the system. The architecture theory also includes a diagram defining exactly how the architecture theory is to be parameterized and how the final system specification is composed. For example, a parallel architecture theory is defined as a set of class theories, the parallel process composition operator,

$$\_ \parallel \_ : \textbf{process}, \textbf{process} \rightarrow \textbf{process}$$

and a satisfaction relation, $\models$, which defines the relationship between class specifications and their models (32:6-3).

The Architecture Matching Subsystem is used to bind architecture theories to functional specifications. The system analyst selects class theories from the functional specification and matches them to processes in a predefined architecture theory. Each class theory (or group of class theories) in the functional specification must correspond to a process while all communication between class theories or class theory groups must correspond to a communication channel in the selected architecture theory. This is an example of an *imposed* architecture (32:2-8). In this research, I assume the existence of architecture theories and the ability to create them using graphically oriented techniques, and instead focus on the acquisition of class theories in the form of domain models.

2.5 Summary

This chapter has presented an overview of a theory-based specification acquisition system set in a software development framework. The software development framework uses three components to synthesize software: (1) a Specification Acquisition Mechanism to help develop theory-based system specifications, (2) a Design Refinement Mechanism which uses algorithmic, architectural, and data structure refinements to the system specification to produce an implementation in an abstract target language, and (3) a Generation & Optimization Mechanism which converts the abstract program into an optimized program in a compilable target language.

The Specification Acquisition Mechanism is used to develop large-scale theory-based system specifications. First, a domain engineer develops a domain model using an object-oriented interface to the Domain Theory Composition Subsystem. Then, a system analyst refines the domain model into a functional specification by removing unneeded domain model components and adding problem specific constraints. Finally, the functional specification is combined with an architecture theory to produce the complete theory-based system specification.

The presentation of the proposed Specification Acquisition System in this chapter has been informal. Subsequent chapters present a formal definition of the Domain Theory Composition Subsystem. More specifically, the remaining chapters define a mathematically sound foundation for a theory-based model of object-orientation and the translation from a graphically based object-oriented domain model to a theory-based domain model.

III. Theories and Specifications

3.1 Introduction

The software development framework presented in Chapter II is predicated on the use of formal, mathematically-based software specifications. There are two types of specifications commonly used to describe behavior in a formal specification: operational and definitional. An operational specification is basically a "recipe" for an implementation that satisfies the specification requirements while a definitional specification describes behavior by listing the properties that an implementation must have (43:5). Definitional specifications have several advantages over operational specifications: they are (1) generally shorter and clearer, (2) easier to modularize and combine together, and (3) easier to reason about. This last advantage, the ability to reason about them, is the key reason they are used in automated systems.

It is generally recognized that creating correct, understandable formal specifications is difficult, if not impossible, without the use of some structuring technique or methodology (33, 16). Algebraic theories provide the advantages of definitional specifications and the desired structuring techniques. Algebraic theories are defined in terms of a collection of values called *sorts*, a set of *operations* defined over the sorts, and a set of *axioms* defining the semantics of the sorts and operations. The structuring of algebraic theories is provided by category theory operations and provides an elegant way in which to combine smaller algebraic theories into larger, more complex theories.

Categories are an abstract mathematical construct consisting of category objects and category arrows. In general, category objects are the objects in the category of interest while category arrows define a mapping from the internal structure of one category object to another. In this research, the category objects of interest are algebraic specifications and the category arrows are specification morphisms. In this category, **Spec**, specification morphisms map the sorts and operations of one algebraic specification into the sorts and operations of a second algebraic specification such that the axioms in the first specification are theorems in the second specification. Thus, in essence, a specification morphism defines an embedding of functionality from the first algebraic specification in the second specification.

Use of algebraic specification for specification of data types was pioneered in the mid 1970s by Goguen et al., Liskov, and Zilles (16, 66). Extension of these concepts to objects and object-orientation was initially presented by Goguen and Meseguer (38) and is an increasingly common representation technique in the formalization of object-orientation (7, 14, 65, 27, 69).

3.2 Algebraic Specification

In this section, I define the important aspects of algebraic specifications and how to combine them using category theory operations to create new, more complex specifications. As described above, category theory is an abstract mathematical theory used to describe the external structure of various mathematical systems. Before showing its use in relation to algebraic specifications, I give a formal defintion (89).

Definition 3.2.1 Category. *A category C is comprised of*

1. *a collection of things called* C-objects*;*
2. *a collection of things called* C-arrows*;*
3. *operations assigning to each C-arrow f a C-object dom f (the domain of f) and a C-object cod f (the "codomain" of f). If a = dom f and b = cod f this is displayed as*

$$f : a \rightarrow b \quad \text{or} \quad a \xrightarrow{f} b$$

4. *an operation, "$\circ$", called* composition*, assigning to each pair $\langle g, f \rangle$ of C-arrows with dom g = cod f, a C-arrow $g \circ f$:domf $\rightarrow$ cod g, the* composite of f and g *such that the* Associative Law *holds: Given the configuration*

$$a \xrightarrow{f} b \xrightarrow{g} c \xrightarrow{h} d$$

of C-objects and C-arrows, then

$$h \circ (g \circ f) \quad = \quad (h \circ g) \circ f.$$

5. *an assignment to each C-object, b, a C-arrow, $id_b : b \rightarrow b$, called the* identity arrow on b, *such that the* Identity Law *holds: For any C-arrows $f : a \rightarrow b$ and $g : b \rightarrow c$*

$$id_b \circ f = f \quad and \quad g \circ id_b = g.$$

3.2.1 The Category of Signatures. In algebraic specifications, the structure of a specification is defined in terms of an abstract collection of values, called *sorts*, and operations over those sorts. This structure is called a *signature* (88). A signature describes the structure that an implementation must have to satisfy the associated specification; however, a signature does not specify the semantics of the specification. The semantics are added later via axiomatic definitions.

Definition 3.2.2 Signature. *A signature $\Sigma = \langle S, \Omega \rangle$, consists of a set S of* sorts *and a set Ω of operation symbols defined over S. Associated with each operation symbol is a sequence of sorts called its rank. For example, $f : s_1, s_2, \ldots, s_n \rightarrow s$ indicates that f is the name of an n-ary function, taking arguments of sorts $s_1, s_2, \ldots, s_n$ and producing a result of sort s. A nullary operation symbol, $c : \rightarrow s$, is called a constant of sort s.*

An example of a signature is shown in Figure 3.1. In the signature RING there is one sort, ANY, and five operations defined on the sort.

```
signature  RING   is
sorts  ANY
operations
                  plus   :  ANY × ANY   → ANY
                  times  :  ANY × ANY   → ANY
                  inv    :        ANY   → ANY
                  zero   :              → ANY
                  one    :              → ANY
end
```

Figure 3.1 Ring Signature

In my research, a signature defines the structure needed to describe object classes (attributes and operations) in a formal way. Signatures provide the ability to define the internal structure of a specification; however, they do not provide a method to reason about relationships between specifications. To create theory-based algebraic specifications that parallel object-oriented specifications, specification refinements on theories similar to those used in object-oriented approaches (inheritance, aggregation, etc.), must be available. There must be a well-defined theory about how to reason about the external structure of these specifications (i.e., how they relate to one another).

As might be expected, signatures (as the "C-objects") with the correct "C-arrows" form a category which is of great interest in this research. For signatures, the C-arrows are called *signature morphisms* (88). Signatures and their associated signature morphism form the category, **Sign**.

Definition 3.2.3 Signature Morphism. *Given two signatures $\Sigma = \langle S, \Omega \rangle$ and $\Sigma' = \langle S', \Omega' \rangle$, a signature morphism $\sigma : \Sigma \to \Sigma'$ is a pair of functions $\langle \sigma_S : S \to S',\ \sigma_\Omega : \Omega \to \Omega' \rangle$, mapping sorts to sorts and operations to operations such that the sort map is compatible with the ranks of the operations, i.e., for all operation symbols $f : s_1, s_2, \ldots, s_n \to s$ in Ω, the operation symbol $\sigma_\Omega(f) : \sigma_S(s_1), \sigma_S(s_2), \ldots, \sigma_S(s_n) \to \sigma_S(s)$ is in Ω'. The composition of two signature morphisms, obtained by composing the functions comprising the signature morphisms, is also a signature morphism. The identity signature morphism on a signature maps each sort and each operation onto itself. Signatures and signature morphisms form a category,* **Sign**, *where the signatures are the C-objects and the signature morphisms are the C-arrows.*

Given the signature RING (Figure 3.1) and RINGINT (Figure 3.2), a signature morphism σ :RING$\to$RINGINT, is shown in Figure 3.3. As required by Definition 3.2.3, σ consists of two functions, σ_S and σ_Ω as shown. σ_S maps the sort ANY to Integer while σ_Ω maps each operation to an operation with a compatible rank.

Signature morphisms map sorts and operations from one signature into another and allow the restriction of sorts as well as the restriction of the domain and range of operations. However, to build up more complex signatures, introduction of new sorts and operations into a signature is required. This is accomplished via a signature *extension* (32).

```
spec   RINGINT  is
sorts  Integer
operations
      +    : Integer × Integer    → Integer
      ×    : Integer × Integer    → Integer
      -    : Integer              → Integer
      0    :                      → Integer
      1    :                      → Integer
end
```

Figure 3.2 Integer Ring Signature

$\sigma_S = \{\text{ANY} \mapsto \text{Integer}\}$
$\sigma_\Omega = \{\text{plus} \mapsto +, \text{times} \mapsto \times, \text{inv} \mapsto \text{-}, \text{zero} \mapsto 0, \text{one} \mapsto 1\}$

Figure 3.3 Signature Morphisms: RING → RINGINT

Definition 3.2.4 Extension. *A signature* $\Sigma_2 = \langle S_2, \Omega_2 \rangle$ *extends a signature* $\Sigma_1 = \langle S_1, \Omega_1 \rangle$ *if* $S_1 \subseteq S_2$ *and* $\Omega_1 \subseteq \Omega_2$.

Signature morphisms are used to rename and refine sorts and to restrict the domain and range of operations, while extensions are used to add new sorts and operations to signatures. These operations allow the definition of entirely new signatures and the growth of complex signatures from existing signatures.

3.2.2 The Category of Specifications. The basic definitions required to develop the category of signatures and signature morphisms were presented in Section 3.2.1; however, the semantics required for software specifications have yet to be introduced. To model these semantics, the definition of a signature is extended with *axioms* which define the intended semantics of the signature operations. A signature with associated axioms is called a *specification* (88).

Definition 3.2.5 Specification. *A specification* SP *is a pair* $\langle \Sigma, \Phi \rangle$ *consisting of a signature* $\Sigma = \langle S, \Omega \rangle$ *and a collection* Φ *of* Σ*-sentences (axioms).*

Although a specification includes semantics, it does not implement a program nor does it define a particular implementation. A specification only defines the semantics required of a valid implementation. In fact, for most specifications, there are a number of implementations that satisfy the specification. Implementations that satisfy all axioms of a specification are called models of the specification (88). To formally define a model, I first define a Σ-algebra (88).

Definition 3.2.6 Σ-algebra or Σ-model. *Given a signature $\Sigma = \langle S, \Omega \rangle$, a Σ-algebra $A = \langle A_S, F_A \rangle$ consists of two families:*

1. *a collection of sets, called the carriers of the algebra, $A_S = \{A_s \mid s \in S\}$; and*
2. *a collection of total functions, $F_A = \{f_A \mid f \in \Omega\}$ such that if the rank of f is $s_1, s_2, \ldots, s_n \rightarrow s$, then f_A is a function from $A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n}$ to A_s. The symbol $\times$ indicates the Cartesian product of sets here.*

Definition 3.2.7 Model. *A* model *of a specification* SP $= \langle \Sigma, \Phi \rangle$ *is a Σ-algebra, M, such that M satisfies each Σ-sentence (axiom) in Φ. The collection of all such models M is denoted by* Mod*[SP]. The sub-category of* **Mod**(Σ) *induced by* Mod*[SP] is also denoted by* Mod*[SP].*

```
spec  RING  is
sorts  ANY
operations
      plus  :  ANY × ANY  → ANY
      times :  ANY × ANY  → ANY
      inv   :  ANY        → ANY
      zero  :             → ANY
      one   :             → ANY
axioms
      ∀a, b, c ∈ ANY
             a plus (b plus c) = (a plus b) plus c
             a plus b = b plus a
             a plus  zero  = a
             a plus (inv a) =  zero
             a times (b times c) = (a times b) times c
             a times one = a
             one times a = a
             a times (b plus c) = (a times b) plus (a times c)
             (a plus b) times c = (a times c) plus (b times c)
end
```

Figure 3.4 Ring Specification

An example of a specification is shown in Figure 3.4. This specification is the original RING signature of Figure 3.1 enhanced with the axioms that define the semantics of the operations. Valid models of this specification include the set of all integers, $\mathbf{Z}$, with addition and multiplication as well as the set of integers modulo 2, $\mathbf{Z_2} = \{0, 1\}$, with the inverse operation (-) defined to be the identity operation.

As signatures have signature morphisms, specifications have specification morphisms. Specification morphisms are signature morphisms that ensure that the axioms in the source specification are theorems (are provable from the axioms) in the target specification. Showing that the axioms of the source specification are theorems in the target specification is a proof obligation that must be shown for each specification morphism. Specifications and specification morphisms enable the creation and modification of specifications that correspond to valid signatures within the category **Sign**. Before formally defining specification morphism, I must first define a *reduct* (88).

Definition 3.2.8 Reduct. *Given a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ and a Σ'-algebra A', the σ-reduct of A', denoted $A' \mid_\sigma$, is the Σ-algebra $A = \langle A_S, F_A \rangle$ defined as follows (with $\Sigma = \langle S, \Omega \rangle$):*

$$A_S = A'_{\sigma(s)} \text{ for } s \in S, \text{ and for } f_A = (\sigma(f))_{A'}, \text{ for } f \in \Omega$$

A reduct defines a new Σ-algebra (or Σ-model) from an existing Σ'-algebra. It accomplishes this by selecting a set or function for each sort or operation in Σ based on the signature morphism from Σ to Σ'. Thus if we have a signature, Σ', and a Σ'-model, we can create a Σ-model for a second signature, Σ, by defining a signature morphism between them and taking the reduct based on that signature morphism. A reduct is now used to extend the concept of a signature morphism to form a *specification morphism* (88).

Definition 3.2.9 Specification Morphism. *A specification morphism from a specification $SP = \langle \Sigma, \Phi \rangle$ to a specification $SP' = \langle \Sigma', \Phi' \rangle$ is a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ such that for every model $M \in Mod[SP']$, $M|_\sigma \in Mod[SP]$. The specification morphism is also denoted by the same symbol, $\sigma : \Sigma \rightarrow \Sigma'$.*

I now turn to the definition of theories and theory presentations. Basically a *theory* is the set of all theorems that logically follow from a given set of axioms (89). A *theory presentation* is a specification whose axioms are sufficient to prove all the theorems in a desired theory but nothing more. Put succinctly, a theory presentation is a finite representation of a possibly infinite theory. To formally define a theory and theory presentation I must first define *logical consequence* and *closure* (89).

Definition 3.2.10 Logical Consequence. *Given a signature Σ, a Σ-sentence φ is said to be a logical consequence of the Σ-sentences $\varphi_1, \ldots, \varphi_n$, written $\varphi_1, \ldots, \varphi_n \models \varphi$, if each Σ-algebra that satisfies the sentences $\varphi_1, \ldots, \varphi_n$ also satisfies φ.*

Definition 3.2.11 Closure, Closed. *Given a signature Σ, the closure $\overline{\Phi}$ of a set of Σ-sentences Φ is the set of all Σ-sentences which are the logical consequence of Φ, i.e., $\overline{\Phi} = \{\varphi \mid \Phi \models \varphi\}$. A set of Σ-sentences Φ is said to be closed if and only if $\Phi = \overline{\Phi}$.*

Definition 3.2.12 Theory, presentation. *A theory* T *is a pair $\langle \Sigma, \overline{\Phi} \rangle$ consisting of a signature Σ and a closed set of Σ-sentences, $\overline{\Phi}$. A specification $\langle \Sigma, \Phi \rangle$ is said to be a presentation for a theory $\langle \Sigma, \overline{\Phi} \rangle$. A model of a theory is defined just as for specifications; the collection of all models of a theory* T *is denoted* Mod*[T]. Theory morphisms are defined analogous to specification morphisms.*

Specification morphisms complete the basic toolset required for defining and refining specifications. This toolset can now be extended to allow the *combination*, or composition, of existing specifications to create new specifications. Often two specifications that were originally extensions from the same ancestor need to be combined. Therefore, the desired combined specification consists of the unique parts of two specifications and some "shared part" that is common to both specifications (the part defined in the shared ancestor specification). This combining operation is called a *colimit* (89). The colimit operation creates a new specification from a set of existing specifications. This new specification has all the sorts and operations of the original set of specifications without duplicating the "shared" sorts and operators. To formally define a colimit, I must first define a cone (89).

Definition 3.2.13 Cone. *Given a diagram D in a category C and a C-object c, a cone* from the vertex c to the base D *is a collection of C-arrows* $\{f_i : c \rightarrow d_i \mid d_i \in D\}$*, one for each object* d_i *in the diagram D, such that for any arrow* $g : d_i \rightarrow d_j$ *in D, the diagram shown in Figure 3.5 commutes i.e.,* $g \circ f_i = f_j$.

$$\begin{tikzcd}
& c \arrow[dl, "f_i"'] \arrow[dr, "f_j"] & \\
d_i \arrow[rr, "g"] & & d_j
\end{tikzcd}$$

Figure 3.5 Cone Diagram

Definition 3.2.14 Colimit. *A colimit for a diagram D in a category C is a C-object c along with a cone* $\{f_i : d_i \rightarrow c \mid d_i \in D\}$ *from D to c such that for any other cone* $\{f_i' : d_i \rightarrow c' \mid d_i \in D\}$ *from D to a vertex* c'*, there is a unique C-arrow* $f : c \rightarrow c'$ *such that for every object* d_i *in D, the diagram shown in Figure 3.6 commutes; i.e.,* $f \circ f_i = f_i'$.

$$\begin{tikzcd}
& d_i \arrow[dl, "f_i"'] \arrow[dr, "f_i'"] & \\
c \arrow[rr, dashed, "f"] & & c'
\end{tikzcd}$$

Figure 3.6 Colimit Diagram

Conceptually, the colimit of a set of specifications is the "shared union" of those specifications based on the morphisms between the specifications. These morphisms define equivalence classes of sorts and operations. For example, if a morphism for specification A to specification B maps sort α to sort β, then α and β are in the same equivalence class and thus is a single sort in the colimit specification of A, B, and the morphism between them. Therefore, the colimit operation creates a new specification, the colimit specification, and a cocone morphism from each specification to the colimit specification. These cocone morphisms satisfy the condition that the translation of any sort or operation along any set of morphisms in the diagram leading to the colimit specification are equivalent (54:23). An example of the colimit operation is shown in Figures 3.7 and 3.8. Given the BIN-REL, REFLEXIVE, and TRANSITIVE specifications in Figure 3.7, the "colimit specification" would be the PRE-ORDER specification as shown in the diagram in Figure 3.8. Notice that the sorts E, X, and T belong to the same equivalence class in PRE-ORDER. Likewise, the operations $\bullet$, $=$, and $<$ also form an equivalence class in PRE-ORDER. Thus PRE-ORDER defines a specification with one sort, $\{E, X, T\}$ and one operation, $\{\bullet, =, <\}$, which is both transitive and reflexive. The specification BIN-REL defines the "shared" parts of the colimit but adds nothing to the final specification.

A category in which the colimit of all possible C-objects and C-arrows exists is called *co-complete.* As shown by Goguen and Burstall (33, 34), the category **Sign** and **Spec** are both cocomplete; therefore, the colimit operation may be used freely within the category **Spec** to define the construction of complex theories from a group of simpler theories.

Using morphisms, extensions, and colimits as a basic toolset, there are a number of ways that specifications can be constructed: (88, 37)

- Build a specification from a signature and a set of axioms;
- Form the union of a collection of specifications;
- Translate a specification via a signature morphism;
- Hide some details of a specification while preserving its models;
- Constrain the models of a specification to be minimal;
- Parameterize a specification; and
- Implement a specification using features provided by others.

Many of these methods are useful in translating object-oriented specification development into theory-based specification development. For instance, object-oriented inheritance looks very similar

```
spec  Bin-Rel  is
sorts  E
operations
     •       :  E, E  → Boolean
end
```

```
spec  Reflexive  is
sorts  X
operations
     =       :  X, X  → Boolean
axioms
     ∀x ∈ X
             x = x
end
```

```
spec  Transitive  is
sorts  T
operations
     <       :  T, T  → Boolean
axioms
     ∀x, y, z ∈ T
             (x < y ∧ y < z) ⇒ x < z
end
```

```
spec  Pre-Order  is
sorts  {E, X, T}
operations
     {•, =, <} : {E, X, T}, {E, X, T} → Boolean
axioms
     ∀x, y, z ∈ {E, X, T}
             x {•, =, <} x
             (x {•, =, <} y  ∧  y {•, =, <}z) ⇒ x {•, =, <} z
end
```

Figure 3.7   Specification Colimit Example

$$\begin{tikzcd}
 & \text{Bin-Rel} \arrow[dl, "{\{E\to X, \bullet\to =\}}"'] \arrow[dd, "c"] \arrow[dr, "{\{E\to T, \bullet\to <\}}"] & \\
\text{Reflexive} \arrow[dr, "c"'] & & \text{Transitive} \arrow[dl, "c"] \\
 & \text{Pre-Order} &
\end{tikzcd}$$

Figure 3.8   Example Colimit Diagram

to an extension to a colimit specification where the diagram of the colimit specification consists of the superclasses of the target specification. Detailed formal semantics of these object-oriented specification refinement concepts is discussed in Chapter IV.

3.3 Functors

Sections 3.2.1 and 3.2.2 defined the basic categories and construction techniques used to build large-scale software specifications. In this section, I extend these concepts further to define models of specifications and how they are related to the construction techniques used to create their specifications. Before describing this relationship, I define the concept of a *functor* which maps c-Objects and c-Arrows from one category to another in such a way that the identity and composition properties are preserved (71).

Definition 3.3.1 *Given two categories* $\mathbf{A}$ *and* $\mathbf{B}$, *a* functor $\mathcal{F} : \mathbf{A} \rightarrow \mathbf{B}$ *is a pair of functions, an* object *function and a* mapping *function. The object function assigns to each object* X *of category* $\mathbf{A}$ *an object* $\mathcal{F}(X)$ *of* $\mathbf{B}$; *the mapping function assigns to each arrow* $f : X \rightarrow Y$ *of category* $\mathbf{A}$ *an arrow* $\mathcal{F}(f) : \mathcal{F}(X) \rightarrow \mathcal{F}(Y)$ *of category* $\mathbf{B}$. *These functions satisfy the two requirements:*

$$\begin{array}{rl} \mathcal{F}(1_X) = 1_{\mathcal{F}(X)} & \textit{for each identity } 1_x \textit{ of } \mathbf{A} \\ \mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f) & \textit{for each composite } g \circ f \textit{ defined in } \mathbf{A} \end{array} \tag{3.1}$$

Basically a functor is a morphism of categories. Actually, I have already presented two functors in Section 3.2.2: the *reduct* functor that maps models of one specification (in the category $\mathbf{Mod}[X_1]$) into models of a second specification (in the category $\mathbf{Mod}[X_2]$) and the *models* functor that maps specifications in the category **Spec** to their category of models, $\mathbf{Mod}[X]$, in **Cat**, the category of all sufficiently small categories. An example of the use of the *reduct* and *models* functors is given in Chapter IV.

3.4 Summary

This chapter presented the basic mathematical structure upon which the theoretical foundations of a theory-based object model is based. The chapter started by presenting the abstract mathematical concept of a category, which is a set of c-Objects and a set of c-Arrows with specific properties. Then a signature was introduced and defined to be comprised of a set of sorts and a set of operations over those sorts. When combined with signature morphisms, signatures form the category of **Sign**. The signatures and the category **Sign** were then extended to specifications and the category **Spec** by including axioms to define the signature semantics. Σ-algebras, or models, were then introduced and defined as a set of functions and sets that implement the semantics of a given specification. Finally, functors were formally defined and two examples given: the models functor that creates models of a specification and the reduct functor that creates models of one specification from models of another specification.

The mathematical foundations laid in this chapter are used in the remaining chapters to formally define object-oriented concepts. Specifically, the next two chapters use the elements introduced in this chapter to define some basic properties of formally composed theory-based object-oriented systems as well as domain model composition techniques.

IV. Theoretical Foundations

4.1 Introduction

This chapter discusses the theoretical foundations for a theory-based model of object-orientation. To date, the informal definitions of objects, classes, and inheritance have not been agreed on much less their formal definitions. This chapter formally defines basic object-oriented concepts such as objects, classes, and inheritance using algebraic specifications in a category theory setting. Algebraic specifications capture the internal structure and semantics of the individual classes while category theory operations define the relationships between class specifications. This chapter presents a general theoretical setting in order to remain applicable to many views of object orientation.

Section 4.2 formally defines object classes as theory presentations and discusses the definition and implications of internal class consistency. Section 4.3 defines the category theory setting for class theories, models of class theories, and object instances. Finally, Section 4.4 formally defines inheritance based on a generally accepted notion of object-oriented inheritance and extends that definition to multiple inheritance.

4.2 Classes

The building block of object-orientation is the concept of an object class. A class is the blueprint from which instances of the class, called objects, are created. There are two notions of a class: a class type and a class set. A ***class type*** defines the structure of a group of similar objects as well as their response to external stimuli. A class type is generally defined by two components: attributes and operations. Attributes are observable characteristics of objects and may vary over the life of the object. Although an object's attribute values are generally visible to other objects, modification of those attribute values may only be performed by the object itself. Attribute value modification is usually performed in response to some external stimulus, usually in the form of a message or event being received by the object. Receipt of a message or event causes an operation to occur which in turn may modify the object's attribute values or cause additional messages or events to be generated. Thus a class type defines a set of operations which are used to view attribute values as well as respond to messages or events. These operations define the external interface of all objects in the class. All objects that conform to the class type definition are in the *class set.*

Given the fact that class types define a set of operations over a similar collection of objects, class type definitions may be modelled naturally as *specifications*, or *theory presentations.* Sorts are used to describe collections of data values used in the specification and include a distinguished sort, the class sort. The *class sort* is the set of all possible object *names* in the class and provides a reference to specific objects within a system. However, individual objects are not explicitly represented in specifications — specifications only define the structure and behavior of objects in the class. Objects themselves are implementation artifacts and are discussed in detail in Section 4.3.1. Attributes are defined implicitly by operations which return specific data values associated with a given object. The semantics of operations, as well as invariants between class attribute values, are defined using first order predicate logic *axioms.* In general, axioms define operations by describing their effects on attribute values or by composing other operations. I now formally define a class type.

Definition 4.2.1 Object Class Type - *A class type, C, is a signature, $\Sigma = \langle S, \Omega \rangle$ and a set of axioms, Φ, over Σ (i.e., a theory presentation, or specification) where*

S *denotes a set of sorts including the class sort*
Ω *denotes a set of operations over* S
Φ *denotes a set of axioms over* Σ

A basic assumption for a class C, is that the effect of each operation in Ω is completely defined over its domain by Φ. That is, each function, $f : A \rightarrow B$ in Ω, is required to have a provably functional relation between A and B. This requirement is not as restrictive as it initially appears. If the result of an operation does not make sense in a given object state (i.e., divide by zero, etc.), the effect of the operation can still be defined. In most cases, if an object is in an inappropriate state prior to operation invocation or the operation parameters are invalid, there is no change in the object. This behavior is axiomatized via appropriate preconditions. For example, given an *integer* object with a *divide* operation that divides the integer by a supplied parameter, the divide operation only makes sense when the parameter is non-zero. Thus axioms describing the desired behavior are:

$$\text{parameter} \neq 0 \Rightarrow \text{value(divide(integer,parameter))} = \text{integer / parameter}$$
$$\text{parameter} = 0 \Rightarrow \text{value(divide(integer,parameter))} = \text{integer}$$

The assumption that all operations are completely defined over their domain is critical in defining the effects of inheritance in Section 4.4.

4.2.1 Internal Class Consistency. It is impossible to show that a given class type is *correctly* defined without the existence of formal requirements documents. Since class type definition is based on operations and axioms defined by imperfect humans, the best that can be achieved is to show the class type definition is internally consistent; however, Church and Turing independently showed that, in general, proving that a set of first order equations is inconsistent is not possible (18:45). Therefore, in this research, I assume that user provided specifications are consistent and only attempt to show that further specification composition operations (inheritance, aggregation, etc.) maintain that consistency.

Lano and Haughton present three conditions for internal class consistency.

1. $\exists\ (c \in C) \mid INV_C(c)$
2. $\forall\ (c \in C, o \in operations(C))\ Pre_C(o) \wedge INV_C(c) \Rightarrow \exists\ (c' \in C) \mid o(c) = c' \wedge Inv_C(c')$
3. $\forall\ (c, c' \in C, o \in operations(C))\ Pre_C(o) \wedge INV_C(c) \wedge c' = o(c) \Rightarrow Inv_C(c')$ (4.1)

where $INV_C(x)$ are the invariant constraints of a class C applied to an object x, $operations(C)$ are the operations declared in the class C, and $Pre_C(o)$ are the explicit preconditions of the operation o as defined in the class C.

The first condition states that a model of the class type must exist. Specifications are not necessarily implementable. I can easily declare inconsistent axioms as shown below.

$$\begin{array}{l} x < 0 \Rightarrow x = x + 1 \\ x = x - 1 \end{array}$$

Assuming the normal semantics of $+$ and $-$ over a total order, no model exists where $x = x - 1$ and $x = x + 1$ are both true simultaneously. Therefore, by assuming internal class consistency, I also assume that models exist for all user provided specifications. This assumption is important in the discussion of Section 4.3.

The second condition requires that the preconditions defined for each operation are stronger than the preconditions necessary for the operation to work correctly while the last condition ensures that all operations preserve the class invariant (i.e., applying an operation to an object in a valid state results in an object in a valid state). Actually, this last condition is redundant since the second condition requires that given a valid object, an operation must result in a valid object. Therefore, conditions one and two provide a complete definition of internal class consistency.

For example, if the *divide* operation is defined by

$$divide(x, y) = x/y$$

internal consistency is violated. The axiom defines no precondition; however, the operation invocation $divide(1, 0)$, results in an invalid state since $1/0$ is undefined. The correct definition of *divide* requires an appropriately strong precondition. Internal consistency does allow the explicit precondition to be stronger than the actual condition. Again, if the *divide* definition was rewritten as

$$\begin{array}{l} y > 0 \Rightarrow \text{divide(x,y)} = x \;/\; y \\ y \leq 0 \Rightarrow \text{divide(x,y)} = x \end{array}$$

internal consistency is maintained even though the explicit precondition, $y > 0$, is stronger than the required precondition, $y \neq 0$.

4.3 Categorical Setting

Figure 4.1 shows the category theory setting for the definition of classes and objects. C and D represent class types defined within the category **Spec** with a specification morphism, $\sigma : C \rightarrow D$.

The *model* functor **Mod: Spec $\rightarrow$ Cat** maps each specification in **Spec** to a category of models in **Cat** (where **Cat** is the category of all sufficiently small categories). Thus given the class type D, **Mod**[D], represents the category of all models of the class type specification D. As discussed in Section 4.2.1, because I assume that specifications are internally consistent, valid models of each specification exist. An *implementation* of a class type D is defined as some model $m \in$ **Mod**[D].

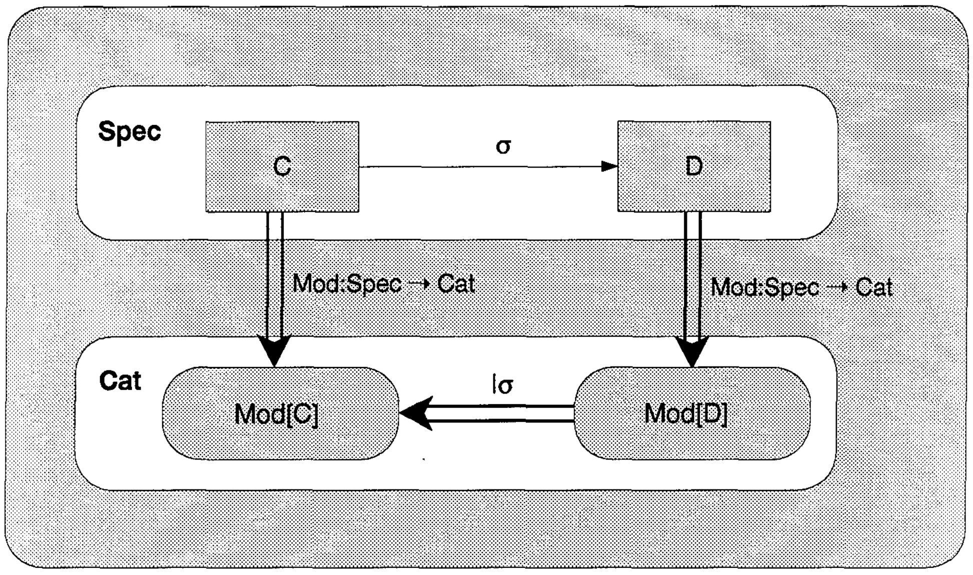

Figure 4.1 Object Reduct Framework

The specification morphism $\sigma : C \rightarrow D$ induces a *reduct* functor, denoted $D\mid_\sigma$, from **Mod**[D] to **Mod**[C] defined as

$\forall s \in S$, $C_s = D_{\sigma(s)}$ and
$\forall f \in \Omega$, $f_C = \sigma(f)_D$
where
$C = \langle S, \Omega \rangle$ with Φ,
C_s is a set from of model in **Mod**[C], and
f_C is a function from that same model in **Mod**[C].

Therefore, if class type compatibility is required between C and D, as implied by σ, compatible models of C and D may be obtained by constructing a model of C from a model of D using the reduct functor. (Here, class type compatibility means that if C and D have common sorts and operations as defined by σ, their models must have common sets and functions.) Example 4.3.1 illustrates the effects of the reduct functor on models of C and D.

Example 4.3.1 *Let C and D be class types as defined in Figure 4.2 where the notation $\mathcal{D}_{cs} < \mathcal{C}_{cs}$ denotes that the class sort of $\mathcal{D}$ is a subsort of the class sort of $\mathcal{C}$ (i.e., $D_{cs} \subseteq C_{cs}$). Then, let Cmod and Dmod represent particular models of C and D in the categories* **Mod***[C] and* **Mod***[D]. As shown in Figure 4.3, Dmod consists of four sets (S_1 for $\mathcal{C}_{cs}$, S_2 for $\mathcal{D}_{cs}$, S_3 for sort A, and S_4 for sort B) and four functions (f_1 for α, f_2 for β, f_3 for m_1, and f_4 for m_2).*

The reduct functor Dmod $|_\sigma$ defines the model Cmod from the model Dmod by selecting those sets and functions from Dmod that correspond to sorts and operations that exist in both $\mathcal{C}$ and $\mathcal{D}$ as defined by the morphism σ. Therefore, if Dmod consists of the four sets and four functions described above, Cmod consists of two sets (S_1 for $\mathcal{C}_{cs}$ and S_3 for sort A, and two functions (f_1 for α and f_3 for m_1) as shown in Figure 4.4.

class $\mathcal{C}$ is
class sort $\mathcal{C}_{cs}$
sorts A
attributes
  $\alpha : \mathcal{C}_{cs} \to$ A
operations
  $m_1 : \mathcal{C}_{cs}$, A $\to \mathcal{C}_{cs}$
axioms
  axioms omitted
end-class

class $\mathcal{D}$ is
class sort $\mathcal{D}_{cs} < \mathcal{C}_{cs}$
sorts A, B
attributes
  $\alpha : \mathcal{C}_{cs} \to$ A
  $\beta : \mathcal{D}_{cs} \to$ B
operations
  $m_1 : \mathcal{C}_{cs}$, A $\to \mathcal{C}_{cs}$
  $m_2 : \mathcal{D}_{cs}$, B $\to \mathcal{D}_{cs}$
axioms
  axioms omitted
end-class

$$\sigma = \{\mathcal{C}_{cs} \mapsto \mathcal{C}_{cs}, \mathcal{A} \mapsto \mathcal{A}, \alpha \mapsto \alpha, m_1 \mapsto m_1\}$$

Figure 4.2 Example 1 Class Type Definitions

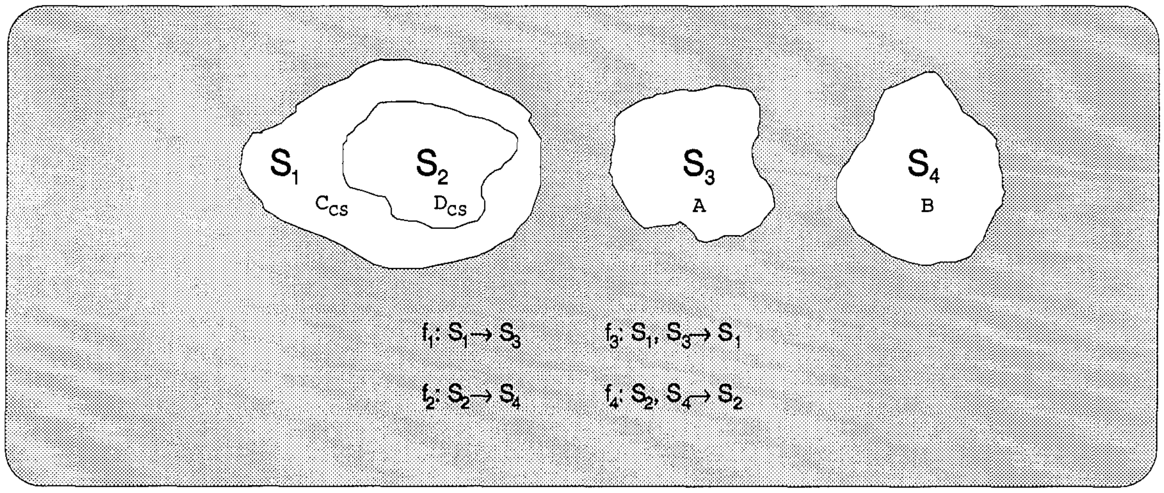
S_1
C_{CS}
S_2
D_{CS}
S_3
A
S_4
B
$f_1: S_1 \rightarrow S_3$
$f_3: S_1, S_3 \rightarrow S_1$
$f_2: S_2 \rightarrow S_4$
$f_4: S_2, S_4 \rightarrow S_2$

Figure 4.3 Dmod

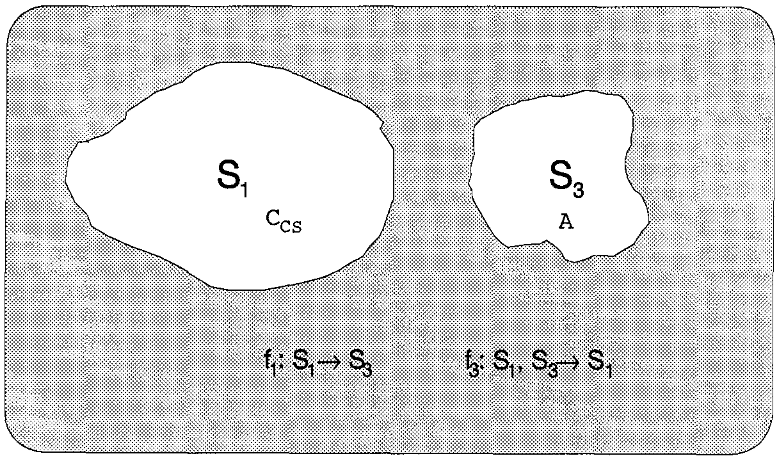

Figure 4.4 Cmod

4.3.1 Object Instances. As discussed earlier, objects are not explicitly part of the class type specification but are actually implementation artifacts. Thus objects are entities that behave according to a given implementation of a class type. In the categorical setting described above, a formal definition of an object instance can be given.

Definition 4.3.1 Object Instance - *An object, o, is a tuple, $o = \langle \eta, \mathcal{CT}, \pi \rangle$ where η is a unique name from the set in the class type model representing the class sort, $\mathcal{CT}$ is the class type model, and π is a set of variables indexed on attributes defined in the class type, $\{a_1, a_2, ...a_n\}$. An object is a member of a class, C if η is in the class type model set representing C_{cs}.*

The unique name of the object, η, is assigned at object creation and does not change over the life of the object, while π represents the current state of the object and may be modified. The class type model, $\mathcal{CT}$, defines how a given object is interpreted and generally does not change. However, as discussed in Section 4.4, because an object of a subclass is a member of the superclass as well, an object may be *reduced* to its superclass representation in which case the class type model becomes the superclass type model. When an object name is passed as a parameter to a class operation, the values upon which operations act are the values of the variables in π.

Attributes are implicitly defined in the class type through the definition of attribute viewer operations. These attribute viewers are actually *projection* functions and return a single attribute value from π. An attribute viewer, α, defined in a class type, C, is an operation from the class sort of C, C_{cs}, to a second sort, S_α, which includes all valid attribute values (i.e., $\alpha : C_{cs} \rightarrow S_\alpha$). Therefore, in an object instance of class C, a variable in π indexed on attribute α must take on values in S_α. Formally, π is defined as

$$\pi = \{a_\alpha : S_\alpha \mid \alpha \in attributes(C)\}$$

where *attributes(C)* is the set of all attributes implicitly defined by attribute viewer operations from the class type C.

Because the reduct functor creates models of one class from another, an object-reduct function may be defined to create objects in one class from objects in another. The effect of the object-reduction function mirrors the effect of the reduct functor and is defined similarly.

Definition 4.3.2 Object-Reduct Function - *Given a specification morphism, $\sigma : C \rightarrow D$, between two class types and $D_{cs} \subseteq C_{cs}$, the object-reduct function, denoted $\_ \mid_\sigma$ reduces object instances of class D to instances of class C as follows:*

$$\begin{array}{rcl} \mathbf{ob}_C.\eta & = & \mathbf{ob}_D.\eta \\ \mathbf{ob}_C.\mathcal{CT} & = & \mathbf{ob}_D \mid_\sigma \\ \mathbf{ob}_C.\pi & = & \{a_\alpha \mid a_\alpha \in \mathbf{ob}_D.\pi \wedge \alpha \in attributes(C)\} \end{array} \tag{4.2}$$

Therefore, when a specification morphism exists between two class types, objects of one type can be created from objects of the other as shown in Theorem IV.1.

Theorem IV.1 *Given a specification morphism, $\sigma : C \rightarrow D$, between two internally consistent class types such that $D_{cs} \subseteq C_{cs}$, the object-reduct function, as defined in Definition 4.3.2, exists.*

Proof: Because C and D are defined consistently the category of models for each specification (**Mod**[C] and **Mod**[D]) exists and σ induces the reduct functor $\_ \mid_\sigma$: **Mod**[D]→**Mod**[C].

1. Since $D_{cs} \subseteq C_{cs}$, the object name of each object in D exists in C and thus $\mathbf{ob}_C.\eta = \mathbf{ob}_D.\eta$.
2. $\mathbf{ob}_C.\mathcal{CT}$ is defined by the reduct functor $\_ \mid_\sigma$.
3. Since all attributes defined in C are mapped to attributes in D by σ, there exists a corresponding variable in $\mathbf{ob}_D.\pi$ for each attribute variable in $\mathbf{ob}_C.\pi$.

□

Example 4.3.2 illustrates the desired effect of the object-reduct function on object instances.

Example 4.3.2 *Given the class type definitions of C and D and models Cmod and Dmod as defined in Example 4.3.1, the objects* $\mathbf{ob}_C$ *and* $\mathbf{ob}_D$ *can be defined over Cmod and Dmod. If* $\mathbf{ob}_D$ *is the tuple $\langle \eta, Dmod, \{a_1, a_2\}\rangle$, where a_1 is a value in set S_3 and a_2 is a value in set S_4, then* $\mathbf{ob}_C = \langle \eta, Cmod, \{a_1\}\rangle$ *where a_1 is a value in set S_3 and $Cmod = Dmod \mid_\sigma$.*

Because functions and sets are copied from models of one class to create models of another, the functions must provide the same behavior on objects of both classes. This behavioral equivalence is shown by the commutative diagram in Figure 4.5. Theorem IV.2 states that this diagram does in fact commute.

$$\begin{tikzcd}
d \arrow[r, "\_|_\sigma"] \arrow[d, "\sigma(f)_D"'] & d|_\sigma \arrow[d, "f"] \\
\sigma(f)_D(d) \arrow[r, "\_|_\sigma"] & f(d|_\sigma)
\end{tikzcd}$$

Figure 4.5 Behavioral Equivalence of Objects

Theorem IV.2 *If $\sigma : C \rightarrow D$ is a specification morphism between two internally consistent specifications and f_C is a function in the model of C created from the function $\sigma(f)_D$ in the model of D via the reduct functor induced by σ such that $f_C = \sigma(f)_D$, then for all objects, $d \in D$, $f_C(d\,|_\sigma) = \sigma(f)_D(d)\,|_\sigma$.*

Proof: Assume without loss of generality that the attributes $a_1 \ldots a_n \in attributes(D\,|_\sigma)$ and that $a_1 \ldots a_q \in attributes(D)$ such that $attributes(D\,|_\sigma) \subseteq attributes(D)$. Also, assume that if $d\,|_\sigma = \langle \eta, D, a_1 \ldots a_q \rangle$ then $\sigma(f)_D(d) = \langle \eta, D, b_1 \ldots b_q \rangle$. Note: If $d = \langle \eta, D, a_1 \ldots a_q \rangle$ then $d = \langle \eta, D\,|_\sigma, a_1 \ldots a_n \rangle$.

Then, if $d = \langle \eta, D, a_1 \ldots a_q \rangle$,

$$\begin{aligned}
\sigma(f)_D(d)\,|_\sigma &= \sigma(f)_D(\langle \eta, D, a_1 ... a_q \rangle)\,|_\sigma \\
&= \langle \eta, D, b_1 ... b_q \rangle))\,|_\sigma \\
&= \langle \eta, D\,|_\sigma, b_1 ... b_n \rangle)) \\
&= f_c(\langle \eta, D\,|_\sigma, a_1 ... a_n \rangle) \\
&= f_c(d\,|_\sigma)
\end{aligned}$$

□

4.4 Inheritance

Class inheritance plays an important role in object-orientation; however, the correct use of inheritance is not uniformly agreed upon. Many languages provide "ad-hoc" inheritance that allows a subclass to redefine or even remove attributes or operations inherited from its superclass. However, most authors see the necessity to restrict the amount of modification freedom in a subclass. In this research, I require a *generalization-specialization* inheritance relationship. There are two types of inheritance that satisfy the generalization-specialization relationship: extension and restriction. In an extension, a subclass simply adds new attributes or operations, whereas in a restriction a subclass constrains attribute values from a superclass. To allow a subclass to be freely substituted for its superclass in any situation and to make reasoning about the class's properties easier, I require that a subclass only *extend* the features of its superclass. Liskov defines these desired effects as the "substitution property" (67):

> If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

Bourdeau and Cheng (14) interpret the substitution property to mean that an object of class S may be substituted for an object of class T at any point in time. This effectively means that the object of class S may be in any valid state prior to its substitution for a class T object. This interpretation requires that a subclass object must always be in a state that directly maps to a state defined in its superclass and only allows the addition of substates and concurrent states within the subclass. I find Bourdeau and Cheng's interpretation too restrictive and interpret the substitution property to mean that a subclass object, when created and stimulated only within an environment created for its superclass, behaves as a superclass object. My interpretation allows a subclass object to respond to new messages or events that take it into new states that do not exist in its superclass; however, when starting in the initial states as defined in the superclass and only responding to messages or events inherited from the superclass, the subclass object must behave exactly as an object from its superclass and may not enter a state defined only in the subclass.

To hold to this notion of generalization–specialization there are two requirements: (1) the substitution property holds and (2) class consistency is maintained. The only way to ensure the substitution property holds in all cases is to ensure that the effects of all superclass operations performed on an object are equivalent in the subclass and the superclass. Before showing how the substitution property and class consistency are preserved, object-equivalence must be defined.

Definition 4.4.1 Object-Equivalence - *Two objects, c1 and c2, of a class type C are equivalent over C if and only if the value of all attributes defined in C are equal in c1 and c2, or,*

$$c1 \equiv_C c2 \Leftrightarrow \forall\ (a)\ (a \in attributes(C) \Rightarrow a(c1) = a(c2))$$

Now I can present a formal definition of the substitution property. In this definition, o' is the operation in class D inherited from class C.

$$\forall\ (d)\ d \in D \wedge o \in operations(C) \Rightarrow \exists\ (c)\ c \in C \wedge (c \equiv_C d \Rightarrow o(c) \equiv_C o'(d)) \qquad (4.3)$$

An acceptable definition of inheritance would then provide a mapping from the sorts, operations, and attributes in the superclass to those in the subclass that preserve the semantics of the superclass. This is the basic definition of a specification morphism and provides us a formal definition of inheritance.

Definition 4.4.2 Inheritance - *A class D is said to inherit from a class C, denoted $D < C$, if there exists a specification morphism from C to D and the class sort of D is a subsort of the class sort of C (i.e., $D_{cs} \subseteq C_{cs}$).*

This definition states that all sorts and operations from class C are embedded in class D, that a new sort, the class sort of D, is defined as a subsort of the class sort of C, and that the axioms in C are theorems in D. While Definition 4.4.2 provides a concise, mathematically precise definition of inheritance, its ability to ensure the preservation of class consistency and the substitution property as stated in Equations 4.1 and 4.3 must be shown.

Before proving that Definition 4.4.2 preserves the substitution property, two lemmas must be proved. The first lemma shows that an object and its reduct are object-equivalent and the second states that given a specification morphism between two classes, an object in the subclass is also an object in the superclass.

Lemma 4.4.1 *Given a specification morphism,* $\sigma : C \rightarrow D$, $\forall\ (d)\ \ d \in D \wedge c = d\mid_\sigma \Rightarrow c \equiv_C d$.

Proof: By Definition 4.3.2, $d\mid_\sigma .\pi = \{a_\alpha \mid a_\alpha \in d.\pi \wedge \alpha \in attributes(C)\}$ thus it is obvious that $\forall a \in attributes(C)$ if $c = d\mid_\sigma$ then $a(c) = a(d\mid_\sigma)$ and by Definition 4.4.1, $c \equiv_C d$. □

Lemma 4.4.2 *If there exists a specification morphism between two classes* $\sigma : C \rightarrow D$ *and* $D_{cs} \subseteq C_{cs}$ *then for every object* $d \in D$ *there exists some* $c \in C$ *such that* $d \equiv_C c$.

Proof: By the definition of a specification morphism, $\sigma : C \rightarrow D$, a reduct functor, $\_\mid_\sigma$, creates models of C from models of D. The object-reduct function as defined in Definition 4.3.2 takes objects defined over D and creates objects defined over C with identical attribute values for all attributes in C. Therefore, $\forall\ (d)\ d \in D \Rightarrow \exists\ (c)\ c \in C \wedge c = d\mid_\sigma$ and by Lemma 4.4.1, $c \equiv_C d$. □

Now it is possible to show that inheritance, as defined in Definition 4.4.2, does in fact preserve the substitution property and thus is a valid definition for this model.

Theorem IV.3 *Given a specification morphism,* $\sigma : C \rightarrow D$, *between two internally consistent classes* C *and* D, $D_{cs} \subseteq C_{cs}$, *and that the model of* C, *Cmod is created via the reduct functor induced by* σ *from the model of* D, *Dmod, then substitution property holds.*

Proof:

1. By Lemma 4.4.2, for all objects in D there exists an object in C such that $c = d\mid_\sigma$.
2. If *Cmod* is constructed from *Dmod* using the reduct functor, then for all objects $o \in operations(C)$ there exists some $f_C \in Cmod$ such that $\sigma(f)_D \in Dmod$ and $f_C = \sigma(f)_D$.

3. Then, by Theorem IV.2 $\forall\ (d)\ \exists\ (c)$ such that $d \in D \wedge c \in C$ then $c = d\mid_\sigma\ \Rightarrow f(c) = f(d\mid_\sigma) = f(d)\mid_\sigma$, or $f(c) \equiv_C \sigma(f)_D(d)$.

□

4.4.1 Multiple Inheritance. Multiple inheritance requires a slight modification to the notion of inheritance as defined in Definition 4.4.2. The set of superclasses must first be combined and then used to "inherit from".

Definition 4.4.3 Multiple Inheritance - *A class D multiply inherits from a collection of classes $\{C_1\ ..\ C_n\}$ if there exists a specification morphism from the colimit of $\{C_1\ ..\ C_n\}$ to D such that the class sort of D is a subsort of each of the class sorts of $\{C_1\ ..\ C_n\}$.*

The colimit operation allows the combination of any number of classes, along shared parts, to create a single specification with all the sorts, operations, and axioms of the original classes. This colimit specification can then be extend with the definition of the new class sort, attributes, and operations.

This definition ensures that the subclass D inherits (in the sense of Definition 4.4.2) from each superclass in $\{C_1\ ..\ C_n\}$. The proof is shown in Theorem IV.4 below.

Theorem IV.4 *Given a specification morphism from the colimit of $\{C_1\ ..\ C_n\}$ to D such that the class sort of D is a subsort of each of the class sorts of $\{C_1\ ..\ C_n\}$, the substitution property holds between D and each of its superclasses $\{C_1\ ..\ C_n\}$.*

Proof: In the category **Spec**, all cocone morphisms from any $C_j \in \{C_1\ ..\ C_n\}$ to the colimit specification composed with the extension from the colimit specification to D is a specification morphism from C_j to D as shown in Figure 4.6. Thus multiple inheritance implies a specification morphism between each C_j and D and thus by Theorem IV.3, the substitution property holds. □

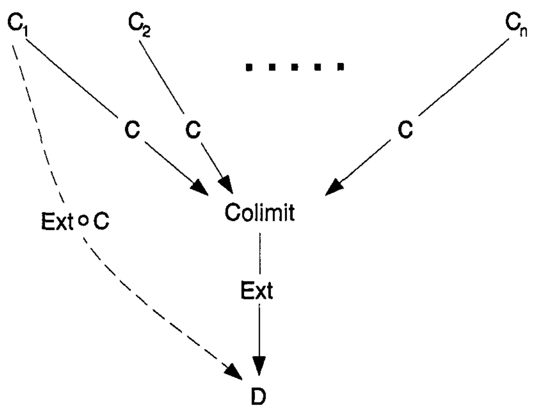

Figure 4.6 Multiple Inheritance Colimit

It is important to note that Definition 4.4.3 only ensures valid inheritance when all operations are fully defined in each specification $\{C_1 \mathinner{..} C_n\}$. Failure to ensure fully defined operations may result in an inconsistent colimit specification.

Consider the example shown in Figure 4.7. A class E is created by multiply inheriting from D' and D'' which both are subclasses of an superclass C where the operation m is not fully defined. While everything works syntactically, the resulting class is inconsistent due to the axioms defined in D' and D''.

$$x(e) \geq 0 \Rightarrow x(m(e)) = x(e) + 1$$
$$x(e) \geq 0 \Rightarrow x(m(e)) = x(e) + 2$$

Both D' and D'' are valid subclasses of C yet they modified the behavior of a m in such a way that the resulting multiply inherited operation definition is inconsistent. Thus class consistency conditions do not hold.

4.5 Summary

This section establishes the mathematical foundation for a formal theory-based model of object-oriented concepts for the system defined in Chapter II and shown in Figure 2.2. These

```
class C is
import Integer
class sort C
attributes
  x : C → Integer
operations
  m : C → C
axioms
  ∀ (c: C)  x(c) < 0 ⇒ x(m(c)) = x(c) - 1
end-class
```

```
class D′ is
class sort D′ < C
axioms
  ∀ (d: D)  x(d) ≥ 0 ⇒ x(m(d)) = x(d) + 1
end-class
```

```
class D″ is
class sort D″ < C
axioms
  ∀ (d: D)  x(d) ≥ 0 ⇒ x(m(d)) = x(d) + 2
end-class
```

```
class E is
import Integer
class sort E < D′, D″
attributes
  x : E → Integer
operations
  m : E → E
axioms
  ∀ (e: E) x(e) < 0 ⇒ x(m(e)) = x(e) - 1;
  ∀ (e: E) x(e) ≥ 0 ⇒ x(m(e)) = x(e) + 1;
  ∀ (e: E) x(e) ≥ 0 ⇒ x(m(e)) = x(e) + 2
end-class
```

Figure 4.7 Inconsistent Multiple Inheritance

foundations are general in nature and are applicable to a number methodologies within the object-oriented paradigm.

First, classes were defined as theory presentations or specifications within the category **Spec** while the models of each class, defined by the **Mod** functor, form a category within the category **Cat**. Models of a class were associated with an *implementation* of the class and a mathematically sound method for creating compatible models of a class was defined. This method is based on the existence of a specification morphism between classes and thus the existence of a *reduct functor* that creates models of one specification from models of another. The theoretical concept of an object instance was then introduced along with an object "reduct" function based on the reduct functor. These concepts were used to show the desired effect of inheritance.

Finally, a formal definition of inheritance was presented based on the "substitution property". This formal definition of inheritance was then shown to preserve the substitution property in Theorem IV.3. These results were then extended to multiple inheritance. The next two chapters build on the mathematical foundations presented in this chapter by defining a theory-based object model and its semantics that incorporates the concepts of classes, objects, and inheritance defined in this chapter.

V. Formal Object Modeling Technique Semantics

5.1 Introduction

This chapter present a formal semantics for a restricted version of Rumbaugh's OMT. These restrictions are imposed on each of the three OMT models–object, dynamic, and functional– to ensure automatic translation from graphical notation to algebraic specifications. The semantics of the restricted OMT models are arrived at by defining a formal semantics for each of Rumbaugh's models. Section 5.2 describes the basic requirements placed on a graphically based diagram in order to ensure automated translation. Section 5.3 describes the restrictions and formal semantics for the object model while Sections 5.4 and 5.5 describe the restrictions and formal semantics for the dynamic and functional models.

5.2 Translation Requirements

Translation of graphically based models into algebraic specifications in the category **Spec** requires that the two criteria be met. First, all entities in the model must correspond to components of an algebraic specification or be defined using category theory constructs. This means that all important features of the model must be representable as specifications, functions, sorts, first-order axioms defined over those sorts and functions, or category theory operations between specifications. For instance, in a data flow diagram, a process translates to a function while data flows between processes are defined by sorts.

The second requirement is that all models be deterministic, that is, there must be a single valid interpretation of the model. Again, in a data flow diagram, the usual interpretation is that the processes and data flows within the diagram only define *possible* data flow paths, and does not imply any specific sequence of control. While the flow of data through the system may be obvious to a user based on the types of data involved and certain naming conventions, the normal interpretation of data flow diagrams does not provide the degree of determinism necessary to use them in an automated system without modifying the semantics or adding additional information. Therefore, the translation requirements for any graphically based model are given below.

I.	All model entities must be represented by pure functions, sorts, first-order axioms, or category theory constructs.
II.	Model semantics must be deterministic.

5.3 Object Model

This section discusses the semantics defined for the OMT object model and the problems encountered in translating it into algebraic specifications. Section 5.3.1 presents a brief overview of the OMT object model, Section 5.3.2 describes proposed restrictions to the OMT object model, and Section 5.3.3 derives the semantics of the restricted object model from the formal OMT object model semantics defined by Bourdeau and Cheng (14).

5.3.1 Overview of Rumbaugh's Object Model. The OMT object model defines the structure of a domain based on classes of objects and the relationships between them. A class defines the structure of a similar set of objects. This structure is defined by *attributes*, which are data that describe various aspects of an object, and *operations* that describe how an object behaves. Relationships between objects fall into one of three categories: *associations*, *aggregation*, and *inheritance.* Associations are general relationships between two or more classes. Aggregation describes a "part-of" relationship between an object and a subobject which is used to make up the aggregate object. Inheritance describes a "generalization-specialization" relationship between two classes of objects where if A is a subclass of B then objects of class A are objects of class B as well.

Figure 5.1 shows an example of a typical OMT object model domain, a rocket. The OMT diagram shows six object classes: Rocket, Airframe, FuelTank, PropulsionDevice, JetEngine, and RocketEngine. The lines drawn between object classes represent relationships between them. The $\Diamond$ on the line from the Rocket to the Airframe, FuelTank, and PropulsionDevice denotes an aggre-

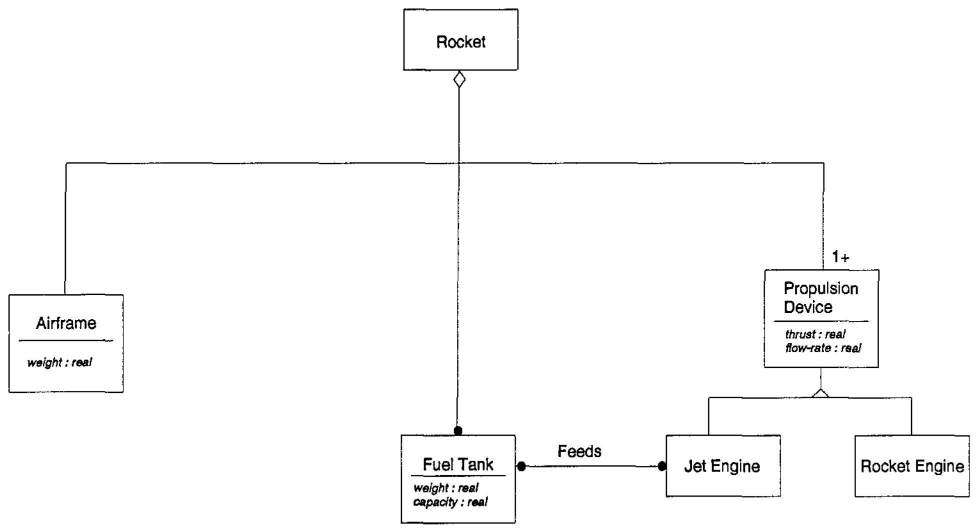

Figure 5.1 Typical Object Model

gation relationship. That is, a Rocket consists of an Airframe, FuelTank, and a PropulsionDevice. The $\triangle$ symbol between the PropulsionDevice class and the JetEngine and RocketEngine classes denotes inheritance. Thus an object that is a member of the JetEngine or RocketEngine class is also a member of the PropulsionDevice class. Finally, the line between the FuelTank and the JetEngine denotes an association that relates members of the FuelTank class to members of the JetEngine class. The text below the class name in object class rectangles defines attributes. Thus a FuelTank object has two attributes, weight and capacity, each of which is a *real* datatype. The endpoints of an association/aggregation line denotes a particular multiplicity in the association and aggregation relationships as show in Figure 5.2. These multiplicities define the number of relationships in which a particular object may participate. In Figure 5.2, each object at the other end of the relationship may be in a relationship with exactly one, zero or more, zero or one, one or more, or a numerically specified number of objects of the type *Class.*

5.3.2 The Restricted OMT Object Model. The only problem inherent in Rumbaugh's definition of the object model, as described in Section 5.3.1, is the ability of the user to define operations. This capability allows users to define operations that have no relation to the dynamic

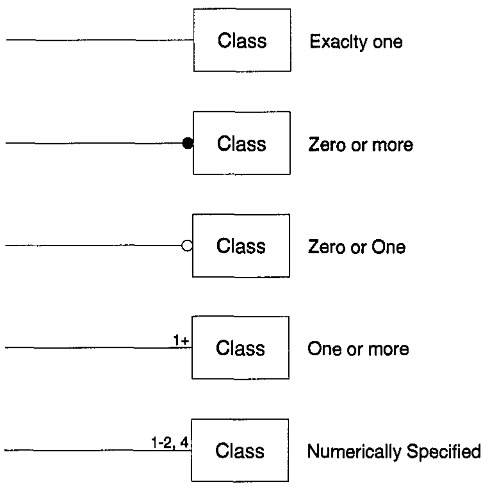

Figure 5.2 Relationship Multiplicities

or functional model and, therefore, has the potential to introduce inconsistencies. In this research, I take the approach that all operations are defined as either a process in the functional model or an action in the dynamic model. Therefore, the only restriction to the OMT object model is to disallow the introduction of operations.

Assumption V.1 *All class methods and operations are introduced in the OMT dynamic or functional model.*

5.3.3 Object Model Semantics. Relatively little exists describing the formal semantics of object models. Some generic data base work (31) describes structural aspects of object-oriented systems but they fail to provide a complete semantics for an OMT-type object diagram. Bourdeau and Cheng (14) have developed a formal semantics for an extended version of the OMT object model notation. They use the Larch specification language (43) to describe modular algebraic specifications based on the OMT object model. In this section I formally define the semantics of the OMT object model based on the semantics proposed by Bourdeau and Cheng. (Note: the following rules are a subset of those defined by Bourdeau and Cheng in (14). Many of their rules deal with OMT object model extensions, such as state, which are part of the basic OMT dynamic or functional models and are omitted here.)

The object model itself is a specification which simply includes the object classes and associations derived from the object diagram. Classes and associations are defined as follows.

Definition 5.3.1 *A* **Restricted OMT Object Model**, *OM, is a set of specifications determined by Definitions 5.3.2 through 5.3.5.*

Definition 5.3.2 *An* **Object Class** *C in an object model O defines a specification where*

(OM-1) *The class C is denoted by a sort of the same name.*

(OM-2) *For each attribute, α, in class C of type D, there is a function signature*

$$\alpha : C \rightarrow D$$

(OM-3) *If D in rule (OM-2) references a separate class specification, then the specification for D is included (imported) into the specification of class C.*

The rules above define the structural aspect of object classes only. Relational aspects such as association, aggregation, and inheritance are defined below. The next three definitions define additional rules which, when used on aggregate classes, associations, and subclasses fully defines the object model.

Definition 5.3.3 *An* **Aggregate**, *with components $D_1 ... D_k$ defines a specification D using rules OM-1 – OM-3 where*

(OM-4) *For each component $D_1 ... D_k$, the aggregate relation is denoted by a predicate* has-part *which relates class D to D_n. If the component has a role name in the aggregation, the role name is used in place of* has-part.

$$\textit{has-part} : D, D_n \rightarrow \textit{Boolean}$$

(OM-5) *Axioms defining the multiplicity constraints of the aggregation are added to D.*

Thus rules OM-4 and OM-5 define predicates in the aggregate class definition that relate a given aggregate object to its components. Bourdeau and Cheng also provide a detailed methodology for defining multiplicity axioms; however, in this research I use equivalent axioms defined in Section 6.4.1. The next relation defined by Bourdeau and Cheng is the subtype relation which implements inheritance.

Definition 5.3.4 *Let D be a* **subtype** *of class C. Then a specification is defined for both C and D using rules OM-1 – OM-5 and where*

(OM-6) *The subtype relation is defined by an operation named* simulates *in D relating class D to C.*

$$simulates : D \rightarrow C$$

In essence, the *simulates* operation takes an object of the subtype and produces an object of the supertype, thus implementing Bourdeau and Cheng's interpretation of the *substitution property.* As discussed in Section 4.3, I use a slightly different interpretation of the substitution property than do Bourdeau and Cheng. Basically, they assume that a subtype object D must *always* be substitutable for its supertype object C, while I interpret the property to mean that if, from object creation, an object from class D is restricted only to operations defined on class C then the object is indistinguishable from an object of class C. Bourdeau and Cheng define the following constraint that the *simulates* operation must satisfy:

$$\forall\ (d : D)\ a \in attributes(d) \Rightarrow a(d) = a(simulates(d)) \tag{5.1}$$

Bourdeau and Cheng's constraint restricts subtype objects further than my interpretation, which formalized is:

$$\begin{aligned} &\forall\ (d : D)\ a \in attributes(D) \wedge o \in operations(D) \wedge a(d) = a(simulates(d)) \\ &\quad \Rightarrow\ a(o'(simulates(d))) = a(simulates(o(d))) \end{aligned} \tag{5.2}$$

where o' is the inherited operation in D derived from operation o in C. Equation 5.2 incorporates the intent of Equation 5.1 given my interpretation of the *substitution property.*

Definition 5.3.5 *An* **Association,** *R, relating objects from classes $D_1 ... D_k$ defines a specification A using rules OM-1 – OM-3 and where*

(OM-7) *The association R is denoted by a predicate of the same name in specification A:*

$$R : D_1 ... D_k \rightarrow Boolean$$

(OM-8) *Axioms defining the multiplicity constraints of R are added to A.*

In effect, rules (OM-7) and (OM-8) state that for an association, a new specification is created with a boolean predicate of the same name which defines which objects of the two associated classes are related.

5.4 Dynamic Model

This section discusses the semantics of the OMT dynamic model and the problems encountered in translating it into algebraic specifications. Section 5.4.1 presents a brief overview of the dynamic model, Section 5.4.2 discusses the problems in translating the dynamic model into algebraic specifications, Section 5.4.3 defines restrictions to the dynamic model to counter those problems, and Section 5.4.4 derives the semantics of the restricted dynamic model from the formal semantics of Harel's statecharts (45).

5.4.1 Overview of Rumbaugh's Dynamic Model. The concept of state is vital to object-orientation. State, as defined by Rumbaugh, is an abstraction of an object's attribute values and is represented in his model via a statechart. A typical statechart is shown in Figure 5.3. Statecharts have five parts: states, transitions, events, guards, and actions. The *state* of a system "summarizes the information concerning past input and that is needed to determine the behavior of the system on subsequent inputs" (47:13). Therefore, a system typically resides in a state between inputs and may change state based on additional input. *Transitions* are the changes of state based on given input and the current state. Transition labels define the input event, guard, and actions associated with a transition as shown below.

$$event \; [\; guard\text{-}condition \;] \; / \; action_1 ... action_n$$

Events are the instantaneous transmittal of information from one object to another. In statecharts, communication is assumed to occur as global broadcasts. If an object generates an event, it is received by all other objects who have a transition labeled with the same event name. Besides the receipt of the event itself, additional information may be transmitted via event parameterization.

It is assumed that if there is no explicit transition from a state S upon receipt of an event, E, then if event E is received while in state S, no actions occur. *Guard Conditions* allow a system to check that, upon receipt of an event, certain conditions hold before a transition takes place. *Actions* are the behavior initiated by some transition. Actions include computation of data, modification of object attributes, or broadcasting of additional events. Although Harel and Rumbaugh allow statecharts to place actions *inside* states (the equivalent of a Moore machine from finite automata) as well as on transitions, for convenience, in this research I limit all actions to transitions (a Mealy machine). This does not represent a semantic restriction as the equivalence of Mealy and Moore machines is well known (47:43). A special type of action is a *send* action. A send action specifies that an event is to be broadcast to the system. A typical send action might be:

send(*event-name*(*parameter list*))

In this case the *event-name* is broadcast to all receiving objects along with all data passed via the parameter list. Other actions become operations on the current object.

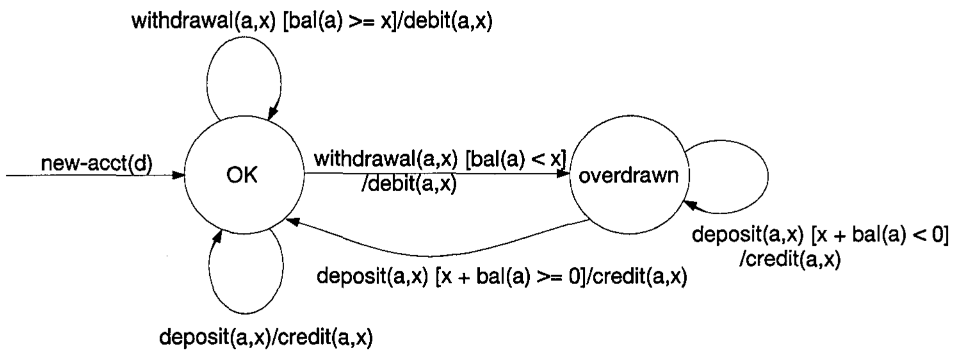
withdrawal(a,x) [bal(a) >= x]/debit(a,x)
new-acct(d)
OK
withdrawal(a,x) [bal(a) < x] /debit(a,x)
overdrawn
deposit(a,x) [x + bal(a) < 0] /credit(a,x)
deposit(a,x) [x + bal(a) >= 0]/credit(a,x)
deposit(a,x)/credit(a,x)

Figure 5.3 Account Dynamic Model

Statecharts often allow for concurrent states or substates. Concurrent state diagrams are used when the attributes of a class type may be partitioned into subsets. The state of an object becomes a tuple consisting of the object state in each concurrent state diagram. An example of a concurrent state diagram is shown in Figure 5.4. In this example, events $e1$, $e2$, and $e3$ cause the first component of the object state to change while events $e4$, $e5$, and $e6$ cause the second component of the object state to change. The events in the concurrent diagrams do not have to be distinct while the effects of those events on object attributes are. For example, assume the object state of an object whose dynamic model is represented by Figure 5.4 is $\langle 1,4\rangle$. If the following string of events, ($e1$, $e4$, $e5$, $e2$), is received, the object ends up in state $\langle 3,6\rangle$ by the following transitions.

$$\langle 1,4\rangle \xrightarrow{e1} \langle 2,4\rangle \xrightarrow{e4} \langle 2,5\rangle \xrightarrow{e5} \langle 2,6\rangle \xrightarrow{e2} \langle 3,6\rangle$$

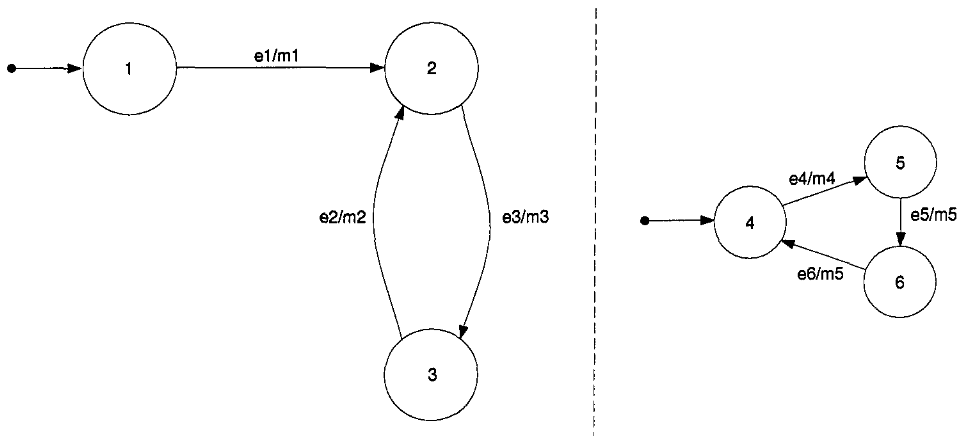

Figure 5.4 Concurrent State Diagram

In a non-concurrent statechart, an object must be in a single state; however, within a state, substates may exist that refine the state allowing the object to change its substate while remaining in the superstate as shown in Figure 5.5. Such substates "inherit" the transitions of their superstate. In the theory-based object model, substates are handled similarly to concurrent states – by adding additional substate attributes whose values are only meaningful when the object is in the appropriate superstate. Thus, the object in Figure 5.5 must always be in a superstate 1, 2, or 3; however, while in state 2, the events $e4$, $e5$, and $e6$ allow the object to change it substate and remain in state 2. Once an $e2$ event is received, the superstate transitions to state 3 and the substate attribute value is no longer meaningful. The object will not change state even if an $e4$, $e5$, or $e6$ event is received.

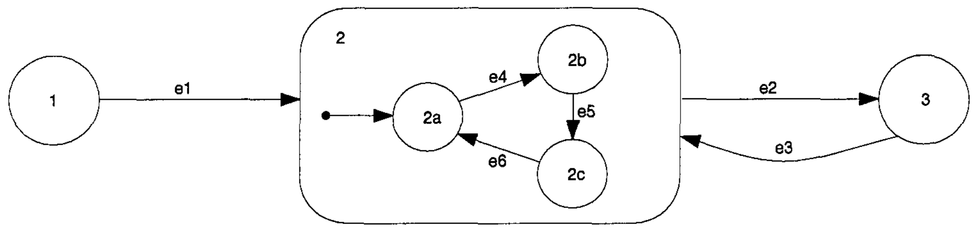

Figure 5.5 SubState Diagram

5.4.2 Dynamic Model Translation Problems. This section discusses problems inherent in translating the dynamic model, as specified by Rumbaugh, into algebraic theories and describes the incompatibility of the dynamic model with the translation requirements described in Section 5.2.

There is little problem associated with translating individual statecharts into sorts, functions, and first-order axioms in algebraic specifications. Each statechart, substatechart, and component of concurrent statecharts defines an attribute function. The states within each diagram define a sort where the values in the sort are the states in the diagram. Each state also defines a nullary function for each individual state or substate. Events and actions translate into functions. Axioms are used to define the effects of transitions on state attributes and actions. Additional axioms may also be generated to show that receipt of events with no defined transitions from a given state result in no changes. Actions are modelled as either functions representing operations on the object or as events to be broadcast to the system. Use of the categorical constructs of morphisms and colimits allows specification of event transmission.

Even though translation seems straightforward, some inconsistencies may occur when inheritance is taken into account. For instance, substates may be used to refine a superclass state as long as the superclass events defining transitions from the superstate are not used in the substate diagram. Attempting to override an exit transition from the superstate results in inconsistencies as shown in Figure 5.6. Rumbaugh interprets this statechart to mean that when an object is in state $2b$ and event $e2$ occurs, the substate transition overrides the superstate transition and the state moves to 2c; however, this violates the substitution property since the subclass object no longer behaves as a superclass object.

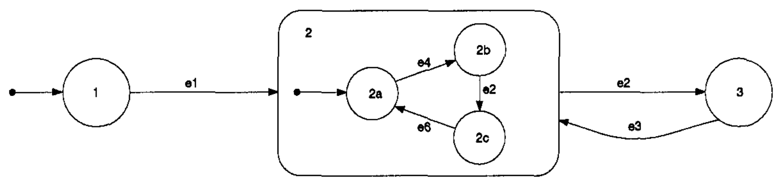

Figure 5.6 Invalid Substate Diagram

Figure 5.7 shows a valid concurrent state diagram. This diagram is valid since the states and events of the two concurrent subdiagrams are distinct. However, not all concurrent dynamic models satisfy this consistency condition. Concurrent state diagrams are only valid when the attributes of the class are partitionable and the actions in a concurrent subdiagram affect only the attributes in a single partition. When this condition is violated, inconsistencies between diagrams may result. Therefore, when a concurrent statechart is introduced in a subclass to refine the dynamic model, the actions associated with transitions in the concurrent component must affect only the subclass attributes. This requires that the actions in the concurrent component be operations defined in the subclass and that those actions modify only subclass attributes.

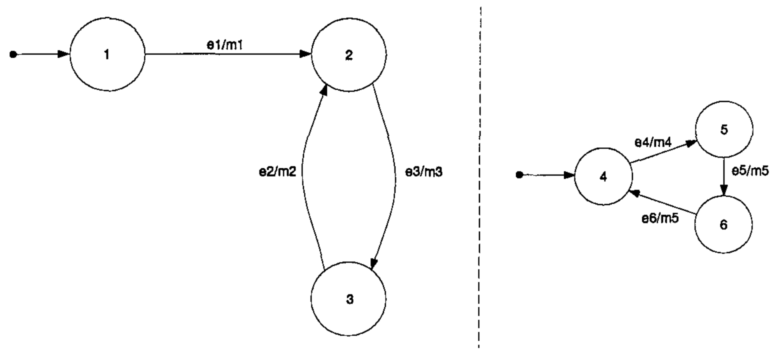

Figure 5.7 Inheritance of Dynamic Behavior - Concurrent Diagram

5.4.3 The Restricted OMT Dynamic Model. With the exception of substate transition overriding, there are no inconsistencies with statecharts as used by Rumbaugh; however, for simplicity, I limit the dynamic model to a Mealy machine representation where all actions must occur on transitions as discussed in Section 5.4.1. I also assume that if the user specifies guard conditions for a set of transitions on event e from state s, then the guards for that set of transitions are *consistent* and *complete.* This may require the user to define transitions that "do nothing"; however, this assumption ensures that a complete and consistent set of axioms can be generated from the dynamic model without resorting to considerable reasoning about the validity of the guard conditions. I also assume that all guard conditions are written in O-SLANG syntax and are based only on the parameters passed to the object by the event and the object's attribute values. The user may specify multiple actions on a transition; however, since non-send actions represent operation invocations and there is no sequence implied by the order of actions, multiple non-send actions occur in parallel. If two actions have a specific sequence, the user can easily create an operation that implements this sequential behavior; therefore, in general, only one non-send action is specified per transition. My restrictions to Rumbaugh's dynamic model are shown below.

1. **Assumption V.2** *All actions must occur on transitions between states.*
2. **Assumption V.3** *All guard conditions must be written in valid* O-SLANG *syntax using only event parameters, object attributes, and constants.*
3. **Assumption V.4** *Guard conditions for a set of transitions for a single event are consistent and complete.*
4. **Assumption V.5** *Only one non-send action may be specified per transition.*
5. **Assumption V.6** *Events on transitions leading from a superstate may not be used in substates (i.e., no overriding superstate transitions).*
6. **Assumption V.7** *Concurrent statecharts must partition the attributes affected by methods in different concurrent sections into disjoint sets.*

5.4.4 Dynamic Model Semantics. In this section I formally define the semantics of the restricted dynamic model. As expected, the semantics of dynamic model statecharts are based the standard automata definition (47:43).

Definition 5.4.1 *A* **Mealy machine** *is a six-tuple* $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ *where*

- Q *is a finite set of states.*
- Σ *is a input alphabet.*
- Δ *is the output alphabet.*
- δ *is the transition function mapping* $Q \times \Sigma \rightarrow Q$.
- λ *is a mapping from* $Q \rightarrow \Delta$ *giving the output for each transition.*
- $q_0 \in Q$*, is the initial state.*

The mapping from the Mealy machine to a statechart is straightforward and given in the definition of a statechart below.

Definition 5.4.2 *A dynamic model* **Statechart** *is a six-tuple* $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ *where*

- Q *is the set of states in the statechart.*
- Σ *is the set of input events.*
- Δ *is the set of output events and actions.*
- δ *is the transition function mapping* $Q \times \Sigma \rightarrow Q$ *based on transition arrows and guard conditions.*
- λ *is a mapping from* $Q \rightarrow \Delta$ *giving the output for each transition based on transition arrows and guard conditions.*
- $q_0 \in Q$*, is the initial state.*

Substates and concurrent states are a simple translation into this ordinary automata (45) due to Rumbaugh's simplification of statecharts and the additional restrictions of Section 5.4.3. Substates are "unfolded" into their superstate by transferring incoming superstate transitions to the initial state of the substate statechart and adding outgoing superstate transitions to each substate as shown in Figure 5.8.

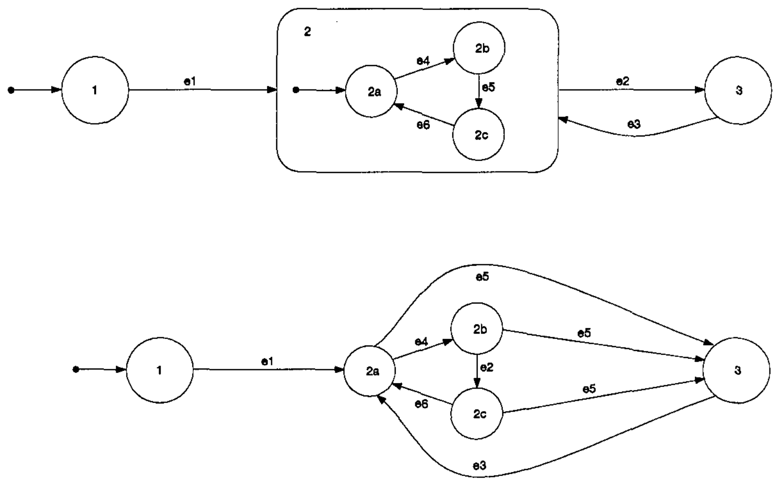

Figure 5.8 Unfolding Substates into a Single Statechart

Concurrent state diagrams, such as shown in Figure 5.4, are translated by creating a state for each combination of concurrent state values and adding the appropriate transitions. The translation of Figure 5.4 is shown in Figure 5.9. In this example, the events and operations of the two concurrent subdiagrams are disjoint. Although operations must be disjoint, events may be shared. If events $e1$ and $e4$ in Figure 5.4 are equivalent (i.e., they have the same name), then the translation to a single-level, non-concurrent diagram is shown in Figure 5.10. In this example, state $(1,5)$ is not included since event $e1$ now transitions the state directly from $(1,4)$ to $(2,5)$. The composition of operations $m1$ and $m4$ is not problematic since in the restricted dynamic model they are required to modify a disjoint sets of attributes.

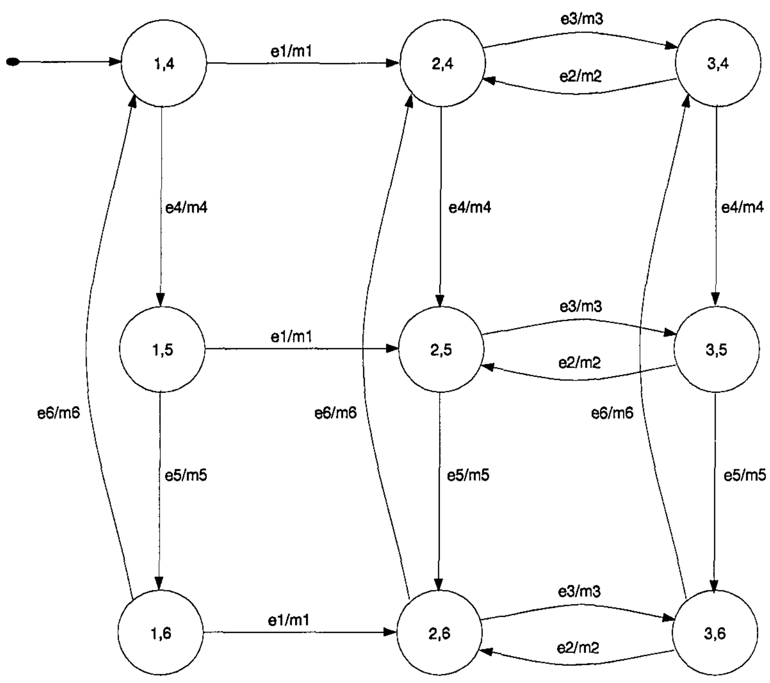

Figure 5.9 Composition of Single Statechart from Concurrent Diagram

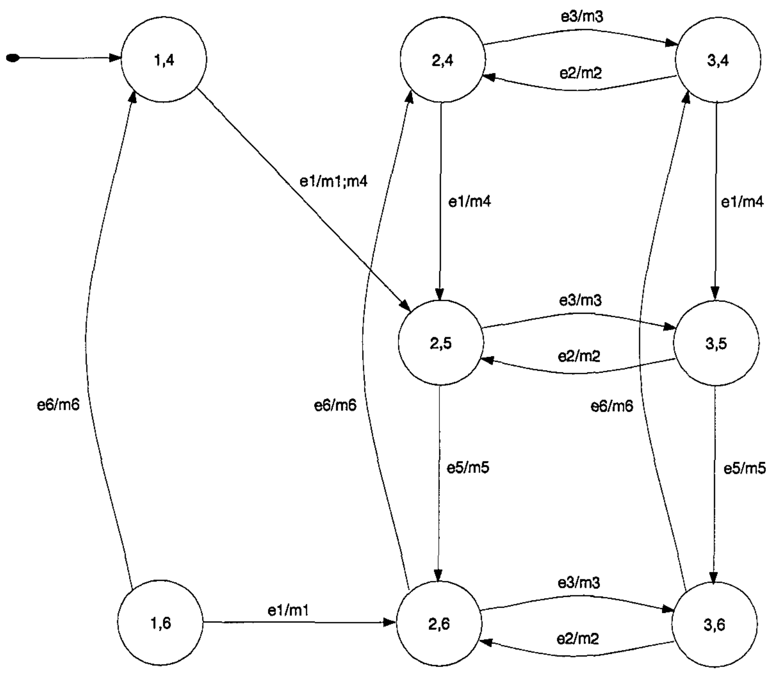

Figure 5.10 Composition of Single Statechart from Concurrent Diagram

5.5 Functional Model

This section discusses the OMT functional model and the problems encountered in attempting to translate it into algebraic specifications. Section 5.5.1 presents an overview of the OMT functional model, Section 5.5.2 discusses the problems encountered in trying to translate the OMT functional model, Section 5.5.3 describes my proposed restrictions to functional model, and Section 5.5.4 derives the semantics of the restricted functional model from the semantics of generalized data flow diagrams.

5.5.1 Overview of Rumbaugh's Functional Model. The OMT functional model is defined using standard data flow diagrams describing the computations that a system must perform. The functional model is intended to be used in conjunction with the object and dynamic models to complete the system definition. Whereas the object model defines system components and the dynamic model defines system control, the functional model defines what computations occur in the system. The functional model describes what outputs are derived from inputs, but not how, or in what order, this transformation is accomplished. The "how" is an implementation question while the "order" of the computations is defined in the dynamic model.

As stated above, the functional model uses data flow diagrams to describe what computations occur in the system and the relationship of inputs to outputs. Data flow diagrams consist of four basic entities: processes, data flows, data stores, and actors. Processes transform input data values into output data values and are represented by ovals in the data flow diagram. Rumbaugh states that "the lowest-level processes are pure functions without side effects" (83:124) while higher-level processes may have side effects such as modifying data stores or other external objects. In general, high-level processes represent non-side affecting operations or actions defined in the dynamic model. Each process may be decomposed into subprocesses which take the inputs to its higher-level process and provide a more detailed description of the transformations necessary to produce the high-level outputs. These "nested" data flow diagrams require that all higher-level process input data flows be present in the nested data flow diagram and that the outputs of the higher-level process be computed by processes in the nested data flow diagram. Eventually, all nested data flow diagrams terminate with purely functional processes.

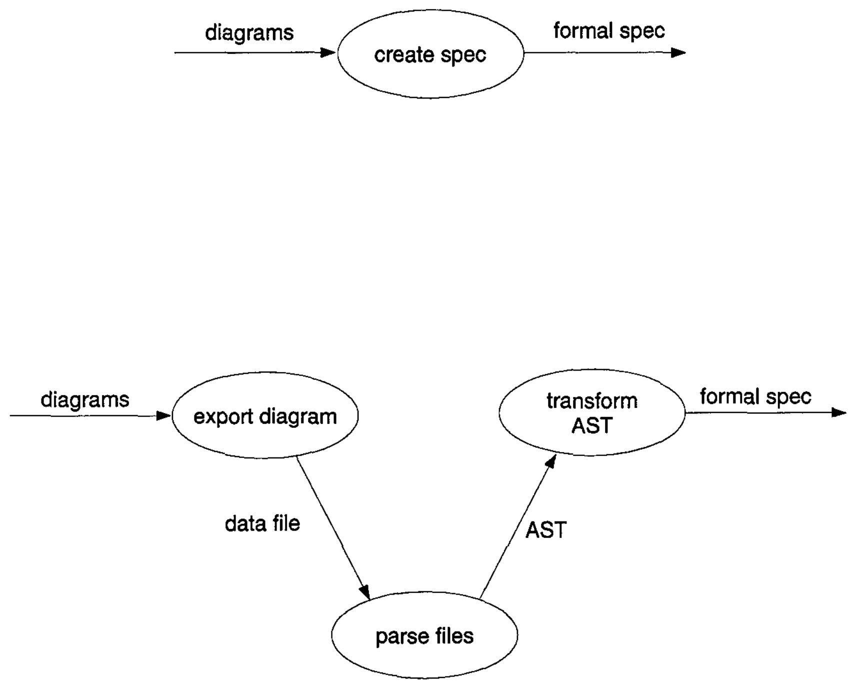

Figure 5.11 High-Level and Nested Data Flow Diagrams

In the functional model, data flows are denoted by lines drawn between processes and represent data values input to, or output from a process. Arrows indicate the direction the data flows. A data flow from one process to another specifies that the data is output by the first process and input to the second process. Often, data flows are "forked" to denote copying the data value and sending it to more than one process as shown in Figure 5.12. When a data flow represents an aggregate data type, the data flow may "split" into its various components. Likewise, aggregate components may be composed into an aggregate value by merging two or more data flows into a single data flow.

Data stores are passive objects in the system used to store data and are represented by parallel lines. Data stores are usually objects, object classes, or associations defined in the object model.

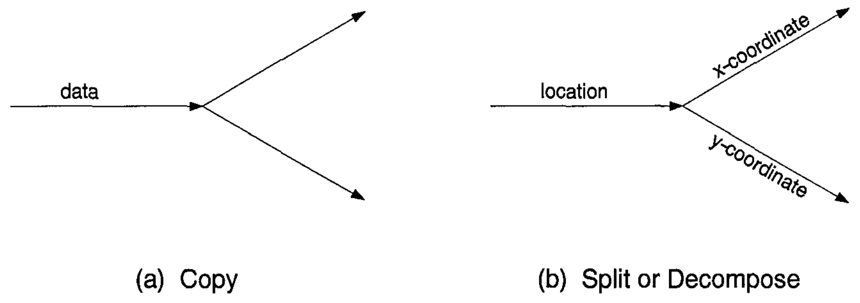

Figure 5.12 Data Flows

Data flows into a data store represent modifications to the stored data while data flows out of a data store represent data retrieved from the data store. Data flows with hollow-tipped arrows represent the dynamic creation of a new object. Such arrows flow from a process to a data store and are unique to the OMT functional model. Figure 5.13 shows the use of data stores in a data flow diagram. In Figure 5.13 the *Bank-Accounts* data store is the set, or class, of accounts in a bank. The *get-account* process chooses an account based on an account number entered by a user. The selected account then becomes a data store and is manipulated by the *deposit* process.

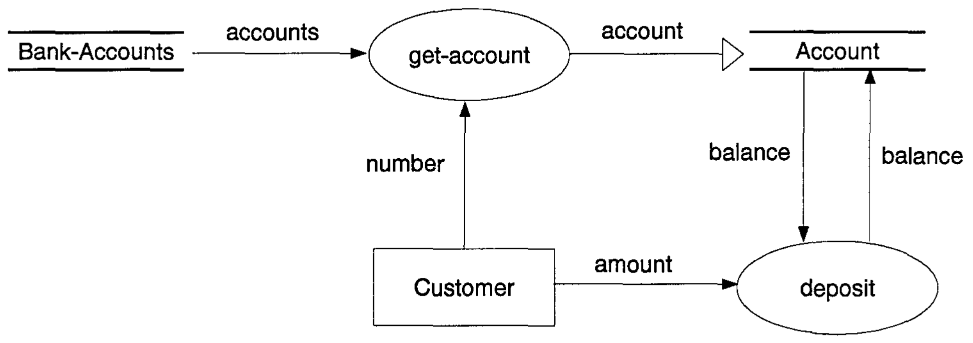

Figure 5.13 Data Stores

Actors are objects outside the system that provide input to or consume the output of data flow diagrams. Actors are not actually part of the system and are not modeled further in OMT. Actors are represented by rectangles and their inclusion in the functional model simply provides the context for the computations defined within.

Rumbaugh also allows for *control flows* within the functional model. Control flows are boolean values that affect whether a process is performed and are not input values to the process. Control flows are denoted by dashed lines between processes. According to Rumbaugh, control flows "can occasionally be useful, but they duplicate information in the dynamic model and should be used sparingly" (83:129).

5.5.2 Functional Model Translation Problems. This section discusses problems inherent in translating the functional model, as specified by Rumbaugh, into algebraic specifications. Some of these problems are caused by translation requirements discussed in Section 5.2 while some are problems inherent to data flow diagrams in general.

The first problem is the representation of actors. Since actors are only used to set the context of the functional model and do not affect computations or data flows involved in the model, actor objects can be ignored as long as all system inputs and outputs are accounted for. Therefore, in the ensuing discussion of the functional model, I assume that actors are unimportant to translating the functional model and may be excluded.

The second problem is Rumbaugh's use of the functional model. While he states that high-level processes should equate to operations in the object model or actions in the dynamic model, he tends to a take a system-level view when developing his functional model and, as a result, his processes are unrelated to the object or dynamic models. This system-level use forces the specifier to backtrack and attempt to determine exactly where these new processes should reside. However, if I stick to his original statement that processes represent operations and actions and use the functional model only to decompose non-side affecting operations (i.e., queries) and previously defined actions, this problem disappears and the three models become integrated. While many of the operations and actions of basic object classes are simple enough to write axiomatic definitions, operations and actions of aggregate objects tend to require more thought and decomposition. Thus, in this research, I assume that only non-side affecting operations or previously defined actions are decomposed using functional models and that these are generally used at the aggregate level.

Next, problems associated with translation requirement *I*, that all entities in the model must be represented by pure functions, sorts, and first order axioms, are addressed. Initially, it seems that processes may simply be represented by functions, data flows by sorts and variables in first order axioms, and data stores by specific values in the sorts. However, the first problem with this simplistic approach is the requirement to use pure functions to represent processes. While Rumbaugh states that the lowest-level processes are pure functions, higher-level processes are allowed to cause side affects (i.e., modify other objects, classes of objects, or associations) within the system. Since algebraic specifications require pure functions at all levels, the question is how to represent side effects in pure functions. Functions in algebraic specifications may emulate side affects by requiring the object of interest to be input to and output from the function. If I assume processes only affect the objects, object classes, or associations within the aggregate for which they are being designed, all processes may be defined as pure functions. Because process side effects may only affect the current aggregate object, if the aggregate object is passed as an input parameter to a function and returned as an output parameter, then the process is a pure function.

The second problem related to translation requirement *I* involves the use of data stores. Since data stores represent objects, object classes, or associations within the aggregate object, passing the aggregate object as an input parameter to each process allows each processes access to any object within the aggregate. The process may modify or send events to aggregate components as specified and return their modified values upon completion. Actually, if a particular process only accesses or modifies a single component within the aggregate, only that particular component need be passed as a parameter. In effect, the process in question becomes an operation on the component it accesses or modifies.

The final, and most difficult, set of problems is related to translation requirement II, that all functional models be completely deterministic. Unfortunately, Rumbaugh himself states that the functional model does not uniquely specify results of side-effecting processes and shows only possible data paths. Basically what Rumbaugh is saying is that just because certain processes and data paths are shown in the functional model, not all processes or data flows are necessarily used to compute the high-level process outputs. For instance, Figure 5.14 shows a functional model for a process *update data*. The data flow diagram at the bottom depicts the decomposition. The data *modifications* is input to process A which computes two outputs x and y. These outputs become inputs to processes B and C respectively. Process B retrieves a data value d from the *Data store* and updates *Data store* with data d as well. Process C, on the other hand, simply updates *Data store* with a data value d. As Rumbaugh states, this diagram only shows us *possible* data flows and processes used to implement *update data*. Do B and C both execute and update different aspects of *Data store* and does the order of their execution matter? Or, does either B or C execute (based on a decision made in A) and update *Data store*? Obviously, the diagram does not provide enough information to decide, and therefore cannot be deterministically translated into a set of functions and axioms that implement *update data*.

Since all processes are required to be pure functions, process A must output both x and y everytime it is called, thus requiring some control feature to determine when B or C is executed. One solution is to require the sorts for both x and y to provide a "distinguished" value, such as *undefined*, for A to output indicating which process should execute. An alternative solution is to require A to output explicit control flow information. Both of these choices are particularly uninspiring. Impregnating sorts with distinguished values to control execution is unacceptable as it inhibits reuse and is theoretically impure . This example only requires the definition of a single distinguished value; however, in more complex control situations, additional distinguished values might be added to the point where reuse of specifications in different contexts might become unmanageable due to conflicting sort values. Use of control flows is also unacceptable since control

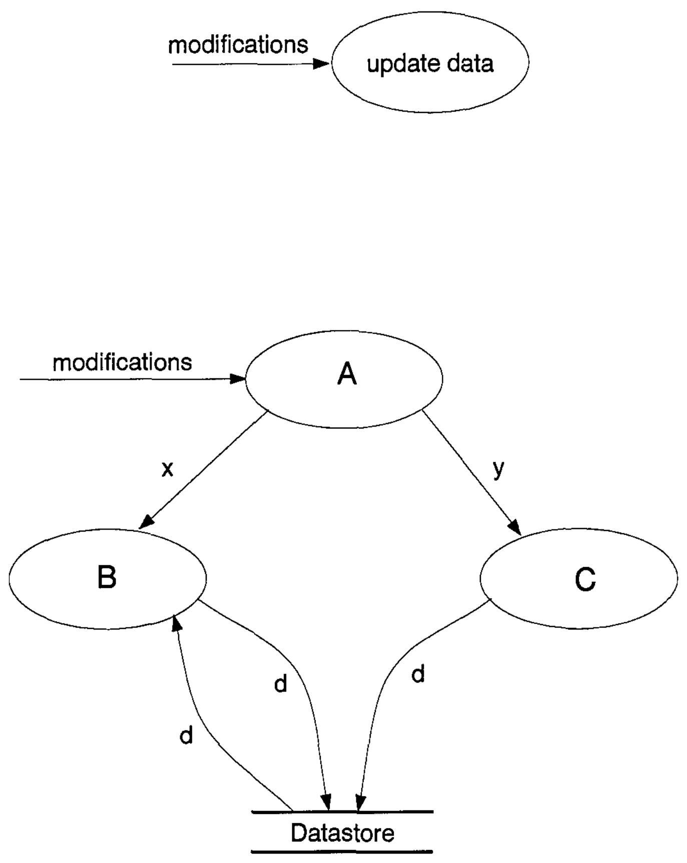

Figure 5.14 A Nondeterministic Functional Model

flow is defined in the dynamic model. Use of additional control flows in the functional model leads to confusion and almost certainly inconsistent specifications. A better approach is to place all control flow information in the dynamic model using additional states and transitions as shown in Figure 5.15, or to define these decisions using axiomatic process definition. Either solution removes such conditional execution and thus produces a deterministic functional model.

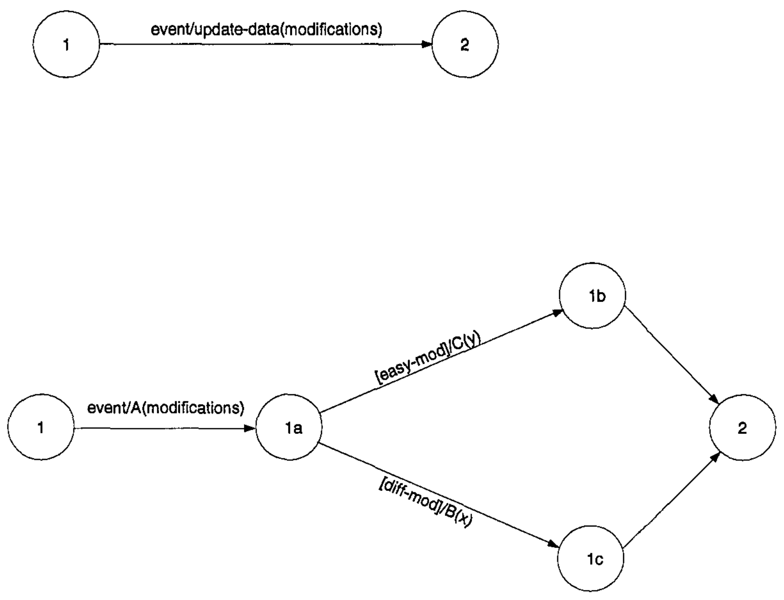

Figure 5.15 Placing Control in Dynamic Model

The final problem with the functional model is a problem with data flow diagrams in general: How are sequencing, iteration, and conditional execution represented? In Rumbaugh's functional model this is not a problem since he allows the functional model to represent "possible paths". However, as pointed out above, to translate the functional model into algebraic specifications, it must be deterministic. Therefore, the sequencing/iteration/conditional execution problem is the same problem discussed above and requires the same solution: specify all control in the dynamic model or directly using axioms. Sequencing, iteration, and conditional execution can all be handled by either method.

5.5.3 The Restricted OMT Functional Model. To deterministically translate the functional model into algebraic specifications, certain restrictions were suggested in Section 5.5.2. These suggestions are embodied in the following rules and ensure translation requirements I and II are met.

1. **Assumption V.8** *Only non-side affecting operations (i.e., queries) or actions (side affecting operations) defined in the dynamic model are decomposed using functional models.*
2. **Assumption V.9** *Processes only affect the objects, object classes, or associations within the class (aggregate) for which it is being designed.*
3. **Assumption V.10** *Data stores must be class sets or associations while data flows are individual objects and links. If a data store is a class set, the name of the data store is assumed to be "c-CLASS" or a role name (where "c" is the name of the class). If the data store is an association, the name of the data store is assumed to be "c-ASSOC".*
4. **Assumption V.11** *All processes are pure functions. All data stores accessed must be input as parameters to the process.*
5. **Assumption V.12** *All processes must be either 1) a method, or 2) a purely functional operation:*
   - *(a) All processes implementing dynamic model actions are assumed to be side affecting and take an object (and possibly other parameters) as input and return the modified object as output.*
   - *(b) All processes/subprocesses that access or modify data stores must input the appropriate class set or association.*
   - *(c) All processes/subprocesses that modify data stores may output only a single data store and thus become "leaf" process in the functional model.*

6. **Assumption V.13** *All retrievals from data stores are performed by a "leaf" process that accesses only a single data store. This retrieval occurs before modifications to data stores may occur.*

7. **Assumption V.14** *By convention, data flows to or from a data store are label with <class-name : class-set-sort> or <association-name : association-sort>.*

8. **Assumption V.15** *There is no explicit ordering of updates to the same data store. All sequencing requirements must be specified in the dynamic model only.*

9. **Assumption V.16** *All subprocesses execute and produce all outputs whenever the higher-level process is executed.*

10. **Assumption V.17** *Conditions, loops, and sequencing requirements are specified in the dynamic model only. Sequencing in the functional model is derived solely from data dependencies.*

11. **Assumption V.18** *All uniquely named data flows into or out of a process are individual inputs or outputs of the process and completely define the input and output parameters of the process. Data flows from a single process with identical names and types are considered a single output and may be used as inputs to multiple processes or data stores.*

12. **Assumption V.19** *Data flows are represented as* name : type *and are assumed to be unique. If two identically named data flows are* outputs *from the same process, they are assumed to be a single output.*

13. **Assumption V.20** *Composition and decomposition of aggregate values is performed by user-defined processes. Forking and joining of data flows is not supported.*

14. **Assumption V.21** *Copying of data may be performed by a specific process or by outputting multiple copies of a data flow from a single process. Copying by splitting data flows is not supported.*

5.5.4 Functional Model Semantics. In this section I formally define the semantics of the restricted OMT functional model. A few formal definitions of data flow diagram semantics have been proposed. Adler (2) and Tao (92) propose graph-based semantics with a relation defining the precedence, or "is used to compute", relationship that exists between data flows. Vazquez (94) defines semantics of data flow diagrams by sentences over a Σ-term algebra, where Σ is an algebraic signature defining the basic constructs of data flow diagrams including some very simplistic control structures. Since control is not included in the restricted functional model, I choose to model the semantics of data flow diagrams using the approach of Tao and Kung. Tao and Kung formally define data flow diagrams as a directed graph with a precedence relation over the data flows as defined below.

Definition 5.5.1 *A* **Data Flow Diagram** *is a quadruple $D = (C, F, K, R)$ where*

- *$C = P \cup S \cup E$ is a nonempty finite set of components consisting of pairwise disjoint sets: P, the set of processes; S the set of data stores; and E the set of external entities.*
- *$F \subseteq (P \times P) \cup (P \times S) \cup (S \times P) \cup (P \times E) \cup (E \times P)$ is the set of data flows; each with a unique name.*
- *$K \subseteq P \cup S$ is the set of subsystem components, and $C - K$ the environment.*
- *$R \subseteq F \times F$ is a precedence relation between elements of F.*

This definition defines a data flow diagram as a directed graph with a set of components (processes and data stores) within the environment (the system) being modelled as well as outside the environment (actors) to set the diagram "context". Each data flow is assumed to have a unique name and transfer data directly from one component to another (i.e., no splitting or forking). Because general data flow diagrams do not require all paths to be executed, certain data flow inputs to a process may not be used to compute all outputs of the process. Thus Tao and Kung define a *precedence relation* to describe exactly which inputs are required to produce which outputs as defined below.

Definition 5.5.2 *The* **Precedence Relation** *of a data flow diagram, D, denoted R_D, is the transitive closure of the union of the precedence relations for all components $c \in K$, or $R_D = (\bigcup_{c \in K} R_c)^+$ where R_c denotes the precedence relation for the component c and $^+$ is the transitive closure. R_c is defined as:*

- *If $c \in P$ then R_c is the empty set $\{\}$ if $P \in C - K$; otherwise, P is a mapping from $I(c)$ to $O(c)$ such that $(d_i, d_j) \in R_c$ if and only if d_i is used by P to produce d_j.*
- *If $c \in S$ then R_c is the empty set $\{\}$ if $S \in C - K$; otherwise, $d_i \in I(c)$ and $d_j \in O(c)$, $(d_i, d_j) \in R_c$ if and only if d_i and d_j contain some data item in common.*
- *If $c \in E$ then R_c is the empty set $\{\}$.*

where $I(c)$ and $O(c)$ are the input to and outputs from component C.

The precedence relation defines precisely which data flows must be defined prior to the computation of other data flows. The relation is defined over the components in the environment. Just because a data flow is an input to a process does not mean it is used to compute a particular output of the same process. That information lies solely in the internal semantics of the process. For example, assume we have a process, *ProduceReports* that produces two reports, a summary report and an error report, as shown in Figure 5.16. In this case, the summary report is generated as long as there are valid or invalid accounts input to the process whereas the error report is only generated when there are invalid accounts input to the process. In this case, $R_{ProduceReports} = \{(invalid\text{-}accounts, error\text{-}report)\}$, since neither valid or invalid accounts are required to produce a summary report; however, an invalid account is required to produce an error report.

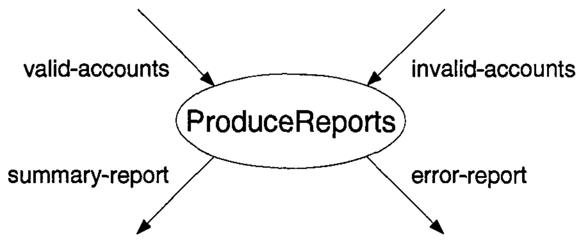

Figure 5.16 Non-Deterministic Precedence

Therefore, to define the precedence relation, the analyst must know the functionality of each process. That is, he or she must know which output data is dependent upon which input data. Although the precedence relation captures the data flow relationships, it does not provide control information. For instance, in Figure 5.14, it is still not known whether process B or C executes, or in what order the input and output to the data store occur.

Given the restrictions placed on general data flow diagrams in the restricted OMT functional model in Section 5.5.3, the semantics of data flow diagrams as defined by Tao and Kung can be refined to define the semantics of the restricted OMT functional model.

Definition 5.5.3 *A* **Restricted OMT Functional Model** *is a quadruple* $D = (C, F, K, R)$ *where*

- $C = P \cup S \cup E$ *is a nonempty finite set of components consisting of pairwise disjoint sets:* P*, the set of processes;* S *the set of data stores; and* $E = \{Extern\}$ *where* $Extern$ *represents any entity external to the object being defined.*
- $F \subseteq (P \times P) \cup (P \times S) \cup (S \times P) \cup (P \times E) \cup (E \times P)$ *is the set of data flows; each with a unique name.*
- $K = P \cup S$ *is the set of subsystem components.*
- $R \subseteq F \times F$ *is a precedence relation between elements of* F*.*

Because the functional model restrictions require all inputs to precede all outputs, and for all data store reads to precede data store writes, the precedence relation becomes computable directly from C and F and requires no analyst intervention.

Definition 5.5.4 *The* **Precedence Relation** *for a restricted OMT functional model,* D*, denoted* R_D*, is the transitive closure union of the precedence relations for all components* $c \in K$*, or* $R_D = (\bigcup_{c \in K} R_c)^+$ *where* R_c *denotes the precedence relation for the component* c *and* $^+$ *is the transitive closure.* R_c *is defined as:*

- *If* $c \in P$ *then* $R_c = \{(d_i, d_j) \mid d_i \in I(c) \wedge d_j \in O(c)\}$.
- *If* $c \in S$ *then* $R_c = \{(d_i, d_j) \mid d_i \in I(c) \wedge d_j \in O(c)\}$.
- $R_{Extern} = \{\}$.

Thus, for the restricted functional model, the precedence relation simply defines which data flows must be computed before others based on their graph location. With these semantics, structural aspects of the restricted functional model are completely defined. For a subdiagram that defines a higher-level process, this defines the semantics of the higher-level process in terms of lower-level processes. The internal functionality of these low-level processes are defined axiomatically.

5.6 Summary

This report describes the OMT object model as used by Rumbaugh as well as problems encountered when attempting translate it into algebraic specifications. Restrictions and exact interpretations of the object model, dynamic model, and functional model entities are presented which allow the object model to be deterministically translated into algebraic specifications. The formal semantics of the restricted object model are represented by Bourdeau and Cheng's algebraic specification, the semantics of the restricted dynamic model are derived from the semantics of statecharts as defined by Harel, while the semantics of the restricted functional model are derived from generalized data flow diagram semantics.

The semantics defined in this chapter are used in Chapter VI to help define the theory-based model. Their main use, however, is in Chapter VII where they are used to show that the translations defined in that chapter preserve the semantics of the individual models.

VI. A Theory-Based Object Model

6.1 Introduction

This chapter defines a theory-based object model based on Rumbaugh's semi-formal Object Modeling Technique (OMT) (83). The theory-based object model is described using algebraic specifications to define object classes while relationships between classes are defined via category theory operations within the category **Spec**. The algebraic theory language used to capture the internal class structure as well as the relationships between classes is O-SLANG, an object-oriented derivative of Slang (54). A complete description of O-SLANG is contained in Appendix B.

This theory-based object model is designed to faithfully capture the essence of an object-oriented specification in a formal framework and to provide the capability to reason about the resulting specification. This framework is based on a formal definition of generally accepted object-oriented concepts. Section 6.2 presents the basic concepts of object classes including attributes, methods, events, operations, and states and how they are captured in an algebraic specification. The next three sections discuss relationships between objects using algebraic class specifications and category theory concepts: Section 6.3 defines the mechanism and effects of inheritance, Section 6.4 presents the concept of links and associations, and Section 6.5 discusses a unique specification, the aggregate class. The final section, Section 6.6, explains how *event theories* and category theory operation are used to create communication paths in a domain model based on events defined in the dynamic model.

There are four basic premises upon which the theory-based object model is based.

1. The model is designed to capture information necessary for domain modeling. Domain modeling is concerned with capturing general object classes and operations within a domain. System specifications are derived from domain models by selecting the specific number and types of object classes as well as providing detailed system level requirements as discussed in Chapter II.

2. The two central concepts of object-orientation are object classes and the relationships between them. These relationships include association, generalization-specialization, and aggregation.

3. The model assumes consistent user defined class specifications. The model only ensures that this consistency is maintained when composing specifications.

4. All three Rumbaugh models map into classes using a restricted notion of Rumbaugh semantics as defined in Chapter V. For instance, Rumbaugh's object model is used to define classes and their relationships. There is a dynamic model for each class which defines how the class responds to incoming events and a functional model to define how aggregate actions are transformed into component actions.

6.2 Classes

The building block of object-orientation is the object *class*. There are two types of classes: abstract and concrete. A *concrete* class is a blueprint from which instances of the class, called objects, are created. An *abstract* class is a class with no direct instances but whose descendents do have direct instances (83:61) and are discussed in detail in Section 6.2.10. Concrete classes have two parts: a class type and a class set. Class sets are discussed in Section 6.2.8. A *class type* defines the structure of an object and its response to external stimuli based its current state. A class type also has two components: attributes and operations. An attribute is an observable characteristic of an object and may either be constant or change over time. Attribute values do not necessarily uniquely define individual objects – two distinct objects may have identical attribute values. Although an object's attribute values are generally accessible to other objects, modifications of those values may only be done by the object itself, thus providing greater reuse potential. Objects communicate via message passing or through global events. In message passing, one object sends a message to another object. The message is then processed by the receiving object, possibly causing it to change state or to send additional messages. In an event driven system, objects generate events which are broadcast globally and "captured" by other objects in the system. In general, the object generating an event does not know its destination nor does the receiving object know its source.

In the theory-based object model, I define an object class type as a theory presentation as defined in Definition 4.2.1. For a given class C, S represents the sorts in the class including a class sort and any other sorts referenced in the theory while Ω is a set of theory operations and are used to represent attributes, operations, methods, and events. Each of these is discussed in detail below.

6.2.1 Sorts. *Sorts* are collections of values. Sorts may represent conventional data types such as integers, reals, or strings, or they may be abstract, representing such things as people, places, or ideas. The theory-based object model has two distinguished sort types: class sorts and state sorts. A *class sort* is the set of all possible object *names* in the class. Each object within the class has a unique name from the class sort. Objects themselves are not explicitly represented in a class type or specification: they are maintained external to the class type. By defining the class sort as a set of object names, objects may be referenced without having to maintain multiple copies of the object. A second set of sorts in a class type are the *state sorts.* Elements of a state sort are the individual class states as defined in the dynamic model.

6.2.2 Attributes. *Attributes* are visible operations that take an object name and return the value of a particular characteristic associated with that object. Attribute operations provide information about an object; they do not modify the object in any way. *State attributes* return values in the state sort representing the current state of an object. There is generally one state attribute per state sort. Multiple state sorts and attributes are used to define concurrent states and substates. A more complete discussion of state sorts and state attributes can be found in Section 6.2.7.

6.2.3 Methods. *Methods* are non-visible operations that modify an object's attribute values and are defined by *actions* in the dynamic model or *functions* in the functional model. In the theory-based object model, a method is not visible to external objects. Communication between objects is handled strictly by events. A method may modify none, some, or all of an object's non-state attribute values while event operations may only modify state attributes. Formal parameters of a method consist of an object name followed by other additional parameters. The return value of a method is the name of the object. Although the name is unchanged, returning the name of the object allows the nesting of method and attribute calls. Generally, the effect of a method on an object is defined by its effect on each of the non-state attributes of the object. Since the method returns the name of the object passed to it, a method invocation may be "embedded" in an attribute invocation allowing the effect of the method on the attribute to be precisely specified as shown below.

$$attribute\text{-name}(method\text{-}name(object\text{-}name)) = new\text{-}value\text{-}of\text{-}attribute$$

In a concrete class, it is assumed that the effect of each method is completely defined for all possible object states and input parameters.

Assumption VI.1 *In a concrete class, the effect of each method on each normal attribute in the class is completely defined for all states and input parameters.*

If the general result of a method applied to an object in a particular state cannot be computed or doesn't make sense (i.e., divide by zero, etc.), the effect of the method must still be defined. In most cases, if an object is in an inappropriate state prior to the method invocation or parameters passed to the method are invalid, there is simply no effect on the object. These preconditions are specified easily through the use of implication. For instance, for an *integer*, a *divide* method only makes sense when the divisor parameter is non-zero. Axioms describing the desired behavior are easily specified an shown below.

$$parameter \neq 0 \Rightarrow value(divide(integer, parameter)) = integer/parameter$$
$$parameter = 0 \Rightarrow value(divide(integer, parameter)) = integer$$

This assumption plays an important role in defining the effects of inheritance in Section 6.3.

Each class type has a *create* method used to create valid objects of the class type. This method is only used to create objects and assign initial values to attributes.

6.2.4 Events. *Events* are the visible operations that allow objects to communicate. Event operations are derived from the dynamic model and are only allowed to directly modify the state attributes of a class as shown in Figure 6.1. As a side effect, events may cause the other actions to be initiated. These actions might include the invocation of one or more methods or the generation of events to be sent to other objects. Each class type has a default *new* event which triggers the *create* method and initializes the object's state attributes. Event operations are discussed in more detail in Sections 6.3 and 6.6.

Figure 6.1 shows an example of a theory-based representation of an object class type in O-SLANG. The operations *date*, *bal*, and *acct-state* are attributes, *create-acct*, *credit* and *debit* are methods, and *new-acct*, *deposit* and *withdrawal* are events.

6.2.5 Operations. Theory-based object model *Operations* are visible operations that do not meet the criteria of an attribute, method, or an event. These operations are generally used to compute derived attributes, but are not restricted to this purpose; however, operations may not modify any of an object's attribute values.

A common example of a non-derived attribute operation is *attr-equal.* The *attr-equal* operation determines if two objects from a class are equal based on the non-state attribute's values. This operation is especially useful in specifying method invocation after receipt of a particular event. In this case, *attr-equal* specifies that the effect of a method on the non-state attributes is equivalent to the effect of an event on an object. In Figure 6.1, the axiom

$$\mathit{acct\text{-}state}(a) = \mathit{ok} \Rightarrow \mathit{acct\text{-}state}(\mathit{deposit}(a,x)) = \mathit{ok} \wedge \mathit{attr\text{-}equal}(\mathit{deposit}(a,x), \mathit{credit}(a,x))$$

```
class ACCT is
import Amnt, Date
class sort Acct
sorts Acct-State
operations
  attr-equal : Acct, Acct → Boolean
attributes
  date : Acct → Date
  bal : Acct → Amnt
state-attributes
  acct-state : Acct → Acct-State
methods
  create-acct : Date → Acct
  credit, debit : Acct, Amnt → Acct
states
  ok, overdrawn : → Acct-State
events
  new-acct : Date → Acct
  deposit, withdrawal : Acct, Amnt → Acct
axioms
 % state uniqueness and invariant axioms
  ok ≠ overdrawn;
  ∀ (a: Acct) acct-state(a) = ok ⇒ bal(a) ≥ 0;
  ∀ (a: Acct) acct-state(a) = overdrawn ⇒ bal(a) < 0;
 % operation definitions
  ∀ (a,a1: Acct) attr-equal(a, a1) ⇒ date(a) = date(a1) ∧ bal(a) = bal(a1);
 % method definitions
  ∀ (d: Date) date(create-acct(d)) = d ∧ bal(create-acct(d)) = 0;
  ∀ (a: Acct, x: Amnt) bal(credit(a,x)) = bal(a) + x
    ∧ date(credit(a,x)) = date(a) ∧ rate(credit(a,x)) = rate(a)
    ∧ int-date(credit(a,x)) = int-date(a) ∧ check-cost(credit(a,x)) = check-cost(a);
 % event definitions
  ∀ (d: Date) acct-state(new-acct(d))=ok ∧ attr-equal(new-acct(d), create-acct(d))
  ∀ (a: Acct, x: Amnt) acct-state(a)=ok
    ⇒ acct-state(deposit(a,x))=ok ∧ attr-equal(deposit(a,x), credit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=overdrawn ∧ bal(a) + x ≥ 0
    ⇒ acct-state(deposit(a,x))=ok ∧ attr-equal(deposit(a,x), credit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=overdrawn ∧ bal(a) + x < 0
    ⇒ acct-state(deposit(a,x))=overdrawn ∧ attr-equal(deposit(a,x), credit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=ok ∧ bal(a) ≥ x
    ⇒ acct-state(withdrawal(a,x))=ok ∧ attr-equal(withdrawal(a,x), debit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=ok ∧ bal(a) < x
    ⇒ acct-state(withdrawal(a,x))=overdrawn ∧ attr-equal(withdrawal(a,x), debit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=overdrawn
    ⇒ acct-state(withdrawal(a,x))=overdrawn ∧ attr-equal(withdrawal(a,x), a)
end-class
```

Figure 6.1 Object Class

states that if an account is in the *ok* state and a *deposit* event is received, the object stays in the *ok* state and that the effect of the *deposit* event on non-state attributes is equivalent to the effect of the *credit* method on the same object.

6.2.6 Axioms. Class *axioms* are first-order logic statements that must be true for any object of the class. They are used in class specifications to define the semantics of class operations as well as invariants between class attributes. In general, axioms define methods and events by describing their effects on attributes or through composition of other operations.

6.2.7 State. State is vital to object-orientation. As defined by Rumbaugh, *state* is an abstraction of an object's attribute values and is represented by a statechart in the dynamic model. A brief overview and the semantics of statecharts are presented in Section 5.4. To explicitly represent state in the theory-based object model, each class type has at least one *state sort* representing this abstraction of attribute values, a *state attribute* (one for each state sort) which returns an element from its associated state sort, a set of *states* (nullary operations) which are elements in a state sort, and a set of *state invariants* that describe constraints on class attributes that must hold true while in a given state. State attributes are only modified by events as defined by transitions in the class dynamic model. In Figure 6.1, the class state sort is *Acct-State*, the class state attribute is *acct-state*, the state constants are *ok* and *overdrawn*, and the state invariants are

$$\begin{aligned} &\textit{acct-state}(a) = \textit{ok} \Rightarrow \textit{bal}(a) \geq 0; \\ &\textit{acct-state}(a) = \textit{overdrawn} \Rightarrow \textit{bal}(a) < 0; \end{aligned}$$

These axioms state that when the balance of an account is greater than or equal to zero, the account must be in the *ok* state; however, when the balance of the account becomes less than zero, the state must become *overdrawn*. Although not a state invariant, the axiom

$$\textit{ok} \neq \textit{overdrawn}$$

is critical to the correct interpretation of the specification. It ensures that there are two distinct states *ok* and *overdrawn.* Without this axiom, a valid specification model might have a single element in the state sort, equivalent to both *ok* and *overdrawn.* Notice that the specification does not restrict the class state sort to only these values. Limiting the class state sort to these values would not permit valid extensions of the class state by subclasses as discussed in Section 6.3.4.1. The effect of these states on the behavior of the class as shown in Figure 6.2 and is represented by the axioms of the form

$$\textit{acct-state}(a) = ok \wedge bal(a) < x \Rightarrow \textit{acct-state}(withdrawal(a,x)) = overdrawn$$

This particular axiom requires that withdrawals be made only when the account is in the *ok* state prior to the *withdrawal* event and that if the account is overdrawn as a result of a withdrawal, the new state of the account becomes *overdrawn.* *Acct* state transitions and method invocations defined in the dynamic model are defined by the following axioms.

$$\begin{array}{l}
\textit{acct-state}(a) = ok \Rightarrow \textit{acct-state}(deposit(a,x)) = ok \wedge \textit{attr-equal}(deposit(a,x), credit(a,x)); \\
\textit{acct-state}(a) = overdrawn \wedge bal(a) + x \geq 0 \Rightarrow \textit{acct-state}(deposit(a,x)) = ok \\
\quad \wedge \textit{attr-equal}(deposit(a,x), credit(a,x)); \\
\textit{acct-state}(a) = overdrawn \wedge bal(a) + x < 0 \Rightarrow \textit{acct-state}(deposit(a,x)) = overdrawn \\
\quad \wedge \textit{attr-equal}(deposit(a,x), credit(a,x)); \\
\\
\textit{acct-state}(a) = ok \wedge bal(a) \geq x \Rightarrow \textit{acct-state}(withdrawal(a,x)) = ok \\
\quad \wedge \textit{attr-equal}(withdrawal(a,x), debit(a,x)); \\
\textit{acct-state}(a) = ok \wedge bal(a) < x \Rightarrow \textit{acct-state}(withdrawal(a,x)) = overdrawn \\
\quad \wedge \textit{attr-equal}(withdrawal(a,x), debit(a,x)); \\
\textit{acct-state}(a) = overdrawn \Rightarrow \textit{acct-state}(withdrawal(a,x)) = overdrawn \\
\quad \wedge \textit{attr-equal}(withdrawal(a,x), a)
\end{array}$$

These axioms require that the account state become *overdrawn* when a withdrawal is performed that makes the account balance less than zero and that the account state may only change back to *ok* when a deposit is made making the balance greater than or equal to zero.

Dynamic model statecharts allow concurrent states and substates. Concurrent statecharts are used when the attributes of a class type are partitionable into subsets as discussed in Section 5.4.1.

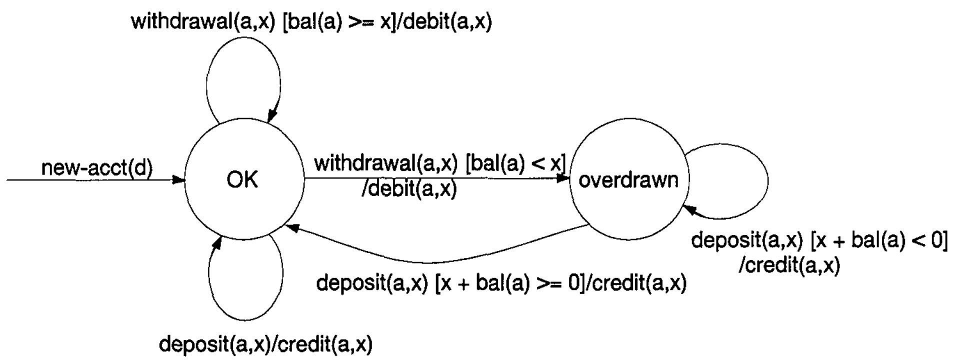
withdrawal(a,x) [bal(a) >= x]/debit(a,x)
new-acct(d)
OK
withdrawal(a,x) [bal(a) < x] /debit(a,x)
overdrawn
deposit(a,x) [x + bal(a) < 0] /credit(a,x)
deposit(a,x) [x + bal(a) >= 0]/credit(a,x)
deposit(a,x)/credit(a,x)

Figure 6.2 Account Dynamic Model

Formally, concurrent states are represented by multiple state attributes, one for each concurrent statechart. Substates are handled similarly to concurrent states – by adding additional substate attributes which are valid only when the class state attribute is in the appropriate state. Additional examples of using and specifying concurrent and substates are described in Section 6.3.4.1.

6.2.8 Class Set. The *Acct* class as defined in Figure 6.1 only specifies a template for creating new objects of the *Acct* class. However, Rumbaugh's informal model implies the ability to collectively manage a set of objects in a class. To provide this capability, a *class set* is created for each class defined (both abstract and concrete).

Definition 6.2.1 Class Set - *A class set is a class whose class sort is a set of objects from a previously defined object class, C. A class set includes a "class event" definition for each event in C such that the reception of a class event by a class set object sends the corresponding event in C to each object of type C contained in the class set object. If the class C is a subclass of $D_1 ... D_n$ then the class set of C is a subclass of the class sets of $D_1 ... D_n$.*

The class set creates a class type whose class sort is a set of objects and some basic operations on that set. Using the specifications of *TRIV* and *SET* as defined in Appendix E and basic category theory operations, the class set can be derived automatically as shown in Figure 6.3. The equivalent O-SLANG specification is shown in Figure 6.4.

```
spec ACCT-CLASS-COLIMIT is
  colimit of diagram
    nodes TRIV, ACCT, SET
    arcs  TRIV → ACCT : {E → Acct}
          TRIV → SET : {}
  end-diagram

spec ACCT-CLASS-SET is
  translate ACCT-CLASS-COLIMIT
  by {Set → Acct-Class, E → Acct}


spec ACCT-CLASS is
import ACCT-CLASS-SET
sort Acct-Class
operations
  new-acct-class : → Acct-Class
  withdrawal : Acct-Class, Amnt → Acct-Class
  deposit : Acct-Class, Amnt → Acct-Class
axioms
  new-acct-class() = empty-set;
  ∀ (a: Acct, ac: Acct-Class, x: Amnt) a ∈ ac ⇔ deposit(a,x) ∈ deposit(ac,x);
  ∀ (a: Acct, ac: Acct-Class, x: Amnt) a ∈ ac ⇔ withdrawal(a,x) ∈ withdrawal(ac,x)
end-class
```

Figure 6.3 SLANG Class Set Specification

```
class ACCT-CLASS is
contained-class ACCT
class sort Acct-Class
events
  new-acct-class : → Acct-Class
  withdrawal : Acct-Class, Amnt → Acct-Class
  deposit : Acct-Class, Amnt → Acct-Class
axioms
  new-acct-class() = empty-set;
  ∀ (a: Acct, ac: Acct-Class, x: Amnt) a ∈ ac ⇔ deposit(a,x) ∈ deposit(ac,x);
  ∀ (a: Acct, ac: Acct-Class, x: Amnt) a ∈ ac ⇔ withdrawal(a,x) ∈ withdrawal(ac,x)
end-class
```

Figure 6.4 O-SLANG Class Set Specification

The specification *ACCT-CLASS-COLIMIT* creates a new specification using the sort *E* in the specification *SET* as a formal parameter and instantiating it using the specification *TRIV*, unifying sort *E* in *SET* with the class sort *Acct* from the *ACCT* class. The diagram for this operation is shown in Figure 6.5.

$$\begin{tikzcd}
\text{Triv} \arrow[r, "i"] \arrow[d, "\{E \to \text{Acct}\}"'] \arrow[dr, "c"] & \text{Set} \arrow[d, "c"] \\
\text{Acct} \arrow[r, "c"] & \text{Acct-Class-Colimit} \arrow[d, "t"] \\
 & \text{Acct-Class-Set} \arrow[d, "i"] \\
 & \text{Acct-Class}
\end{tikzcd}$$

Figure 6.5 Colimit of Accounts

The colimit of *TRIV*, *SET*, and *ACCT* results in an intermediate specification with a set of account objects named *Set*. To eliminate ambiguity, the intermediate specification is translated in the *ACCT-CLASS-SET* specification such that the sort *Set* translates to *Acct-Class* and the sort *E* is translated to *Acct*. The renaming of *E* eliminates the sort name equivalence class {*E, Acct*} created by the colimit operation while the renaming of *Set* creates the class sort of the final specification *ACCT-CLASS*.

ACCT-CLASS imports the *ACCT-CLASS-SET* specification (and with it the *ACCT* class type definition) in order to add additional "class" events. These class events mirror the individual "object" events defined in the class type specification. Class set specifications simply distribute the event invocation to each object currently contained in the class set. Additional operations for selecting individual objects from the class set based on class attributes may also be specified by the designer using an aggregate or association qualifier. Use of qualifiers is discussed in Section 6.4.2 and 6.5.2.

6.2.9 Object-Valued Attributes. As discussed in Section 6.2, each object within the class has a unique name which allows other objects (including itself) to reference it. *Object-Valued attributes* are the mechanism used to reference external objects from within a class type definition and are the key to formally modeling association and aggregation. An object-valued attribute is a class attribute whose sort type is a set of object names (a class set sort). Object-valued attributes behave like normal class attributes. Formally, they are specification operations that take an object name and return an external object name or set of names.

The effects of methods on object-valued attributes are defined similarly to normal attributes. However, instead of directly specifying a new value for the object-valued attribute, an event from the object-valued attribute's class is sent to the object named by the object-valued attribute.

An example of using an object-valued attribute is shown in Figure 6.6. In this example, *Producer* is a class of objects which produce items to be stored in a buffer. *Buffer* is a class of simple buffer objects with *get* and *put* operations. The attribute *buffer-obj* is an object-valued attribute which holds the name of the specific buffer object in which the producer object stores its items. Once the producer and buffer are initialized, each *produce-item* event invokes the *produce* method which causes the item produced by the producer to be *put* into the buffer referenced by the *buffer* object-valued attribute as defined by the following axiom.

$$buffer\text{-}obj(produce(p,i)) = put(buffer\text{-}obj(p),i)$$

The *put* event is made available by importing the *Buffer* class type specification directly into the *Producer* specification. It is important to note that all modifications of objects referenced by object-valued attributes are accomplished by sending events to the objects instead of directly invoking methods since events ensure the object is in the appropriate state before a method is invoked. A direct method invocation could result in errors or inconsistencies in the referenced object.

```
class PRODUCER is
imports Buffer, Item
class sort Producer
sorts Producer-State
operations
  attr-equal : Producer, Producer → Boolean
attributes
  buffer-obj : Producer → Buffer
methods
  create-producer : Buffer → Producer
  produce : Producer, Item → Producer
events
  new-producer : Buffer → Producer
  produce-item : Producer, Item → Producer
axioms
  % operation definitions
  ∀ (p,p1: Producer) attr-equal(p, p1) ⇒ buffer-obj(p) = buffer-obj(p1);
  % event definitions
  ∀ (b: Buffer) attr-equal(new-producer(b), create-producer(b));
  ∀ (i: Item, p: Producer) attr-equal(produce-item(p,i), produce(p,i));
  % method definitions
  ∀ (b: Buffer) buffer-obj(create-producer(b)) = b;
  ∀ (i: Item, p: Producer) buffer-obj(produce(p,i)) = put(buffer-obj(p),i)
end-class
```

Figure 6.6 Object-Valued Attribute Example

6.2.10 Abstract Classes. Abstract classes and concrete classes are defined in the same manner with one exception. Because abstract classes are not instantiable, they are not required to fully define operations and thus Assumption VI.1 does not hold. Basically this allows specification operations (methods or operations) to be defined without fully defining their effect on every attribute. Only characteristics that must hold true in all subclasses need be specified. If an operation is not completely defined, it is called an ***abstract operation.*** Abstract classes are generally used to abstract out common elements of subclasses without having to fully define them.

6.3 Inheritance

In this research, inheritance holds to the substitution property as presented in Section 4.4. Simple inheritance is modelled using specifications morphisms between class specifications as defined in Definition 4.4.2 while multiple inheritance is defined in Defintion 4.4.3.

As discussed in Section 4.2.1, showing that a group of first order axioms are consistent is generally not possible; however, since by Assumption I.1 user defined specifications are consistent, I can develop rules to ensure this consistency is not violated when inheritance is applied. Because methods and events are defined in terms of their affect on attributes, these rules are developed based on the effect of methods and events on attributes. Table 6.1 shows the rules for methods and normal attributes. Basically this tables shows that in a subclass, new axioms may not be generated that define how a method defined in the superclass affects an attribute defined in the superclass.

Table 6.1 Method/Attribute Inheritance Rules

Method defined in	**Attribute defined in**	**Validity of axiomatic definition in subclass of effect of method on attribute**
Superclass	Superclass	Invalid
Superclass	Subclass	Valid
Subclass	Superclass	Valid
Subclass	Subclass	Valid

Because, by Assumption VI.1, superclass methods are completely defined over superclass attributes, additional axioms are not needed and can only lead to inconsistencies in the subclass. Notice that the other three combinations are valid. In fact, the other three combinations are required to ensure that methods introduced in the subclass and superclass are completely defined over attributes introduced in both the superclass and subclass. These same rules hold for events and state attributes as well; however, additional rules for substates and concurrent states are developed in Section 6.3.4.1.

An example of single inheritance using a subclass of the *ACCT* class, *SACCT* – a savings account class, is shown in Figure 6.7. The *import* statement includes all the sorts, operations, and axioms declared in the *ACCT* class directly into the new class while the class sort declaration *SAcct* $<$ *Acct* states that *SAcct* is a subsort of *Acct*, and as such, all operations and axioms that apply to an *Acct* object apply to a *SAcct* object as well. The dynamic model for *SACCT* is shown in Figure 6.8. The import operation defines a specification morphism between *ACCT* and *SACCT* while the subsort declaration completes the requirements of Definition 4.4.2 for inheritance. Therefore, *SACCT* is a valid subclass of *ACCT*, the substitution property holds, and internal class consistency is preserved.

6.3.1 Implications of the Substitution Property. Since my interpretation of the substitution property defined in Equation 4.3 implies that a subclass D must only act like its superclass in an environment design specifically for the superclass, it is possible for a subclass to have states in which it does not behave like its superclass. The substitution property only requires that these new states not be reachable via events available to the superclass. Consider the example shown in Figure 6.9. The statechart in Figure 6.9(b) extends the statechart in Figure 6.9(a) by adding a new state. A subclass object behaves like an object from its superclass as long as event $e4$ sending the object from state 3 to state 4 is not received. Once in state 4, the subclass no longer behaves like a member of the superclass. While definitely not wrong, this effect may not satisfy one's intuition of what is expected in a subclass – superclass relationship. The more restrictive interpretation given by Bourdeau and Cheng (14), as discussed in Section 4.4, forces a subclass object to reside only in states defined in the superclass object and thus would not allow state 4 to be added in the subclass.

6.3.2 Multiple Inheritance. Multiple inheritance is defined in Definition 4.4.2. To create an account that combines the features of a savings account with those of a checking account, *CACCT* (Figures 6.10 and 6.11), the colimit of classes *ACCT*, *SACCT*, *CACCT*, and morphisms from *ACCT* to *SACCT* and *CACCT* is computed as shown in Figure 6.12, where an arrow labeled with an "i" represents an import morphism and a "c" represents a morphism formed by the colimit operation. A simple extension of the colimit specification with the class sort definition

```
class SAcct is
import Acct, Rate
class sort SAcct < Acct
operations
  attr-equal : SAcct, SAcct → Boolean
attributes
  rate : SAcct → Rate
  int-date : SAcct → Date
methods
  create-sacct : Date → SAcct
  set-rate : SAcct, Date, Rate → SAcct
  comp-int : SAcct, Date → SAcct
events
  new-sacct : Date → SAcct
  rate-change : SAcct, Date, Rate → SAcct
  compute-interest : SAcct, Date → SAcct
axioms ∀ (d: Date, r: Rate, a, a1: SAcct)
  % operation definitions
  ∀ (a,a1: SAcct) attr-equal(a, a1) ⇒ rate(a) = rate(a1) ∧ int-date(a) = int-date(a1);
  % create method definition
  ∀ (d: Date) date(create-sacct(d)) = date(create-acct(d)) ∧ bal(create-sacct(d)) = bal(create-acct(d))
    ∧ acct-state(create-sacct(d)) = acct-state(create-acct(d)) ∧ int-date(create-sacct(d)) = d
    ∧ rate(create-sacct(d)) = 0;
  % credit method definitions
  ∀ (s: SAcct, a: Amnt) rate(credit(s,a)) = rate(s) ∧ int-date(credit(s,a)) = int-date(s);
  % debit method definitions
  ∀ (s: SAcct, a: Amnt) rate(debit(s,a)) = rate(s) ∧ int-date(debit(s,a)) = int-date(s);
  % set-rate method definitions
  ∀ (d: Date, r: Rate, a: SAcct) rate(set-rate(a,d,r)) = r ∧ int-date(set-rate(a,d,r)) = d
    ∧ bal(set-rate(a,d,r)) = bal(a) ∧ date(set-rate(a,d,r)) = date(a);
  % comp-int method definitions
  ∀ (d: Date, a: SAcct) rate(comp-int(a,d)) = rate(a) ∧ int-date(comp-int(a,d)) = d
    ∧ bal(a) ≥ 0 ⇒ bal(comp-int(a,d)) = bal(a) + rate(a) * ((d - int-date(a))/days-per-year(d))
    ∧ bal(a) ≤ 0 ⇒ bal(comp-int(a,d)) = bal(a) ∧ date(comp-int(a,d)) = date(a);
  % new event definition
  ∀ (d: Date) acct-state(new-sacct(d)) = ok ∧ attr-equal(new-sacct(d), create-sacct(d));
  % rate-change event definitions
  ∀ (d: Date, r: Rate, a: SAcct) acct-state(a) = ok ⇒ acct-state(rate-change(a,d,r)) = ok
    ∧ attr-equal(rate-change(a,d,r),set-rate(comp-int(a,d),d,r));
  ∀ (d: Date, r: Rate, a: SAcct) acct-state(a) = overdrawn
    ⇒ acct-state(rate-change(a,d,r)) = overdrawn
      ∧ attr-equal(rate-change(a,d,r),set-rate(comp-int(a,d),d,r));
  % compute-interest event definitions
  ∀ (d: Date, a: SAcct) acct-state(a) = ok ⇒ acct-state(compute-interest(a,d)) = ok
    ∧ attr-equal(compute-interest(a,d),comp-int(a,d));
  ∀ (d: Date, a: SAcct) acct-state(a) = overdrawn ⇒ acct-state(compute-interest(a,d)) = overdrawn
    ∧ attr-equal(compute-interest(a,d),a)
end-class
```

Figure 6.7 Savings Class

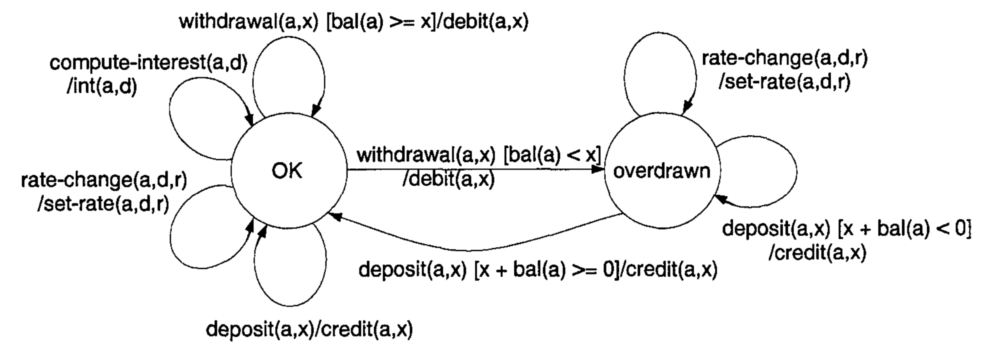

Figure 6.8 Savings Account Dynamic Model

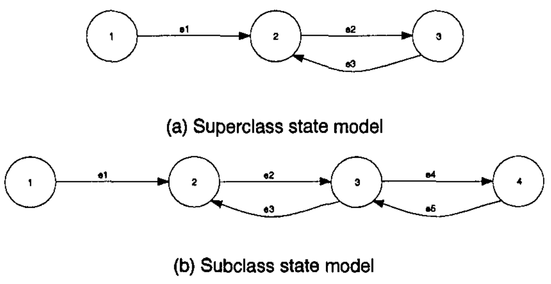

Figure 6.9 Subclass State Extension

$$\textit{Comb-Acct} < \textit{SAcct}, \textit{CAcct}$$

yields the desired class where *Comb-Acct* is a subclass of both *SAcct* and *CAcct.* Figures 6.13, 6.14, and 6.15 show the "long" version of the combined specification with all the attributes, methods, and events inherited by the *Comb-Acct* class.

6.3.3 Subclasses and Class Sort Subsorts. The subclass – superclass relationship corresponds to the subsort – supersort relationship of the class sort. Since a subclass has all the features of the superclass and subclass object can be substituted for superclass objects, it subclass objects are in fact, members of the superclass. Since there is a one-to-one correspondence between objects in a class and names in the class sort, an object in a subclass must have a name in the subsort as well as in the class sort.

The O-SLANG subsort operator < defines a subset relationship among sorts such that for two sorts, A and B in specification S, $A < B \Rightarrow A' \subseteq B'$ where A' and B' are sets representing A

```
class CAcct is
import Acct class sort CAcct < Acct
operations
  attr-equal : CAcct, CAcct → Boolean
attributes
  check-cost : CAcct → Amnt
methods
  create-cacct : Date → CAcct
  set-check-cost : CAcct, Amnt → CAcct
events
  new-cacct : Date → CAcct
  change-check-cost : CAcct, Amnt → CAcct
  write-check : CAcct, Amnt → CAcct
axioms ∀ (a: CAcct, x: Amnt)
 % operation definitions
  attr-equal(a, a1) ⇒ check-cost(a) = check-cost(a1);
 % create method definition
  date(create-cacct(d)) = date(create-acct(d));
  bal(create-cacct(d)) = bal(create-acct(d));
  acct-state(create-cacct(d)) = acct-state(create-acct(d));
  check-cost(create-cacct(d)) = 0;
 % credit method definitions
  ∀ (c: CAcct, a: Amnt) check-cost(credit(c,a)) = check-cost(c);
 % debit method definitions
  ∀ (c: CAcct, a: Amnt) check-cost(debit(c,a)) = check-cost(c);
 % set-check-cost method definition
  ∀ (a: CAcct, x: Amnt) check-cost(set-check-cost(a, x)) = x;
 % new event definition
  ∀ (d: Date) acct-state(new-cacct(d)) = ok ∧ attr-equal(new-cacct(d), create-acct(d));
 % write-check-cost event definition
  ∀ (a: CAcct, x: Amnt) acct-state(a) = ok ∧ bal(a) ≥ x
     ⇒ acct-state(write-check(a,x)) = ok ∧ attr-equal(write-check(a,x), debit(a,x);
  ∀ (a: CAcct, x: Amnt) acct-state(a) = ok ∧ bal(a) < x
     ⇒ acct-state(write-check(a,x)) = overdrawn ∧ attr-equal(write-check(a,x), debit(a,x);
  ∀ (a: CAcct, x: Amnt) acct-state(a) = overdrawn
     ⇒ acct-state(write-check(a,x)) = overdrawn ∧ attr-equal(write-check(a,x), debit(a,x);
 % set-check-cost method definition
  ∀ (a: CAcct, x: Amnt) acct-state(a) = ok
     ⇒ acct-state(write-check(a,x)) = ok ∧ attr-equal(change-check-cost(a,x), set-check-cost(a,x)) ;
  ∀ (a: CAcct, x: Amnt) acct-state(a) = overdrawn
     ⇒ acct-state(write-check(a,x)) = overdrawn ∧ attr-equal(change-check-cost(a,x), set-check-cost(a,x))
end-class
```

Figure 6.10 Checking Class

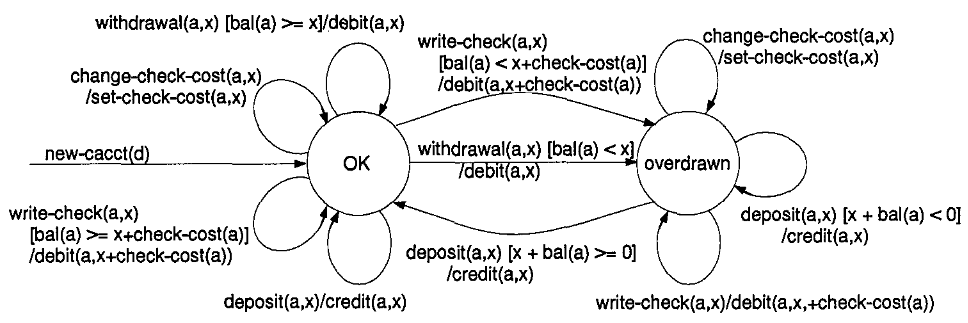
withdrawal(a,x) [bal(a) >= x]/debit(a,x)
change-check-cost(a,x) /set-check-cost(a,x)
new-cacct(d)
OK
write-check(a,x) [bal(a) >= x+check-cost(a)] /debit(a,x+check-cost(a))
deposit(a,x)/credit(a,x)
write-check(a,x) [bal(a) < x+check-cost(a)] /debit(a,x+check-cost(a))
withdrawal(a,x) [bal(a) < x] /debit(a,x)
deposit(a,x) [x + bal(a) >= 0] /credit(a,x)
overdrawn
change-check-cost(a,x) /set-check-cost(a,x)
deposit(a,x) [x + bal(a) < 0] /credit(a,x)
write-check(a,x)/debit(a,x,+check-cost(a))

Figure 6.11 Checking Account Dynamic Model

$$\begin{tikzcd}
 & \text{Acct} \arrow[dl, "i"'] \arrow[dr, "i"] \arrow[dd, "c"] & \\
\text{SAcct} \arrow[dr, "c"'] & & \text{CAcct} \arrow[dl, "c"] \\
 & \text{Comb-Acct} &
\end{tikzcd}$$

Figure 6.12 Colimit of Accounts

and B in a model of S. Thus, if a class D is a subclass of a class C, then the class sort of D, D_{cs}, is a subsort of the class sort of class C, C_{cs}, or $D'_{cs} \subseteq C'_{cs}$.

6.3.4 Behavioral Inheritance. There are two methods of behavioral specification in OMT: the dynamic model and the functional model. Dynamic behavior is specified by a statechart for each class while functional behavior is specified at the system, or aggregate level by data flow diagrams. Section 6.3.4.1 defines the effects of inheritance on the dynamic model while Section 6.3.4.2 describes the effects of inheritance of functional behavior.

6.3.4.1 Dynamic Inheritance. Because the effect of a method or event is based on the object's state and must be equivalent to the effect of a method or event on a superclass object, modification of inherited dynamic behavior must conform to certain rules. Since each class has possibly multiple "state", "substate", or "concurrent state" attributes (as described in Section

class COMB-ACCT is
import SAcct, CAcct
class sort Comb-Acct < SAcct, CAcct
sorts Acct-State
operations
  attr-equal : Comb-Acct, Comb-Acct → Boolean
attributes
  date : Comb-Acct → Date
  bal : Comb-Acct → Amnt
  rate : Comb-Acct → Rate
  int-date : Comb-Acct → Date
  check-cost : Comb-Acct → Amnt
state-attributes
  acct-state : Comb-Acct → Acct-State
methods
  create-acct : Date → Comb-Acct
  create-sacct : Date → Comb-Acct
  create-cacct : Date → Comb-Acct
  create-comb-acct : Date → Comb-Acct
  credit : Comb-Acct, Amnt → Comb-Acct
  debit : Comb-Acct, Amnt → Comb-Acct
  set-rate : Comb-Acct, Date, Rate → Comb-Acct
  int : Comb-Acct, Date → Comb-Acct
  set-check-cost : Comb-Acct, Amnt → Comb-Acct
  write-check : Comb-Acct, Amnt → Comb-Acct
states
  ok : → Acct-State
  overdrawn : → Acct-State
events
  new-acct : Date → Comb-Acct
  new-sacct : Date → Comb-Acct
  new-cacct : Date → Comb-Acct
  new-comb-acct : Date → Comb-Acct
  deposit : Comb-Acct, Amnt → Comb-Acct
  withdrawal : Comb-Acct, Amnt → Comb-Acct
  rate-change : Comb-Acct, Date, Rate → Comb-Acct
  compute-interest : Comb-Acct, Date → Comb-Acct
  change-check-cost : Comb-Acct, Amnt → Comb-Acct
  write-check : Comb-Acct, Amnt → Comb-Acct

Figure 6.13 Combined Account Signature

axioms

```
 % state uniqueness and invariant axioms
  ok ≠ overdrawn;
  ∀ (a: Acct) acct-state(a) = ok ⇒ bal(a) ≥ 0;
  ∀ (a: Acct) acct-state(a) = overdrawn ⇒ bal(a) < 0;
 % operation definitions
  ∀ (a,a1: Acct) attr-equal(a, a1) ⇒ date(a) = date(a1) ∧ bal(a) = bal(a1);
  ∀ (a,a1: SAcct) attr-equal(a, a1) ⇒ rate(a) = rate(a1) ∧ int-date(a) = int-date(a1);
  ∀ (a,a1: CAcct) attr-equal(a, a1) ⇒ check-cost(a) = check-cost(a1);
 % create-acct method definition
  ∀ (d: Date) date(create-acct(d)) = d ∧ bal(create-acct(d)) = 0;
 % create-sacct method definition
  ∀ (d: Date) date(create-sacct(d)) = date(create-acct(d))
     wedge bal(create-sacct(d)) = bal(create-acct(d)) wedge rate(create-sacct(d)) = 0
     wedge int-date(create-sacct(d)) = d;
 % create-cacct method definition
  ∀ (d: Date) date(create-cacct(d)) = date(create-acct(d))
     ∧ bal(create-cacct(d)) = bal(create-acct(d)) ∧ check-cost(create-cacct(d)) = 0;
 % create-comb-cacct method definition
  ∀ (d: Date) date(create-comb-acct(d)) = date(create-acct(d))
     wedge bal(create-comb-acct(d)) = bal(create-acct(d)) wedge rate(create-comb-acct(d)) = 0
     wedge int-date(create-comb-acct(d)) = d ∧ check-cost(create-comb-acct(d)) = 0;
 % credit method definition
  ∀ (a: Acct, x: Amnt) bal(credit(a,x)) = bal(a) + x
     ∧ date(credit(a,x)) = date(a) ∧ rate(credit(a,x)) = rate(a)
     ∧ int-date(credit(a,x)) = int-date(a) ∧ check-cost(credit(a,x)) = check-cost(a);
 % debit method definitions
  ∀ (a: Acct, x: Amnt) bal(debit(a,x)) = bal(a) - x
     ∧ date(debit(a,x)) = date(a) ∧ rate(debit(a,x)) = rate(a)
     ∧ int-date(debit(a,x)) = int-date(a) ∧ check-cost(debit(a,x)) = check-cost(a);
 % set-rate method definitions
  ∀ (a: Acct, d: Date, r: Rate) rate(set-rate(a,d,r)) = r
     ∧ int-date(set-rate(a,d,r)) = d ∧ bal(set-rate(a,d,r)) = bal(a)
     ∧ date(set-rate(a,d,r)) = date(a);
 % comp-int method definitions
  ∀ (a: Acct, d: Date) rate(comp-int(a,d)) = rate(a)
     ∧ int-date(comp-int(a,d)) = d ∧ date(comp-int(a,d)) = date(a);
  ∀ (a: Acct, d: Date) bal(a) ≤ 0 ⇒ bal(comp-int(a,d)) = bal(a);
  ∀ (a: Acct, d: Date) bal(a) ≥ 0 ⇒ bal(comp-int(a,d)) = bal(a)
     + rate(a) * ((d - int-date(a))/days-per-year(d));
 % set-check-cost method definition
  ∀ (a: Acct, x: Amnt) check-cost(set-check-cost(a, x)) = x
     ∧ bal(set-check-cost(a,x)) = bal(a) ∧ data(set-check-cost(a,x) = date(a)
```

Figure 6.14 Combined Account Class Axioms

```
  % new event definition
   ∀ (d: Date) acct-state(new-acct(d)) = ok ∧ attr-equal(new-acct(d), create-acct(d));
   ∀ (d: Date) acct-state(new-sacct(d)) = ok ∧ attr-equal(new-sacct(d), create-sacct(d));
   ∀ (d: Date) acct-state(new-cacct(d)) = ok ∧ attr-equal(new-cacct(d), create-cacct(d));
   ∀ (d: Date) acct-state(new-comb-acct(d)) = ok ∧ attr-equal(new-comb-acct(d), create-comb-acct(d));
  % deposit event definition
   ∀ (a: Acct, x: Amnt) acct-state(a) = ok ⇒ acct-state(deposit(a,x)) = ok
     ∧ attr-equal(deposit(a,x), credit(a,x));
   ∀ (a: Acct, x: Amnt) acct-state(a) = overdrawn ∧ bal(a) + x ≥ 0
     ⇒ acct-state(deposit(a,x)) = ok ∧ attr-equal(deposit(a,x), credit(a,x));
   ∀ (a: Acct, x: Amnt) acct-state(a) = overdrawn ∧ bal(a) + x < 0
     ⇒ acct-state(deposit(a,x)) = overdrawn ∧ attr-equal(deposit(a,x), credit(a,x));
  % withdrawal event definition
   ∀ (a: Acct, x: Amnt) acct-state(a) = ok ∧ bal(a) ≥ x
     ⇒ acct-state(withdrawal(a,x)) = ok ∧ attr-equal(withdrawal(a,x), debit(a,x));
   ∀ (a: Acct, x: Amnt) acct-state(a) = ok ∧ bal(a) < x
     ⇒ acct-state(withdrawal(a,x)) = overdrawn ∧ attr-equal(withdrawal(a,x), debit(a,x));
   ∀ (a: Acct, x: Amnt) acct-state(a) = overdrawn
     ⇒ acct-state(withdrawal(a,x)) = overdrawn ∧ attr-equal(withdrawal(a,x), a);
  % rate-change event definitions
   ∀ (a: Acct, d: Date, r: Rate) acct-state(a) = ok
     ⇒ acct-state(rate-change(a,d,r)) = ok ∧ attr-equal(rate-change(a,d,r),set-rate(comp-int(a,d),d,r));
   ∀ (a: Acct, d: Date, r: Rate) acct-state(a) = overdrawn
     ⇒ acct-state(rate-change(a,d,r)) = overdrawn
     ∧ attr-equal(rate-change(a,d,r),set-rate(comp-int(a,d),d,r));
  % compute-interest event definitions
   ∀ (a: Acct, d: Date) acct-state(a) = ok
     ⇒ acct-state(compute-interest(a,d)) = ok ∧ attr-equal(compute-interest(a,d),comp-int(a,d));
   ∀ (a: Acct, d: Date) acct-state(a) = overdrawn
     ⇒ acct-state(compute-interest(a,d)) = overdrawn ∧ attr-equal(compute-interest(a,d),a);
  % write-check-cost event definition
   ∀ (a: Acct, x: Amnt) acct-state(a) = ok
     ∧ bal(a) ≥ x ⇒ acct-state(write-check(a,x)) = ok
     ∧ attr-equal(write-check(a,x), debit(a,x));
   ∀ (a: Acct, x: Amnt) acct-state(a) = ok
     ∧ bal(a) < x ⇒ acct-state(write-check(a,x)) = overdrawn
     ∧ attr-equal(write-check(a,x), debit(a,x));
   ∀ (a: Acct, x: Amnt) acct-state(a) = overdrawn
     ⇒ acct-state(write-check(a,x)) = overdrawn ∧ attr-equal(write-check(a,x), debit(a,x));
  % set-check-cost method definition
   ∀ (a: Acct, x: Amnt) acct-state(a) = ok
     ⇒ acct-state(write-check(a,x)) = ok ∧ attr-equal(change-check-cost(a,x), set-check-cost(a,x)) ;
   ∀ (a: Acct, x: Amnt) acct-state(a) = overdrawn
     ⇒ acct-state(write-check(a,x)) = overdrawn
     ∧ attr-equal(change-check-cost(a,x), set-check-cost(a,x))
end-class
```

Figure 6.15 Combined Account Class Axioms (Continued)

6.2) that explicitly capture its defining statecharts, the requirements of the substitution property (Equation 4.3) apply to dynamic behavior and are captured in the specification morphism requirement for inheritance. Figures 6.16 through 6.20 show examples of how a Rumbaugh statechart may, and may not be modified.

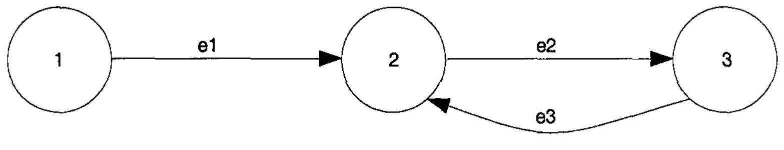

Figure 6.16 Superclass Dynamic Behavior

Figure 6.16 shows the statechart for a superclass, C. This statechart translates into the following axioms.

$$
\begin{aligned}
&state(o) = 1 \Rightarrow state(e1(o)) = 2\\
&state(o) = 1 \Rightarrow state(e2(o)) = 1\\
&state(o) = 1 \Rightarrow state(e3(o)) = 1\\
&state(o) = 2 \Rightarrow state(e1(o)) = 2\\
&state(o) = 2 \Rightarrow state(e2(o)) = 3\\
&state(o) = 2 \Rightarrow state(e3(o)) = 2\\
&state(o) = 3 \Rightarrow state(e1(o)) = 3\\
&state(o) = 3 \Rightarrow state(e2(o)) = 3\\
&state(o) = 3 \Rightarrow state(e3(o)) = 2
\end{aligned}
$$

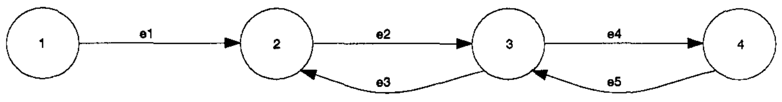

Figure 6.17 Inheritance of Dynamic Behavior - State Extension

To be valid, the subclass statechart must translate into a set of axioms that incorporate the axioms of its superclass (i.e., the axioms from the superclass must be theorems in the subclass). Figure 6.17 shows the statechart for a valid subclass, C_b, of the superclass C. This statechart translates to the following axioms. (NOTE: From this point forward in this section, axioms defining no change are omitted for brevity.)

$$
\begin{aligned}
state(o) = 1 &\Rightarrow state(e1(o)) = 2 \\
state(o) = 2 &\Rightarrow state(e2(o)) = 3 \\
state(o) = 3 &\Rightarrow state(e3(o)) = 2 \\
state(o) = 3 &\Rightarrow state(e4(o)) = 4 \\
state(o) = 4 &\Rightarrow state(e5(o)) = 3
\end{aligned}
$$

To determine if C_b is a valid subclass of C, the axioms of C must appear as theorems in C_b and the internal class consistency conditions must hold. Clearly, the axioms of C are theorems in C_b since each axiom in C appears as an axiom in C_b. Note that, as shown in Figure 4.7, it is possible to add axioms that are inconsistent. However, in this case, there are no inconsistent axioms and thus C_b is a valid subclass of C.

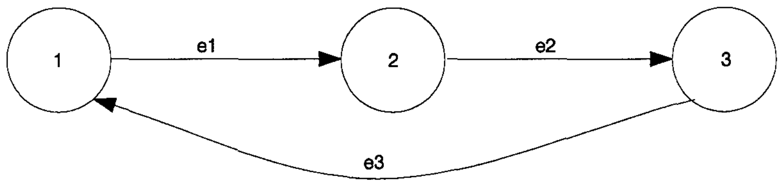

Figure 6.18 Inheritance of Dynamic Behavior - Illegal

Figure 6.18 shows the statechart for an invalid subclass, C_c, of the superclass C. This statechart translates to the following axioms:

$$
\begin{aligned}
state(o) = 1 &\Rightarrow state(e1(o)) = 2 \\
state(o) = 2 &\Rightarrow state(e2(o)) = 3 \\
state(o) = 3 &\Rightarrow state(e3(o)) = 1
\end{aligned}
$$

Clearly class C_c is not a valid subclass of C since the axioms of C are not theorems in C_c. For valid inheritance, there must be a specification morphism from the superclass to the subclass such that the axioms in the superclass are theorems in the subclass. If such a morphism existed in this case, the following axioms must both true in C_c:

$$
\begin{aligned}
state(o) = 3 &\Rightarrow state(e3(o)) = 2 (fromC_c) \\
state(o) = 3 &\Rightarrow state(e3(o)) = 1 (fromC)
\end{aligned}
$$

Obviously, the two axioms are conflicting unless state 1 and state 2 are equivalent. However, as part of the translation process, axioms are generated which specify the uniqueness of states. Therefore the axioms of C are not theorems in C_c and, therefore, C_c is not a valid subclass of C. To ensure class consistency, extension of the statechart may not allow new transitions from a state defined in the superclass using events defined in the superclass. This rule is analogous to not allowing superclass methods to modify superclass attributes as shown in Table 6.1.

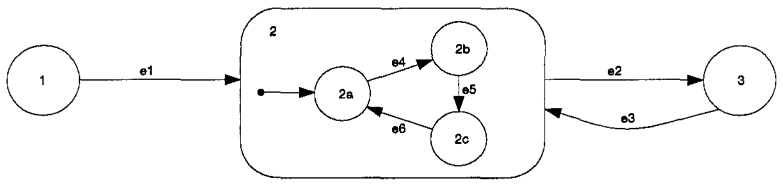

Figure 6.19 Inheritance of Dynamic Behavior - SubState Statechart

Figure 6.19 shows the statechart for a valid subclass, C_d, of the superclass C this time refined using substates. This statechart translates to the following axioms:

$$
\begin{array}{l}
state(o) = 1 \Rightarrow state(e1(o)) = 2 \wedge substate2(e1(o)) = 2a \\
state(o) = 3 \Rightarrow state(e3(o)) = 2 \wedge substate2(e3(o)) = 2a \\
state(o) = 2 \Rightarrow state(e2(o)) = 3 \\
state(o) = 2 \wedge substate2(o) = 2a \Rightarrow state(e4(o)) = 2 \wedge substate2(e4(o)) = 2b \\
state(o) = 2 \wedge substate2(o) = 2b \Rightarrow state(e5(o)) = 2 \wedge substate2(e5(o)) = 2c \\
state(o) = 2 \wedge substate2(o) = 2c \Rightarrow state(e6(o)) = 2 \wedge substate2(e6(o)) = 2a
\end{array}
$$

Although the post-conditions describing the effect of events $e1$ and $e3$ have changed in C_d, the axioms from C can be derived from those in C_d; thus the axioms of C are theorems in C_d. Since there are no inconsistent axioms, the internal class consistency conditions hold and C_d is a valid subclass of C. It is important to note that the only time a substate attribute affects the behavior of an object is when the object is in state 2 and that if event $e2$ is applied any time the state of a C_d object is in state 2, regardless of the substate, the state transitions to 3 as required by C.

In general, substates can be used freely to refine a superclass statechart as long as superclass events which cause transitions from the superstate are not used within the substate statechart (i.e., the superstate exit transition is not overridden). Attempting to override an exiting transition from the superstate results in inconsistent axioms as shown by the axioms below which are the result of replacing event $e6$ by $e2$ in Figure 6.19.

$$\begin{array}{l} state(o) = 2 \Rightarrow state(e2(o)) = 3 \\ state(o) = 2 \wedge substate2(o) = 2c \Rightarrow state(e2(o)) = 2 \wedge substate2(e4(o)) = 2a \end{array}$$

Obviously, both axioms cannot be true since $state(e2(o))$ cannot be both 2 and 3 simultaneously. Note that this interpretation of the statechart does not satisfy the intent of such a statechart as defined by Rumbaugh (83:97). According to Rumbaugh, the intent of such a statechart would be to override the effect of event $e2$ when in state $2c$; however, if the axioms implemented Rumbaugh intended semantics, the statechart would violate the substitution property.

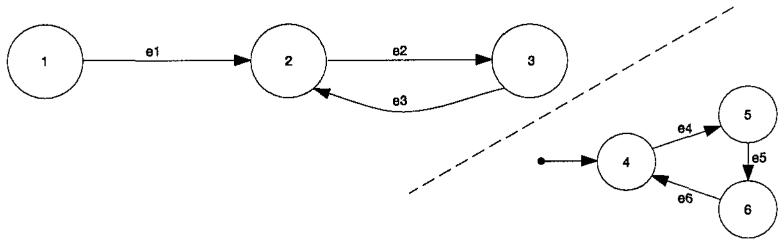

Figure 6.20 Inheritance of Dynamic Behavior - Concurrent Statechart

Figure 6.20 shows the statechart for a valid subclass, C_e, of the superclass C, this time refined using concurrent states. This statechart translates to the following axioms.

$$\begin{array}{l} state(o) = 1 \Rightarrow state(e1(o)) = 2 \\ state(o) = 2 \Rightarrow state(e2(o)) = 3 \\ state(o) = 3 \Rightarrow state(e3(o)) = 2 \\ conc\text{-}state(o) = 4 \Rightarrow conc\text{-}state(e4(o)) = 5 \\ conc\text{-}state(o) = 5 \Rightarrow conc\text{-}state(e5(o)) = 6 \\ conc\text{-}state(o) = 6 \Rightarrow conc\text{-}state(e6(o)) = 4 \end{array}$$

In this example, C_e, as defined by Figure 6.20, is a valid subclass since only new states and events are used in the concurrent statechart. However, not all concurrent statecharts satisfy class consistency conditions. Concurrent statecharts are intended to be used when the attributes of a class are partitionable into distinct subsets. If class attributes are paritionable, then the actions of each concurrent component may only affect attributes in a single partition. When this is not the case, inconsistencies between the statechart components may result. Therefore, when the dynamic model is extended in a subclass by a concurrent statechart, the actions of the concurrent statechart component must modify only the attributes defined in the subclass. This implies that only methods defined in the subclass may be used in a subclass concurrent statechart component and that those methods may not modify attributes defined in the superclass.

In terms of statecharts, then, the substitution property requires that superclass dynamic model be included, as is, into all subclass dynamic models. Additions to the superclass statechart, including substates and concurrent states, that do not violate class consistency conditions are the only allowable extensions that satisfy the substitution property.

6.3.4.2 Functional Inheritance. The second type of behavioral inheritance involves inheritance of the functional model. In general, functions define how data is transformed in the system without regard to when these transformations take place (83). Functions defined in the functional model correspond to actions defined in the dynamic model and are generally modeled at the aggregate level. Inheritance from an abstract class often allows the subclass the freedom to define the actions specified in its dynamic model. In the case of an abstract operation, there is no functional definition or constraints put on the function of the operation in the abstract class; therefore, specialization of such an operation in a concrete class must include its complete functional definition.

In the case of inheritance from concrete classes, the functional behavior (as defined by methods) is completely defined in the superclass. This greatly restricts the ability to specialize these functions in subclasses; however, this is not a problem since overriding of function behavior is not allowed by the definition of inheritance and overriding for other reasons (efficiency, etc.) is purely a design issue and not important in domain modeling. Therefore, the only real way to functionally specialize a subclass is to add new functions in the subclass or to specify the effects of existing superclass functions on new subclass attributes. These new functions would be derived from new actions defined in the dynamic model and only need axiomatic definition in the functional model. New data flow diagrams may be developed to further define the actions, or, if simple enough, axioms may be written directly to define the action as a method. For example, the dynamic model for the savings account class (Figure 6.8) identifies two new actions: *int* and *set-rate*. The definition of these actions are relatively simple and defined without the need for a new data flow diagram, as shown in Figure 6.7. More complex actions might require new data flow diagrams which would generate additional methods to help compute the defined actions.

Introducing new attributes in a subclass requires the extending existing superclass methods definitions to include their effect on the subclass attributes. As long as the requirements specified in Table 6.1 are followed, class consistency is maintained.

6.4 Associations

Rumbaugh defines a *link* as a physical or conceptual connection between object instances while an *association* is a group of links with a common structure and semantics (83). The relationship between associations and links is similar to the relationship between classes and objects. In this model, a link defines what object classes may be connected along with any link attributes, operations, or qualifiers. *Link attributes* and *link operations* are attributes and operations that do not belong to any one of the objects involved in a link, but exist only when there is a link between objects. An association *qualifier* is an attribute that is used to select an associated object based on the value of the qualifier. Often, a qualifier is used to reduce a one-to-many association to a one-to-one association based on the qualifier value.

In this model, associations are represented generically, as a specification that defines a sets of individual links. A link defines a specification that uses object-valued attributes to reference individual objects from two or more classes. Links may also define link attributes, operations, or qualifiers in a manner identical to object classes. Basically, a link is a class whose class-set is an association.

Definition 6.4.1 Link *A link is an object class type with two or more object-valued attributes.*

An example of a link specification between a class of customers (Figure 6.21) and the *ACCT* class is shown in Figure 6.22. To improve reusability and maintainability, integration of account numbers or references directly into the associated classes is not desired. Therefore, a link specification, *CA-Link*, is created to associate customers with their accounts. The *CA-Link* link specification has two object-valued attributes, *customer* and *acct*, and a method to create new instances of the association. Thus, the *CA-Link* link specification can relate objects from the two classes without embedding internal references into the classes themselves. Although the names of the object-valued attributes and sorts correspond to the *CUSTOMER* and *ACCT* classes, the link specification does not formally tie the classes together. This relationship is actually formalized in an aggregate specification as defined in Section 6.5.

An associations is a set of links and is represented as such in this model.

Definition 6.4.2 Association *An association is the class set of a link specification.*

An association between the *ACCT* class and the *CUSTOMER* class is shown in Figure 6.23. The *CA-LINK* class has two object-valued attributes, *customer* and *account*, and a method to create new instances of the association. The *CUST-ACCT* class defines a set of *CA-Link* objects while its axioms define the multiplicity relationships between accounts and customers. In this case, there is exactly one customer per account while each customer may have one or more accounts. Associations with more than two classes are handled in a similar manner by simply adding additional object-valued attributes.

```
class CUSTOMER is
import Name, Address, Cust-No
class sort Customer
operations
  attr-equal : Customer, Customer → Boolean
attributes
  name : Customer → Name
  address : Customer → Address
  cust-no : Customer → Cust-No
methods
  create-customer : Name, Address, Cust-No → Customer
  update : Customer, Name, Address, Cust-No → Customer
events
  new-customer : Name, Address, Cust-No → Customer
  update-customer : Customer, Name, Address → Customer
axioms
 % operation definition
  ∀ (c,c1: Customer) attr-equal(c,c1) ⇔ name(c) = name(c1)
    ∧ address(c) = address(c1) ∧ cust-no(c) = cust-no(c1);
 % create method definition
  ∀ (n: Name, a: Address, cn: Cust-No)
    name(create-customer(n,a,cn)) = n ∧ address(create-customer(n,a,cn)) = a
    ∧ cust-no(create-customer(n,a,cn)) = cn;
 % update method definition
  ∀ (c: Customer, n: Name, a: Address, cn: Cust-No)
    name(update(c,n,a,cn)) = n ∧ address(update(c,n,a,cn)) = a
    ∧ cust-no(update(c,n,a,cn)) = cn;
 % new event definition
  ∀ (n: Name, a: Address, cn: Cust-No)
    attr-equal(new-customer(n,a,cn), create-customer(n,a,cn));
 % update-customer event definition
  ∀ (c: Customer, n: Name, a: Address, cn: Cust-No)
    attr-equal(update-customer(c,n,a,cn), update(c,n,a,cn))
end-class
```

Figure 6.21 Customer Class

```
link CA-LINK is
class sort CA-Link
sorts Customer, Account
operations
  attr-equal : CA-Link, CA-Link → Boolean
attributes
  customer : CA-Link → Customer
  account : CA-Link → Account
methods
  create-ca-link : Customer, Account → CA-Link
events
  new-ca-link : Customer, Account → CA-Link
axioms
 % operation definition
  ∀ (c,c1: Customer)
    attr-equal(c,c1) ⇔ customer(c) = customer(c1) ∧ account(c) = account(c1);
 % create method definition
  ∀ (c: Customer, a: Account)
    customer(create-ca-link(c,a)) = c ∧ account(create-ca-link(c,a)) = a;
 % new event definition
  ∀ (c: Customer, a: Account)
    attr-equal(new-ca-link(c,a), create-ca-link(c,a))
end-link
```

Figure 6.22 Customer Account Link

```
association CUST-ACCT is
link-class CA-Link
class sort Cust-Acct
sorts Accounts, Customers
methods
  image : Cust-Acct, Customer → Accounts
  image : Cust-Acct, Account → Customers
events
  new-cust-acct : → Cust-Acct
axioms
 % multiplicity axioms
  ∀ (ca: Cust-Acct, c: Customer) size(image(ca, c)) ≥ 1;
  ∀ (ca: Cust-Acct, a: Account) size(image(ca, a)) = 1;
 % new event definition
  new-cust-acct() = empty-set;
  ... definition of image operations ...
end-association
```

Figure 6.23 Cust-Acct Association

6.4.1 Multiplicity. For binary associations, there are five categories of association multiplicities: exactly one, many, optional, one or more, or numerically specified. Since multiplicities are based on the number of links of an association in which any given object may participate in, an *image* operation is defined for each class in the association. Basically, in a binary association, the image operation returns a set of objects with which a particular object is associated and is used to define multiplicity constraints as shown in Figure 6.24.

exactly one	$\mapsto$ size(image(a,o)) $= 1$
many	$\mapsto$ size(image(a,o)) ≥ 0
optional	$\mapsto$ size(image(a,o)) $= 1 \vee$ size(image(a,o)) $= 0$
one or more	$\mapsto$ size(image(a,o)) ≥ 1
numerically specified	$\mapsto$ size(image(a,o)) $= x$
numerically specified	$\mapsto$ size(image(a,o)) $\geq x \wedge$ size(image(a,o)) $\leq y$

Figure 6.24 Association Multiplicity Axioms

True ternary associations are relatively rare; however, they can be modeled using an association class. The only differences between binary and ternary associations are the number of object-valued attributes and the signature of the image operation. In a ternary association, the image operation returns a set of object tuples associated with a given object. Since the output is a set of tuples, the same multiplicity axioms shown in Figure 6.24 apply to ternary association as well.

6.4.2 Qualified Associations. Qualifiers are special attributes used to reduce the multiplicity of a binary association, generally from one-to-many to one-to-one. A qualifier distinguishes among a set, or class, of objects. For instance, a customer at a bank may own many accounts. This is a one-to-many association. However, if the *owns* association is modeled with an *account number* qualifier as shown in Figure 6.25, the association becomes one-to-one since each account has a unique account number.

In the theory-based object model, qualifiers are modeled as link attributes with a *qualified image* operation that selects associated objects based on an object and a qualifier. Again, the

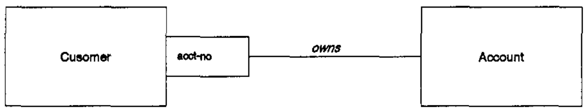

Figure 6.25 Association Qualifier

multiplicity axioms defined in Figure 6.24 can be used to restrict the qualified association using the qualified image operation.

An example of a qualified account association is shown in Figures 6.26 and 6.27. The *CA-Link* specification includes the qualifier *acct-no* as an link attribute. Therefore, to create a new link, a customer, account, and account number must be provided. The *Cust-Acct* association is modified by adding the *acct-no* qualifier to the customer image operation. Thus, as stated by the axiom *size(image(ca,c,n)) = 1*, the multiplicity of the association is changed from one-to-many to one-to-one.

```
link CA-LINK is
class sort CA-Link
sorts Customer, Account, Acct-No
operations
  attr-equal : CA-Link, CA-Link → Boolean
attributes
  customer : CA-Link → Customer
  account : CA-Link → Account
  acct-no : CA-Link → Acct-No
methods
  create-ca-link : Customer, Account → CA-Link
events
  new-ca-link : Customer, Account, Acct-No → CA-Link
axioms
  % operation definition
  ∀ (c,c1: Customer)
    attr-equal(c,c1) ⇔ customer(c) = customer(c1) ∧ account(c) = account(c1)
    acct-no(c) = acct-no(c1);
  % create method definition
  ∀ (c: Customer, a: Account, n: Acct-No)
    customer(create-ca-link(c,a,n)) = c ∧ account(create-ca-link(c,a,n)) = a
    acct-no(new-ca-link(c,a,n)) = n;
  % new event definition
  ∀ (c: Customer, a: Account)
    attr-equal(new-ca-link(c,a), create-ca-link(c,a))
end-link
```

Figure 6.26 Qualified Customer Account Link

```
association CUST-ACCT is
link-class CA-Link
class sort Cust-Acct
sorts Accounts, Customers
methods
  image : Cust-Acct, Customer, Acct-No → Accounts
  image : Cust-Acct, Customer → Accounts
  image : Cust-Acct, Account → Customers
events
  new-cust-acct : → Cust-Acct
axioms
 % multiplicity axioms
  ∀ (ca: Cust-Acct, c: Customer, an: Acct-No) size(image(ca,c,an)) = 1;
  ∀ (ca: Cust-Acct, a: Account) size(image(ca,a)) = 1;
 % new event definition
  new-cust-acct() = empty-set;
  ... definition of image operations ...
end-association
```

Figure 6.27 Qualified Cust-Acct Association

6.5 Aggregation

Aggregation is another concept central to object-orientation. Aggregation is a relationship between two classes where one class, the *aggregate*, represents the entire assembly and the other class, the *component*, is "part-of" the assembly. Aggregate class behavior is defined by its components and the associations and constraints between them. Without aggregate objects, a system composed of multiple subsystems cannot be modeled. Components may or may not exist apart from an aggregate and may be members of several aggregates. Aggregates may have fixed, variable, or a recursive structure (83:59). In a fixed aggregate, the type and number of components are always the same. For example, a car has one body, four wheels, one engine, etc. In a variable aggregate, the type of components in the aggregate are fixed but the number of components vary. For instance, in the banking example, a bank may consist of a number of employees, customers, and bank accounts. While each bank has employees, customers, and accounts, the number of each component varies with time and between banks. Finally, in the recursive aggregate, components may be defined as aggregates made up of additional components of the same type. For example, in a computer program, the program is made up of one or more program blocks. Program blocks consist of statements. Statements may be simple or complex, where complex statements consist of at least one program block.

Not only do aggregate classes allow the modeling of systems from components, but they also provide a convenient context in which to place constraints between components. For example, although the object-valued attributes in *CA-Link* are named *customer* and *account* (Figure 6.23), they are unified with the *CUSTOMER* and *ACCT* classes. Unification of these sorts with the appropriate class sorts requires a higher-level specification that describes how classes and associations interact. This higher-level specification is an aggregate class. Once again, object-valued attributes describe this relationship between aggregate classes and their components. I now formally define an aggregate using the colimit operation and object-valued attributes.

Definition 6.5.1 Aggregate - *A class C is an* aggregate *of a collection of component classes,* $(D_1..D_n)$*, if there exists a specification morphism from the colimit of* $(D_1..D_n)$ *to C such that C has at least one corresponding object-valued attribute for each class sort in* $(D_1..D_n)$.

An aggregate class combines a number of classes via the colimit operation to specify system or subsystem level functionality. The colimit operation also unifies sorts and operations defined in separate classes and associations.

To create system-level aggregates, the colimit of all classes and associations within the system is taken. In the bank account example, the *CUSTOMER*, *ACCT*, and *CUST-ACCT* classes are combined to form an aggregate system. To integrate the components into an aggregate, the sorts from *CUST* and *CUST-ACCT* and the sorts from *ACCT* and *CUST-ACCT* are unified via specification morphisms that define their equivalence as shown in Figure 6.28. The actual specification of the aggregation colimit is shown in Figures 6.29 and 6.30. The *SET* specification is used to unify sorts while the *INTEGER* specification (Appendix E) is included to ensure only a single copy of integers is included. Because each class imports the *SET* specification (Appendix E) which in turn imports the *INTEGER* specification, failure to include the *INTEGER* specification in the colimit would create a unique copy of *INTEGER* for each class in the aggregate. Three copies of the *SET* specification are included in the colimit operation since each class is defined as a unique set and cannot be unified with the other class sets or associations defined in the colimit.

$$\begin{tikzcd}
 & \text{Integer} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Set} \arrow[d, "{\{E \to \text{Acct}, \text{Set} \to \text{Acct-Class}\}}"'] \arrow[dr, "{\{E \to \text{Account}, \text{Set} \to \text{Accounts}\}}"] & \text{Set} \arrow[d, "{\{E \to \text{CA-Link}, \text{Set} \to \text{Cust-Acct}\}}"] & \text{Set} \arrow[dl, "{\{E \to \text{Customer}, \text{Set} \to \text{Customers}\}}"'] \arrow[d, "{\{E \to \text{Customer}, \text{Set} \to \text{Cust-Class}\}}"] \\
\text{Acct-Class} \arrow[dr, "c"'] & \text{Cust-Acct} \arrow[d, "c"] & \text{Cust-Class} \arrow[dl, "c"] \\
 & \text{Bank} &
\end{tikzcd}$$

Figure 6.28 Aggregation Composition

```
aggregate BANK-AGGREGATE is
    nodes INTEGER, SET-1: SET, SET-2: SET, SET-3: SET,
          ACCT-CLASS, CUST-CLASS, CUST-ACCT
    arcs  SET-1 → ACCT-CLASS : {E → Acct, SET → Acct-Class},
          SET-1 → CUST-ACCT : {E → Account, SET → Accounts},
          SET-2 → CUST-CLASS : {E → Customer, SET → Cust-Class},
          SET-2 → CUST-ACCT : {E → Customer, SET → Customers},
          SET-3 → CUST-ACCT : {E → CA-Link, SET → Cust-Acct},
          INTEGER → SET-1 : {},
          INTEGER → SET-2 : {}
          INTEGER → SET-3 : {}
end-aggregate
```

Figure 6.29 Aggregation Specification

Once the *BANK-AGGREGATE* specification is computed, the *CUST-ACCT* association actually associates the *CUSTOMER* class to the *ACCT* class. New operations and axioms can be

```
class BANK is
import BANK-AGGREGATE
class sort Bank
attributes
  acct-obj : Bank → Acct-Class
  cust-obj : Bank → Cust-Class
  cust-acct-assoc : Bank → Cust-Acct
methods
  aggregate methods defined here
events
  aggregate events defined here
axioms
  definition of aggregate methods in terms of components here
  size(a) ≥ 1;
  size(c) ≥ 1;
  size(ca) ≥ 1
end-class
```

Figure 6.30 Aggregate Specification

added to an extension of colimit specification, the *BANK* class type specification (Figure 6.30), to describe aggregate-level interfaces and aggregate behavior based on component events and methods.

6.5.1 Specification of Components. As stated above, components have either a fixed, variable, or recursive structure. All three structures use object-valued attributes to reference other objects and define the aggregate. The difference between them lies in the types of objects that are referenced and the operations and axioms defined over object-valued attributes. In a fixed configuration, once an aggregate references a particular object, that reference may not be changed. The ability of an aggregate object to change the object references of its object-valued attributes is determined by whether a method exists (other than the initialization method) to modify the object-valued attribute. If no methods modify any object-valued attributes then the aggregate is fixed. If methods do modify the object-valued attributes, then the aggregate is variable. An example of a fixed configuration aggregate is the *PRODUCER* class as defined in Figure 6.6. In this example, the component, *buffer*, is defined at initialization and cannot be changed. Although axioms defining the effect of the method *produce* on the attribute *buffer-obj* appears to modify the value of *buffer-obj*, they, in fact, do not. The axioms simply send events to the buffer object which modify its internal state. An example of a variable version of the *PRODUCER* class is shown in Figure 6.31. In this example, the method *change-buf* changes the object reference value of *buffer-obj.*

```
class PRODUCER is
imports Buffer, Item
class sort Producer
sorts Producer-State
operations
  attr-equal : Producer, Producer → Boolean
attributes
  buffer-obj : Producer → Buffer
methods
  create-producer : Buffer → Producer
  produce : Producer, Item → Producer
  change-buf : Producer, Buffer → Producer
events
  new-producer : Buffer → Producer
  produce-item : Producer, Item → Producer
  change-buffer : Producer, Buffer → Producer
axioms
 % operation definitions
 ∀ (p,p1: Producer) attr-equal(p, p1) ⇒ buffer-obj(p) = buffer-obj(p1);
 % event definitions
 ∀ (b: Buffer) attr-equal(new-producer(b), create-producer(b));
 ∀ (i: Item, p: Producer) attr-equal(produce-item(p,i), produce(p,i));
 ∀ (p: Producer, b: Buffer) attr-equal(change-buffer(p,b), change-buf(p,b));
 % method definitions
 ∀ (b: Buffer) buffer-obj(create-producer(b)) = b;
 ∀ (i: Item, p: Producer) buffer-obj(produce(p,i)) = put(buffer-obj(p),i);
 ∀ (b: Buffer, p: Producer) buffer-obj(change-buf(p,b)) = b
end-class
```

Figure 6.31 Object-Valued Attribute Example

A recursive structure is also easily represented using object-valued attributes. In this case, an object-valued attribute is defined in the class type that references its own class sort. For example, a machine may consist of a number of assemblies. Assemblies can be composed from individual parts and other subassemblies, which in turn can be composed of parts and subassemblies, etc. In this case, the machine class type has one object-valued attribute, *assembly-set-obj*, which references a set of assembly objects. The assembly class type definition has two object-valued attributes: a *parts-set-obj* which references a set of parts, and an *assembly-set-obj*, which references a set of assembly objects. There are no inconsistencies with a recursive aggregate as long as the references are not cyclic. Cyclic references are easily avoided via axiomatic specification as defined below for some class X.

$$\begin{array}{lr} \textbf{operations} & \\ \quad \textit{non-cyclic} : X, X \rightarrow \textit{boolean} & \\ \textbf{axioms} & \\ \quad \forall(x, y : X)\textit{non-cyclic}(x, x\textit{-obj}(x)); & (a) \\ \quad \forall(x, y : X)\textit{non-cyclic}(x, y) \Rightarrow x \neq y; & (b) \\ \quad \forall(x, y : X)\textit{non-cyclic}(x, y) \Rightarrow \textit{non-cyclic}(x, x\textit{-obj}(y)); & (c) \end{array} \tag{6.1}$$

These axioms define a *non-cyclic* operation which determines if a cycle exists in a self-referencing class type X. Axiom (a) states the invariant condition that all objects within class X must be non-cyclic over the self-referencing object-valued attribute *x-obj* while axioms (b) and (c) recursively define the operation over all objects within the aggregate.

6.5.2 Qualified Aggregates. Because aggregation is a form of association, qualifiers may be specified between aggregates and their components. Aggregate qualifiers are a special case of association qualifiers and are defined as special attributes used to reduce the multiplicity of an aggregation. Just as in associations, a qualifier is used to distinguish among a set, or class, of objects. For instance, a bank may have many customers; however, if the aggregation is modeled with a customer number qualifier as shown in Figure 6.32, the aggregation becomes one-to-one since each customer has a unique customer number.

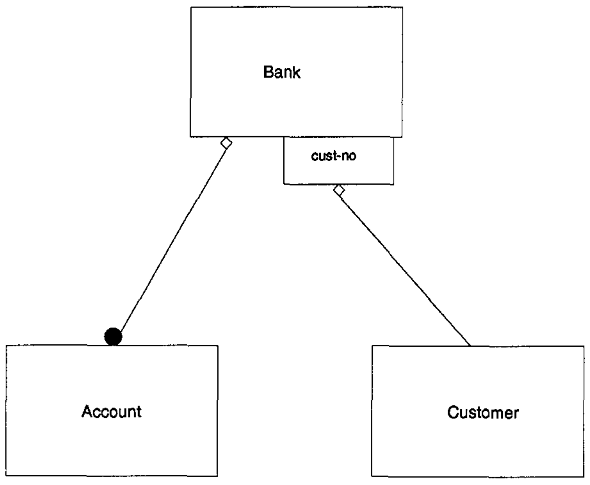

Figure 6.32 Aggregate Qualifier

In the theory-based object model, qualifiers are modeled as an attribute of the qualified class with a *qualified image* operation defined in its class set. This image operation is similar to those for qualified associations that select components from a class set based on the qualifier. Again, the multiplicity axioms of Figure 6.24 are used to restrict the qualified aggregate using the qualified image operation.

An example of a qualified bank – customer aggregation is shown in Figures 6.33 and 6.34. The *CUSTOMER* class type definition includes the qualifier *cust-no* as a normal attribute. Therefore, to create a new customer, a name, address, and customer number must be provided. The *Cust-Class* class set is modified by adding an image operation with the *cust-no* qualifier. The *update-customer* class event is used in conjunction with the *image* operation to perform the *update-customer* event on a single customer as designated via the *cust-no* (e.g., *update-customer(image(customer-class, cust-no), new-name, new-address)*). Thus, as stated by the axiom $size(image(cc, n)) = 1$, the aggregate multiplicity is changed from one-to-many to one-to-one.

6.5.3 Specification of Behavior. Once an aggregate is created via a colimit operation, further specification is required to make the aggregate behave in an integrated manner. First, new aggregate level functions are defined to enable the aggregate to respond to external events. Then, constraints between aggregate components are specified to ensure that the aggregates do not behave in an unsuitable or unexpected manner, and finally, local event communication paths are defined. The definition of new functions and constraints is discussed in this section while communication between objects is discussed in Section 6.6.

6.5.3.1 Specification of Functionality. In an aggregate, components work together to provide the desired functionality. This desire to define functionality across components leads naturally to the use of the functional model. The functional model is used to specify the results of a computation without defining where or how they are computed and is used to define actions gen-

class CUSTOMER **is**
import Name, Address, Cust-No
class sort Customer
operations
  attr-equal : Customer, Customer $\rightarrow$ Boolean
attributes
  name : Customer $\rightarrow$ Name
  address : Customer $\rightarrow$ Address
  cust-no : Customer $\rightarrow$ Cust-No
methods
  create-customer : Name, Address, Cust-No $\rightarrow$ Customer
  update : Customer, Name, Address, Cust-No $\rightarrow$ Customer
events
  new-customer : Name, Address, Cust-No $\rightarrow$ Customer
  update-customer : Customer, Name, Address $\rightarrow$ Customer
axioms
  % *operation definition*
  $\forall$ (c,c1: Customer) attr-equal(c,c1) $\Leftrightarrow$ name(c) = name(c1)
    $\wedge$ address(c) = address(c1) $\wedge$ cust-no(c) = cust-no(c1);
  % *create method definition*
  $\forall$ (n: Name, a: Address, cn: Cust-No)
    name(create-customer(n,a,cn)) = n $\wedge$ address(create-customer(n,a,cn)) = a
    $\wedge$ cust-no(create-customer(n,a,cn)) = cn;
  % *update method definition*
  $\forall$ (c: Customer, n: Name, a: Address, cn: Cust-No)
    name(update(c,n,a,cn)) = n $\wedge$ address(update(c,n,a,cn)) = a
    $\wedge$ cust-no(update(c,n,a,cn)) = cn;
  % *new event definition*
  $\forall$ (n: Name, a: Address, cn: Cust-No)
    attr-equal(new-customer(n,a,cn), create-customer(n,a,cn));
  % *update-customer event definition*
  $\forall$ (c: Customer, n: Name, a: Address, cn: Cust-No)
    attr-equal(update-customer(c,n,a,cn), update(c,n,a,cn))
end-class

Figure 6.33 Qualified Customer Class

```
class CUST-CLASS is
contained-class CUST
class sort Cust-Class
operations
  attr-equal: Cust-Class, Cust-Class → Boolean
methods
  image : Cust-Class, Cust-No → Cust-Class
events
  new-cust-class : → Acct-Class
  update-customer : Cust-Class, Name, Address → Cust-Class
axioms ∀ (c: Cust, n: Name, a: address, cc: Cust-Class)
  ∀ (cc: Cust-Class, n: Name) size(image(cc,n)) = 1;
  new-cust-class = empty-set;
  ∀ (c: Cust, cc: Cust-Class, n: Name, a: Address)
    c ∈ cc ⇔ update-customer(c,n,a) ∈ update-customer(cc,n,a);
  ... definition of image operations ...
end-class
```

Figure 6.34 Qualified Customer Class Set

erated by the dynamic model (83:123). Processes defined in the functional model are implemented using events and attributes defined in the aggregate components.

An example of defining the functional behavior of an aggregate using the functional model is shown in Figure 6.35 with the full specification shown in Figure 6.36. The *Bank* aggregate actually defines three new events (*start-account*, *make-deposit*, and *make-withdrawal*) and a derived attribute *balance*. The functional model defines the method implementing the *start-account* event, *add-account* as shown in Figure 6.35. The *make-deposit* and *make-withdrawal* events map directly to component events and do not require a functional decomposition.

Figure 6.35 shows the functional model for the *add-account* action. The left-hand side is the top-level diagram while the right-hand side shows the decomposition of *add-account*. The *add-account* process adds an account for an established customer. The following axiom defines *add-account* in terms of its subprocesses and data flows and is translated directly from the functional model using the restricted functional model semantics described in Section 5.5.

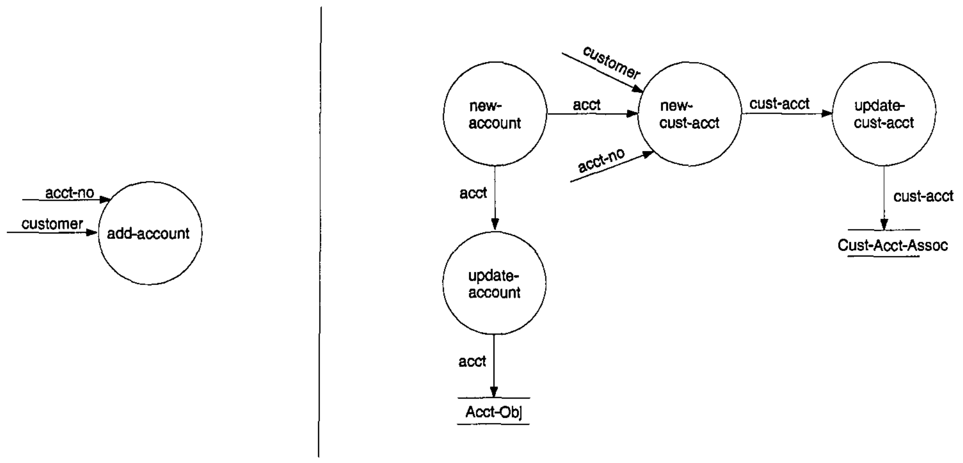

Figure 6.35 Bank Aggregate Functional Model

add-account$(b, customer, acct\text{-}no) = b1$
$\wedge$ *acct* = *new-acct*(*date*) %% assume date is built in
$\wedge$ *acct-obj*$(b1)$ = *update-accts*(*acct-obj*(b), *acct*)
$\wedge$ *cust-acct* = *new-cust-acct*(*customer*, *acct*, *acct-no*)
$\wedge$ *cust-acct*$(b1)$ = *update-cust-acct*(*cust-acct*(b), *cust-acct*);

The *add-account* method has three parameters, the bank object (b) plus an account number and an existing customer object as defined in the functional model, and returns the modified bank object ($b1$). The *add-account* method is defined by its subprocesses. First, a new account is created by invoking the *new-account* event which is then passed to the *update-accts* process which stores the new account in the account class. Then, the new account is passed as a parameter to the *new-cust-acct* event which returns a cust-acct link which relates the customer, the account number, and the new account. Finally, the new cust-acct link is passed to the update-cust-acct process which stores it in the cust-acct association. The subprocesses in Figure 6.35 are not defined here. The *new-account* and *new-cust-acct* processes are the events defined in the account class and cust-acct association respectively and are already available via the aggregate. The *update-accts* and *update-cust-acct* processes either already exist as part of the account class and cust-acct association or may be defined in this specification.

```
class BANK is
import BANK-AGGREGATE
class sort Bank
operations
    balance : Bank, Acct-No → Amnt
    attr-equal : Bank, Bank → Boolean
attributes
    acct-obj : Bank → Acct-Class
    cust-obj : Bank → Cust-Class
    cust-acct-assoc : Bank → Cust-Acct
methods
    create-bank : → Bank
    add-account : Bank, Customer, Acct-No → Bank
    update-accts : Acct-Class, Acct → Acct-Class
    update-cust-acct : Cust-Acct, Cust-Acct-Link → Cust-Acct
events
    new-bank : → Bank
    start-account : Bank, Customer, Acct-No → Bank
    make-deposit, make-withdrawal : Bank, Acct-No, Amnt → Bank
axioms  % invariants
    ∀ (a: Acct-Class, c: Cust-Class) size(a) ≥ 0 ∧ size(c) ≥ 0;
    ∀ (c: Cust-Acct-Class, b: Bank, a: Acct-No) size(c) ≥ 0 ∧ balance(b,a) ≥ 0;
  % definition of operations
    ∀ (b,b1: Bank) attr-equal(b,b1) ⇔ acct-obj(b) = acct-obj(b1)
      ∧ cust-obj(b) = cust-obj(b1) ∧ cust-acct-assoc(b) = cust-acct-assoc(b1);
    ∀ (b: Bank, a: Address, an: Acct-No, c: Customer) size(image(acct-obj(b),c,an)) = 1
      ⇒ singleton(a) = image(acct-obj(b),c,an) ∧ balance(b,an) = bal(a);
  % definition of methods
    acct-obj(create-bank()) = create-acct-class();
    cust-obj(create-bank()) = create-cust-class();
    cust-acct-assoc(create-bank()) = create-cust-acct();
    ∀ (b, b1: Bank, an: Acct-No, c: Customer)
      add-account(b, c, an) = b1
        ∧ acct = new-acct(date) %% date built in
        ∧ acct-obj(b1) = update-accts(acct-obj(b), acct)
        ∧ cust-acct = new-cust-acct(cust, acct, acct-no)
        ∧ cust-acct(b1) = update-cust-acct(cust-acct(b), cust-acct);
  % definition of events
    attr-equal(new-bank(), create-bank());
    ∀ (b:Bank, an:Acct-No, c:Customer) attr-equal(start-account(b,c,an),add-account(b,c,an));
    ∀ (b: Bank, an: Acct-No, c: Customer, am: Amount) size(image(acct-obj(b),c,an)) = 1
      ⇒ attr-equal(image(cust-acct-assoc(make-deposit(b,an,am),c,an)),
        deposit(image(cust-acct-assoc(b),c,an),am));
    ∀ (b: Bank, an: Acct-No, c: Customer, am: Amount)
      size(image(acct-obj(b),c,an)) = 1 ∧ balance(b,an) ≥ x
      ⇒ attr-equal(image(cust-acct-assoc(make-withdrawal(b,an,am))),
        withdrawal(image(cust-acct-assoc(b),c,an),am))
end-class
```

Figure 6.36 Full Aggregate Specification

Both the *make-deposit* and *make-withdrawal* events mirror events defined in the *ACCT* class and thus invoke those events directly. However, because the bank aggregate has the requirement to ensure an account does not overdraw its available cash, a precondition is placed on the *make-withdrawal* event.

$$\begin{aligned}&size(image(acct\text{-}obj(b), c, an)) = 1 \wedge balance(b, an) \geq x\\&\quad\Rightarrow attr\text{-}equal(image(cust\text{-}acct\text{-}assoc(make\text{-}withdrawal(b, an, am))),\\&\qquad withdrawal(image(cust\text{-}acct\text{-}assoc(b), c, an), am))\end{aligned}$$

Because the *ACCT* class allowed an account to be overdrawn exactly once, a precondition overriding the *ACCT withdrawal* precondition was added. This is a case of *restricting* the behavior of the aggregate.

The derived attribute *balance* is different from the attributes defined in Section 6.2 in that it takes additional parameters. The additional parameter is due to the fact that *balance* is not an attribute of a bank, but of an account. The parameter *acct-no* is required to uniquely identify a specific account. The axiom below defines the *balance* attribute.

$$\begin{aligned}&size(image(acct\text{-}obj(b), c, an)) = 1\\&\quad\Rightarrow singleton(a) = image(acct\text{-}obj(b), c, an) \wedge balance(b, an) = bal(a)\end{aligned}$$

Because the *image* function returns a set of accounts, a few axiomatic gymnastics are required to define the operation. While this additional complexity seems unnecessary, use of sets with object-valued attributes provides the most flexible approach to building domain models and can be simplified in the functional specification generated by *Specification Generation/Refinement System* as defined in Chapter II.

6.5.3.2 Specification of Constraints Between Components. In an aggregate, component behavior must often be constrained if the aggregate is to act in an integrated fashion. For instance, in an automobile there is an engine, transmission, and four wheels; however, they do not act independently. When the engine is running and the transmission is engaged, there is a exact relationship that exists between the engine speed, transmission state, and the wheel rotation speed. This relationship is a *constraint* between the automobile components. Generally, these relationships are expressed by axioms defined over component attributes. Because the aggregate is the colimit of its components, the aggregate may access components directly and define axioms relating various component attributes.

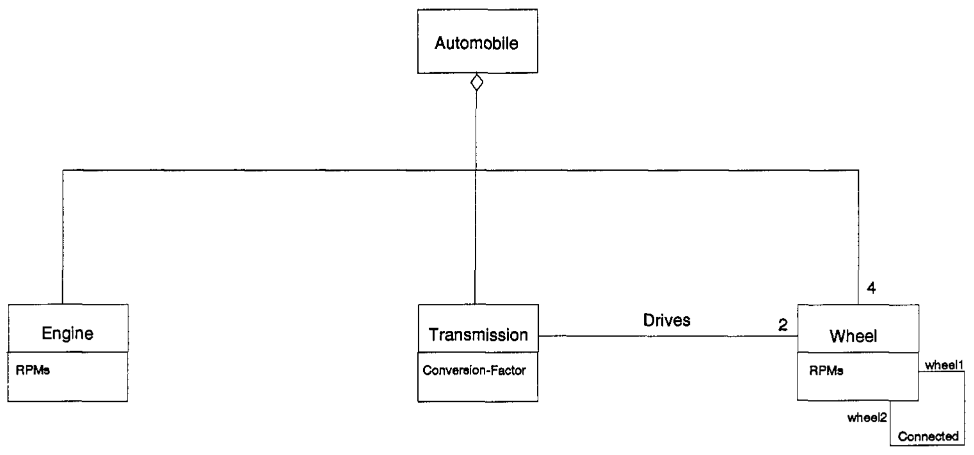

Figure 6.37 Automobile Aggregate Functional Model

A simplified automobile object model is shown in Figure 6.37. The object model contains one engine with an *RPMs* attribute, one transmission with a *Conversion-Factor* attribute, and four wheels, each with an *RPMs* attribute. Two relationships exist between these objects, *Drives*, that relates the transmission to exactly two wheels, and *Connected* that relates two wheels (probably by an axle). Obviously, there are a number of constraints implicit in the object model that must be made explicit in the aggregate. First, as discussed above, the *RPMs* of the engine, *Conversion-Factor* of the transmission, and *RPMs* of the wheels are all related. Also, the wheels driven by the transmission must be "connected", and all "connected" wheels should have the same RPMs. These constraints can be specified in the aggregate specification shown in Figure 6.38. The axiom

$$\begin{aligned}
&\forall(e : Engine, t : Transmission, d : Drives)\\
&\quad e \in \mathit{engine\text{-}obj}(a) \wedge t \in \mathit{transmission\text{-}obj}(a) \wedge d \in \mathit{drives\text{-}assoc}(a)\\
&\Rightarrow rpm(\mathit{wheel\text{-}obj}(d)) = rpm(e) * \mathit{conversion\text{-}factor}(t)
\end{aligned}$$

```
class AUTOMOBILE is
import AUTOMOBILE-AGGREGATE
class sort Automobile
operations
  attr-equal : Automobile, Automobile → Boolean
attributes
  engine-obj : Automobile → Engine-Class
  transmission-obj : Automobile → Transmission-Class
  wheels-obj : Automobile → Wheels-Class
  drives-assoc : Automobile → Drives
  connected-assoc : Automobile → Connected
methods
  create-automobile : → Automobile
events
  new-automobile : → Automobile
axioms
% invariants
  ∀ (ec: engine-class) size(ec) = 1;
  ∀ (tc: transmission-class) size(tc) = 1;
  ∀ (wc: wheels-obj) size(wc) = 4;
  ∀ (d: drives-assoc) size(d) = 2;
  ∀ (c: connected-assoc) size(c) = 2;
% constraints
  ∀ (e: Engine, t: Transmission, d: Drives)
    e ∈ engine-obj(a) ∧ t ∈ transmission-obj(a) ∧ d ∈ drives-assoc(a)
    ⇒ rpm(wheel-obj(d)) = rpm(e) * conversion-factor(t);
  ∀ (c: Connected) c ∈ connected-assoc(a)) ⇒ rpm(wheel1(c)) = rpm(wheel2(c));
% definition of attr-equal
  ∀ (a,a1: Automobile)
    attr-equal(a,a1) ⇒ engine-obj(a) = engine-obj(a1)
    ∧ transmission-obj(a) = transmission-obj(a1)
    ∧ wheels-obj(a) = wheels-obj(a1)
    ∧ drives-assoc(a) = drives-assoc(a1)
    ∧ connected-assoc(a) = connected-assoc(a1);
% definition of create-automobile
    t = new-transmission()
    ∧ w1 = new-wheel() ∧ w2 = new-wheel() ∧ w3 = new-wheel() ∧ w4 = new-wheel()
    ∧ transmission-obj(create-automobile()) = insert(t, new-transmission-class())
    ∧ drives-assoc(create-automobile()) =
      insert(new-drives-link(t,w2),insert(new-drives-link(t,w1),new-drives()))
    ∧ connected-assoc(create-automobile()) =
      insert(new-connected-link(w1,w2),insert(new-connected-link(w3,w4),new-connected()))
    ∧ engine-obj(create-automobile()) = insert(new-engine(), new-engine-class());
% definition of new-automobile
  attr-equal(new-automobile(), create-automobile())
end-class
```

Figure 6.38 Automobile Aggregate Specification

defines the relationship between the *RPMs* of the wheels driven by the transmission, the transmission *conversion-factor* and the engine *RPMs.* While written in set notation, the invariants state that the size of the engine and transmission class sets is only one; therefore, the axiom uniquely identifies the engine, transmission, and each wheel driven by the transmission. The axiom

$$\forall(c : Connected) c \in connected\text{-}assoc(a)) \Rightarrow rpm(wheel1(c)) = rpm(wheel2(c));$$

ensures that the two wheels connected in a "connected" link have the same *RPMs* values. The final constraint, that the two wheels driven by the transmission be connected, is specified implicitly in the specification of the *create-automobile* method. Because the *create-automobile* method creates its components when invoked, the relationships of the wheels can be controlled directly. After the wheel objects ($w1$, $w2$, $w3$, and $w4$) are created, links are created for, and inserted into, the *drives* and *connected* associations as defined below.

$$\begin{aligned}
&\wedge\ drives\text{-}assoc(create\text{-}automobile()) = \\
&\quad insert(new\text{-}drives\text{-}link(t, w2), insert(new\text{-}drives\text{-}link(t, w1), new\text{-}drives())) \\
&\wedge\ connected\text{-}assoc(create\text{-}automobile()) = \\
&\quad insert(new\text{-}connected\text{-}link(w1, w2), insert(new\text{-}connected\text{-}link(w3, w4), new\text{-}connected()))
\end{aligned}$$

Because wheels $w1$ and $w2$ are associated with the transmission via the *drives* association in the first line, they are also associated together via the *connected* association in the second line. Thus, the constraint is satisfied whenever an automobile aggregate object is created.

6.6 Communication

At this point the theory-based object model is sufficient for describing classes, their relationships, and their composition into aggregate classes; however, object communication has not yet been addressed. For example, suppose the banking system described earlier has an *ARCHIVE* object which logs each transaction as it occurs. Obviously, the *ARCHIVE* object must be told when a transaction takes place. This communication is accomplished in one of three ways. The simplest method is to force the *BANK* aggregate to be responsible for directly invoking events in each object to accomplish the archival function and passing the appropriate information to each object. While simple in this example, as aggregate complexity and object interaction increases, an enormous burden is placed on the aggregate.

A second solution is to make each object responsible for handling its own communications. Each object directly communicates with other objects by maintaining internal object-valued attributes and invoking their events directly. Unfortunately, if each object is required to know each object with which it may communicate, reusability is lost since those objects may not exist in a different system.

The third, and preferred, method is a combination of the above techniques. In this method, each object is aware of only a certain set of *events* that it generates or receives. From an object's perspective, these events are generated and broadcast to the entire system and received from the system. In this technique, each event is defined in a separate event theory as shown in Figure 6.39.

```
event ARCHIVE-WITHDRAWAL is
class sort Archive
sorts Acct, Amnt
events
  archive-withdrawal : Archive, Acct, Amnt → Archive
end-class
```

Figure 6.39 Event Theory

An *event theory* consists of a class sort, parameter sorts, and an event signature that are mapped via signature morphisms to sorts and events in the generating and receiving classes. The event theory class sort represents the class sort of any class whose objects can receive the associated event. Because events are actually sent to individual objects represented by object-valued attributes as defined in Section 6.2.9, an event may only be sent to one object in the event theory class sort. Therefore, if the event theory class sort is mapped to the class sort of class X then communication occurs with a single object from class X. However, if the event theory class sort is mapped to the class sort of the *class set* of type X (i.e., X-CLASS), then communication may occur with a set of objects of class X. The other sorts defined in an event theory class are the sorts of other parameters of the event. The final part of an event theory is the event signature itself. This signature is mapped to an event in the receiving classes with compatible parameters as defined in the event theory. Once the event and sorts are mapped to the required class specifications under signature morphisms, the colimit of the classes, the event theory, and the morphisms unify the event and sorts such that any invocation of the event in the generating class is an invocation of the actual event in the receiving class.

Figure 6.40 shows how an event theory would be incorporated into the original *ACCT* class. The *ARCHIVE-WITHDRAWAL* event theory specification is imported into the *ACCT* class and an object-valued attribute, *archive-obj*, is added to reference the archival object. The axioms defining the effect of the withdrawal event are modified to reflect the communication with the *ARCHIVE* object as shown below.

$$\begin{aligned}
&\forall(a : Acct, x : Amnt)\mathit{acct\text{-}state}(a) = ok \wedge bal(a) \geq x \Rightarrow \mathit{acct\text{-}state}(\mathit{withdrawal}(a,x)) = ok\\
&\quad \wedge\ \mathit{archive\text{-}obj}(\mathit{withdrawal}(a,x)) = \mathit{archive\text{-}withdrawal}(\mathit{archive\text{-}obj}(a), a, x)\\
&\quad \wedge\ \mathit{attr\text{-}equal}(\mathit{withdrawal}(a,x), \mathit{debit}(a,x));\\
&\forall(a : Acct, x : Amnt)\mathit{acct\text{-}state}(a) = ok \wedge bal(a) < x \Rightarrow \mathit{acct\text{-}state}(\mathit{withdrawal}(a,x)) = \mathit{overdrawn}\\
&\quad \wedge\ \mathit{archive\text{-}obj}(\mathit{withdrawal}(a,x)) = \mathit{archive\text{-}withdrawal}(\mathit{archive\text{-}obj}(a), a, x)\\
&\quad \wedge\ \mathit{attr\text{-}equal}(\mathit{withdrawal}(a,x), \mathit{debit}(a,x));
\end{aligned}$$

Basically, the axioms state that when a *withdrawal* event is received, the value of the *archive-obj* is modified by the *archive-withdrawal* event defined in the event theory specification. Thus the *ACCT* object knows it communicates with some other object or objects; however, it does not know who they are. With whom an object communicates (or, for that matter, if the object communicates at all) is determined at the aggregate-level where the actual connections between communicating components are made. In this example, for instance, there could be zero, one, or many archival objects.

Figure 6.41 shows a modified *BANK* aggregate that includes the *ARCHIVE-WITHDRAWAL* event theory and an *ARCHIVE-CLASS* specification. The colimit operation includes morphisms from *ARCHIVE-WITHDRAWAL* to *ACCT-CLASS* and *ARCHIVE-CLASS* that unify the sorts

```
class ACCT is
import Amnt, Date, Archive-Withdrawal
class sort Acct
sorts Acct-State
operations
  attr-equal : Acct, Acct → Boolean
attributes
  date : Acct → Date
  bal : Acct → Amnt
  archive-obj : Acct → Archive
state-attributes
  acct-state : Acct → Acct-State
methods
  create-acct : Date, Archive → Acct
  credit, debit : Acct, Amnt → Acct
states
  ok, overdrawn : → Acct-State
events
  new-acct : Date, Archive → Acct
  deposit, withdrawal : Acct, Amnt → Acct
axioms
  ok ≠ overdrawn;
  ∀ (a: Acct) acct-state(a) = ok ⇒ bal(a) ≥ 0;
  ∀ (a: Acct) acct-state(a) = overdrawn ⇒ bal(a) < 0;
 % operation definitions
  ∀ (a,a1: Acct) attr-equal(a,a1)
    ⇒ date(a)=date(a1) ∧ bal(a)=bal(a1) ∧ archive-obj(a)=archive-obj(a1);
 % method definitions
  ∀ (d: Date, o: Archive) date(create-acct(d,o)) = d ∧ bal(create-acct(d,o)) = 0
    ∧ archive-obj(create-acct(d,o)) = o;
  ∀ (a: Acct, x: Amnt) bal(credit(a,x)) = bal(a) + x ∧ date(credit(a,x)) = date(a)
    ∧ rate(credit(a,x)) = rate(a) ∧ int-date(credit(a,x)) = int-date(a)
    ∧ check-cost(credit(a,x)) = check-cost(a);
 % event definitions
  ∀ (d: Date) acct-state(new-acct(d,o)) = ok ∧ attr-equal(new-acct(d,o), create-acct(d,o));
  ∀ (a: Acct, x: Amnt) acct-state(a) = ok ⇒ acct-state(deposit(a,x)) = ok
    ∧ attr-equal(deposit(a,x), credit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a) = overdrawn ∧ bal(a) + x ≥ 0 ⇒ acct-state(deposit(a,x)) = ok
    ∧ attr-equal(deposit(a,x), credit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a) = overdrawn ∧ bal(a) + x < 0
    ⇒ acct-state(deposit(a,x)) = overdrawn ∧ attr-equal(deposit(a,x), credit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a) = ok ∧ bal(a) ≥ x ⇒ acct-state(withdrawal(a,x)) = ok
    ∧ attr-equal(withdrawal(a,x), debit(a,x))
    ∧ archive-obj(withdrawal(a,x)) = archive-withdrawal(archive-obj(a),a,x);
  ∀ (a: Acct, x: Amnt) acct-state(a) = ok ∧ bal(a) < x ⇒ acct-state(withdrawal(a,x)) = overdrawn
    ∧ attr-equal(withdrawal(a,x), debit(a,x))
    ∧ archive-obj(withdrawal(a,x)) = archive-withdrawal(archive-obj(a),a,x);
  ∀ (a: Acct, x: Amnt) acct-state(a) = overdrawn ⇒ acct-state(withdrawal(a,x)) = overdrawn
    ∧ attr-equal(withdrawal(a,x), a)
end-class
```

Figure 6.40 Account Class with Communications

and event signature in *ACCT-CLASS* with the sorts and event signature of *ARCHIVE-CLASS.* This unification creates the communication path between account objects and archive objects. When an account object invokes the *archive-withdrawal* event, it is actually invoking the *archive-withdrawal* event of the archive class object. The simplified *BANK* composition diagram is shown in Figure 6.42. The *INTEGER* and *SET* specifications and the associated morphisms shown in Figure 6.28 are left out for simplicity, but still apply.

```
aggregate BANK-AGGREGATE is
    nodes INTEGER, SET-1: SET, SET-2: SET, ACCT-CLASS,
          CUST-CLASS, CUST-ACCT, ARCHIVE-WITHDRAWAL, ARCHIVE-CLASS
    arcs  SET-1 → ACCT-CLASS : {E → Acct, SET → Acct-Class},
          SET-1 → CUST-ACCT : {E → Account, SET → Acct-Class},
          SET-2 → CUST-CLASS : {E → Customer, SET → Cust-Class},
          SET-2 → CUST-ACCT : {E → Customer, SET → Customers},
          SET-3 → CUST-ACCT : {E → CA-Link, SET → Cust-Acct},
          INTEGER → SET-1 : {},
          INTEGER → SET-2 : {},
          INTEGER → SET-3 : {},
          INTEGER → ARCHIVE-CLASS : {},
          ARCHIVE-WITHDRAWAL → ACCT-CLASS: {},
          ARCHIVE-WITHDRAWAL → ARCHIVE-CLASS: {}
end-aggregate
```

Figure 6.41 Communicating Bank Aggregate Class

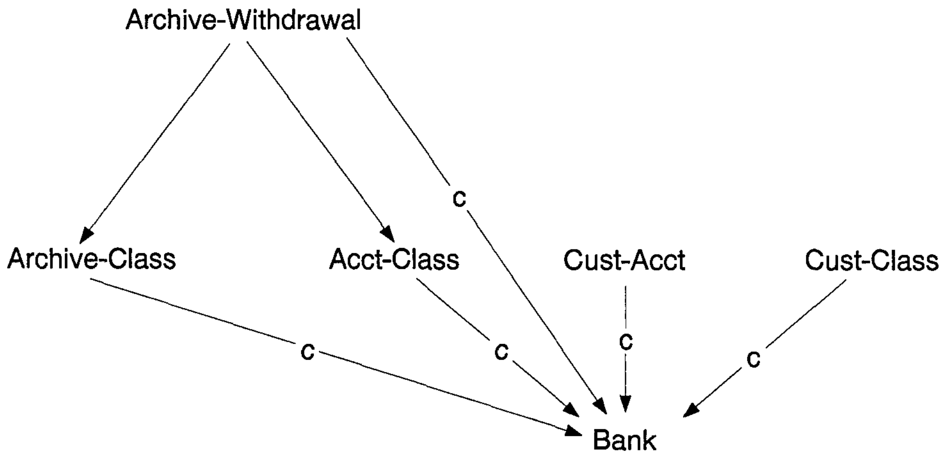

Figure 6.42 Bank Aggregate with Archive

Communicating with objects from multiple classes requires the addition of another level of specification which "broadcasts" the communication event to all interested object classes. The class sort of a *broadcast theory* is called a broadcast sort and represents the object with which the sending object communicates. The broadcast theory then defines an object-valued attribute for each receiving class. Figure 6.43 shows an example of the *ARCHIVE-WITHDRAWAL-MULT* event theory modified to communicate with two classes. In this case, the *ARCHIVE-WITHDRAWAL* theory is used to unify the *ARCHIVE-WITHDRAWAL-MULT* with the *ACCOUNT* class as well as the other two classes. An example of the required colimit specification is shown in Figure 6.44 while the diagram of the specification (showing only the morphisms between event and broadcast theories) is shown in Figure 6.45.

```
event ARCHIVE-WITHDRAWAL-MULT is
class sort Archive
sorts Amnt, Acct, X, Y
attribute
  x-obj : Archive → X
  y-obj : Archive → Y
events
  archive-withdrawal : Archive , Acct, Amnt → Archive
  archive-withdrawal : X, Acct, Amnt → X
  archive-withdrawal : Y, Acct, Amnt → Y
axioms
  ∀ (a: Archive, ac: Acct, am: Amnt)
    x-obj(archive-withdrawal(a,ac,am)) = archive-withdrawal(x-obj(a),ac,am)
    ∧ y-obj(archive-withdrawal(a,ac,am)) = archive-withdrawal(y-obj(a),ac,am)
end-class
```

Figure 6.43 Broadcast Theory

Multiple receiver classes add a layer of specification; however, multiple sending classes is handled very simply. The only additional construct required is a morphism from each sending class to the event theory mapping the appropriate object-valued attribute in the sending class to the class sort of the event theory and the event signature in the sending class to the event signature in the event theory. A diagram showing the effect of a second class sending the same *archive-withdrawal* event is shown in Figure 6.46.

A question requiring further research is how to specify exactly which objects in communicating classes actually communicate. In the banking example using a single archival object, the problem is straightforward. Send the events to the only instance of the class. Determining which classes

```
aggregate BANK-AGGREGATE is
    nodes INTEGER, SET-1: SET, SET-2: SET, ACCT-CLASS, CUST-CLASS,
          ARCHIVE-WITHDRAWAL-1: ARCHIVE-WITHDRAWAL, CUST-ACCT,
          ARCHIVE-WITHDRAWAL-2: ARCHIVE-WITHDRAWAL, ARCHIVE-CLASS,
          ARCHIVE-WITHDRAWAL-3: ARCHIVE-WITHDRAWAL, PRINTER-CLASS
    arcs  SET-1 → ACCT-CLASS : {E → Acct, SET → Acct-Class},
          SET-1 → CUST-ACCT : {E → Account, SET → Acct-Class},
          SET-2 → CUST-CLASS : {E → Customer, SET → Cust-Class},
          SET-2 → CUST-ACCT : {E → Customer, SET → Customers},
          SET-3 → CUST-ACCT : {E → CA-Link, SET → Cust-Acct},
          INTEGER → SET-1 : {},
          INTEGER → SET-2 : {},
          INTEGER → SET-3 : {},
          INTEGER → ARCHIVE-CLASS : {},
          ARCHIVE-WITHDRAWAL-1 → ACCT-CLASS: {},
          ARCHIVE-WITHDRAWAL-1 → ARCHIVE-WITHDRAWAL-MULT: {}
          ARCHIVE-WITHDRAWAL-2 → ARCHIVE-CLASS: {},
          ARCHIVE-WITHDRAWAL-2 → ARCHIVE-WITHDRAWAL-MULT: {Archive → X},
          ARCHIVE-WITHDRAWAL-3 → PRINTER-CLASS: {},
          ARCHIVE-WITHDRAWAL-3 → ARCHIVE-WITHDRAWAL-MULT: {Archive → Y}
end-aggregate
```

Figure 6.44 Unification of Multiple Broadcast Classes

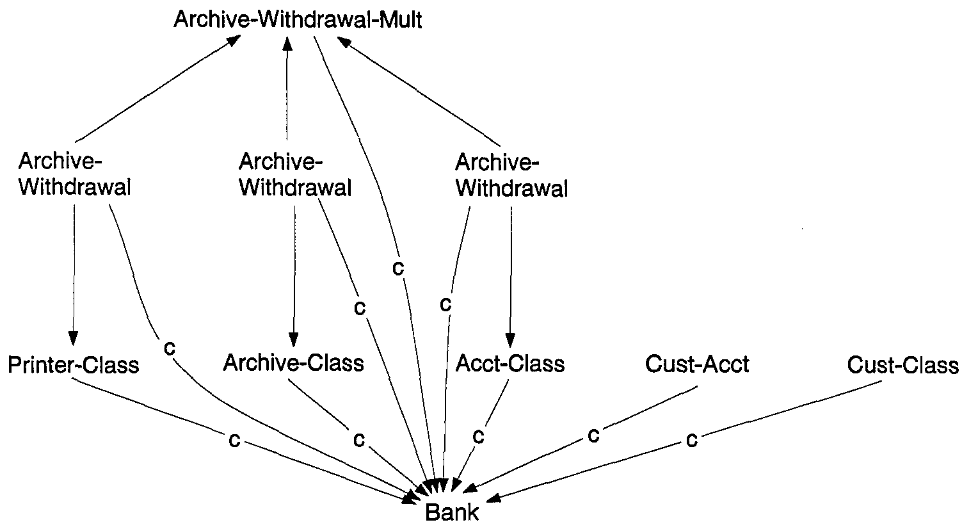

Figure 6.45 Aggregate Using a Broadcast Theory

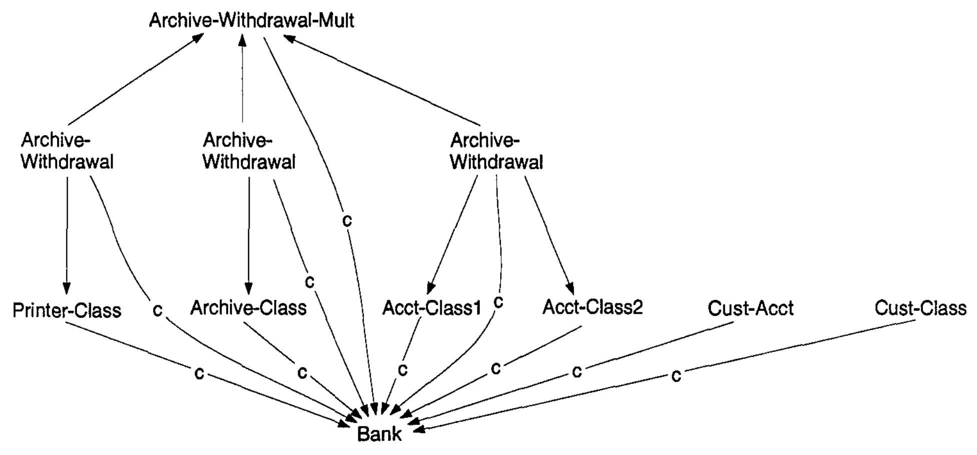

Figure 6.46 Aggregate Using a Broadcast Theory With Multiple Generators

communicate is much simpler and is defined in the dynamic model based on event name equivalence. If there is only one instance of a receiving class, then this model completely describes which objects communicate; all instances of the account class communicate with the archive object and the printer object. However, if certain accounts must send their archive events to certain archive or printer objects, then the model breaks down. An extension to the OMT model is required; however, that extension is not defined in this research. Because the theory-based object model is used for domain modeling, describing what classes may communicate is acceptable. Determining exactly which objects communicate is dealt with in the *System Generation/Refinement Subsystem* as defined in Chapter II.

6.6.1 Communication Between Aggregate and Components. Communication between components is handled at the aggregate level as described above. However, when the communication is between the aggregate and one of its components, the unification of object-valued attributes and class sorts via event theories does not work. Consider the example where a component sends an event that is received by its aggregate. An event theory can be created; however, because the class sort of the aggregate is not created until after the colimit is computed, the object-valued attribute from the event theory and the aggregate class sort cannot be unified in the colimit operation. There are two solutions to this problem.

The first, and simplest solution is to perform the unification at the next higher level aggregate or domain-level specification. This is the solution implemented in the translation defined in Chapter VII. Since each class is a component of an overall domain model class, the unification is performed at that level.

The second solution requires the use of a *sort axiom* that equivalences two or more sorts as shown below:

sort-axiom sort1 = sort2

Using the automobile example discussed above, assume the Engine generates an *engine-warning* event that is received by the Automobile aggregate. The event theory for such an event is shown in Figure 6.47. This event theory is included into the Engine class type definition and, by the colimit operation, the Automobile aggregate. To enable the Automobile aggregate to receive the engine-warning event, it uses a sort-axiom to equivalence the *Automobile* sort of the aggregate with the *Controller* sort from the event theory as shown in Figure 6.48. Use of the sort axiom unifies the *Automobile* sort and the *Controller* sort and thus the signatures of the *engine-warning* events defined in the event theory and the Automobile aggregate are equivalent.

```
event ENGINE-WARNING is
class sort Controller
events
  engine-warning : Controller, Integer → Controller
end-class
```

Figure 6.47 Engine-Warning Event Theory

Communications from the aggregate to the components, or subcomponents, is much simpler. Since the aggregate includes all the sorts, operations, and axioms of all of its components and subcomponents via the colimit operations, it can directly reference those components by the object-valued attributes declared either in itself (in the case of components) or in its components (for subcomponents). Because an aggregate is aware of its configuration, determining the correct object-valued attribute to use is not a problem.

```
class AUTOMOBILE is
import AUTOMOBILE-AGGREGATE
class sort Automobile
sort-axiom Automobile = Controller
operations
  attr-equal : Automobile, Automobile → Boolean
attributes
  engine-obj : Automobile → Engine-Class
  transmission-obj : Automobile → Transmission-Class
  wheels-obj : Automobile → Wheels-Class
  drives-assoc : Automobile → Drives
  connected-assoc : Automobile → Connected
methods
  create-automobile : → Automobile
events
  new-automobile : → Automobile
  engine-warning : Automobile, Integer → Automobile
axioms
  axioms omitted
end-class
```

Figure 6.48 Use of Sort Axiom in Aggregate Specification

6.7 Summary

This chapter presented a theory-based object model based on the restricted semantics presented in Chapter V of the Rumbaugh OMT object-oriented specification methodology. It defined the basic constructs necessary to capture an object-oriented specification as well as some basic relationships that must hold between specific types of object classes. These basic relationships form the laws of object composition and define the relationships between classes based on inheritance, association, and aggregation.

An object class is defined as a theory presentation with operations that represent attributes, methods, events, operations, and states while class inheritance is specified through the use of the *import* block and subsorting in a class specification. Links are defined generically as a class of objects that relate two or more objects from other classes. While link specifications are not exactly the same as a class specification, links may define attributes and operations. Associations are defined simply as a set of link objects. A unique type of specification is introduced to define an aggregate class. An aggregate specification defines a diagram of class specifications and the morphisms in the category

Spec. The classes in an aggregate specification consist of the aggregate's component classes and their associations. Finally, *event theories* and category theory operation are used to formally define the communication paths between classes in a domain model based on events specified in the dynamic model.

The next chapter builds on the information in this chapter by formally defining the transformations required to map graphically-based OMT domain models into the formal theory-based object model described in this chapter.

VII. Translation to Theory-Based Specification

7.1 Introduction

In the previous chapter, a theory-based model of object-orientation was defined based on concepts from Rumbaugh's OMT. This chapter defines the transformation rules to correctly translate a Rumbaugh OMT specification into its theory-based representation based on the theory-based object model. However, because there is no standard representation for OMT specifications other than their graphically-based diagrams, a generic OMT (GOMT) abstract syntax tree is used (as defined in Appendix A) to capture the components of the OMT diagrams in a more computationally familiar form. This GOMT AST was developed to ensure independence from any particular front-end tool. As defined in Appendix D, the demonstration system developed during this research translates the output of a commercial OMT-based tool into the GOMT before it is translated into O-SLANG to ensure isolation of tool-dependent concerns from the actual transformations. Thus, the transformations in this chapter define translations from a OMT specification captured in a GOMT AST into an O-SLANG abstract syntax tree. These transformation rules are described using first order algebraic axioms defined over the GOMT and O-SLANG abstract syntax trees using the notation and names defined in Appendix C.

The correctness of these transformations is established in Section 7.6 by showing that the formal semantics of each model (as defined in Chapter V) is preserved by the transformations defined in Sections 7.2, 7.3, and 7.4. This preservation of semantics is proved by defining a mapping from the theory-based representation in O-SLANG to the formal semantics and then showing that the semantics of an OMT specification is equivalent before and after its translation to O-SLANG.

In this chapter I use the convention that calligraphic uppercase letters such as $\mathcal{A}$ and $\mathcal{C}$ refer to associations and classes from the GOMT AST while outlined letter $\mathbb{A}$ and $\mathbb{C}$ refer to equivalent associations and classes in the O-SLANG AST (i.e., $\mathcal{C}$ is the GOMT class corresponding to the O-SLANG class $\mathbb{C}$). Dot notation is used to refer to both subobjects and attributes of objects in the

ASTs. For example $\mathbb{C}.Name$ would refer to the *name* attribute of class $\mathbb{C}$ while $\mathbb{C}.Class\text{-}Sort$ is the *class-sort* subobject which itself has the two attributes *class-sort-id* and *inherited-sort-id*. The distinction between attributes and subobjects are clear from context and the GOMT and O-SLANG AST definitions.

Axioms defined in this chapter are written as quoted strings of characters. In an axiom, all uppercase strings denote actual characters in the string while object and attribute names denote placeholders for the associated values of those objects and attributes. For example, the axiom

$$\text{“}SIZE(c.Name\text{-}OBJ(X)) = 1\text{”}$$

defines a string where the value of $c.Name$ is inserted. Thus if $c.Name = STUDENT$, the axiom becomes

$$\text{“}SIZE(STUDENT\text{-}OBJ(X)) = 1\text{”}$$

Within axioms, I also use pattern matching placeholders, "..." to match an arbitrary sequence of characters in an axiom. Thus a test

$$if\ \ ax = \text{“}...\ SIZE(IMAGE(A,X)) = 1\ ...\text{”}$$

returns true if the string of characters "$SIZE(IMAGE(A,X)) = 1$" is a substring of ax.

Many objects in the GOMT and O-SLANG ASTs define sets or sequences of objects. When forming a set or sequence, general set former notation is used. The set $S = \{x \mid P(x)\}$ denotes the set S where $x \in S$ only if $P(x)$ is true. Sequences are formed in the same manner using square brackets $[x \mid P(x)]$ instead of curly braces. Standard set and sequence operations are used, with $||$ representing the concatenation of two sequences.

7.2 Object Model Translations

A GOMT object model representation consists of two kinds of entities: classes and associations. This section discusses the translations of each of these in relation to the OMT object model.

7.2.1 Class Translation. The object model of a GOMT Class, $\mathcal{C}$, consists of the following items

- name
- set of superclass names
- set of component connections
- set of attributes

With the exception of the class name, all items are optional. The following transformations convert a GOMT class, $\mathcal{C}$, into an O-SLANG specification, $\mathbb{C}$.

7.2.1.1 Class Specification. If there are any abstract operations in $\mathcal{C}$ then $\mathcal{C}$ generates an *abstract class*, $\mathbb{C}_{abstract}$. If there are no abstract operations in $\mathcal{C}$, $\mathcal{C}$ generates a concrete class, $\mathbb{C}_{concrete}$. The name of $\mathcal{C}$ defines the name of $\mathbb{C}$ as well as the name of its class sort. All other transformations on $\mathbb{C}$ are made without regard to whether $\mathbb{C}$ is abstract or concrete.

$$\begin{aligned}
&\mathcal{C} \in \mathit{GOMT\text{-}DomainTheory.GOMT\text{-}Class} \\
&\quad \wedge (\forall\, (o)\ o \in \mathcal{C}.\mathit{GOMT\text{-}Op} \Rightarrow o.\mathit{is\text{-}abstract} = \mathit{false}) \\
&\qquad \Rightarrow \exists\, (\mathbb{C}_{Concrete})\ \mathbb{C}_{Concrete} \in \mathit{O\text{-}Slang\text{-}DomainTheory} \\
&\qquad\quad \wedge\ \mathbb{C}_{Concrete}.\mathit{Name} = \mathcal{C}.\mathit{Name} \\
&\qquad\quad \wedge\ \mathbb{C}_{Concrete}.\mathit{class\text{-}sort.classsort\text{-}id} = \mathcal{C}.\mathit{Name}
\end{aligned} \tag{OMT-1}$$

$$\begin{aligned}
&\mathcal{C} \in \mathit{GOMT\text{-}DomainTheory.GOMT\text{-}Class} \\
&\quad \wedge (\exists\, (o)\ o \in \mathcal{C}.\mathit{GOMT\text{-}Op} \wedge o.\mathit{is\text{-}abstract} = \mathit{true}) \\
&\qquad \Rightarrow \exists\, (\mathbb{C}_{Abstract})\ \mathbb{C}_{Abstract} \in \mathit{O\text{-}Slang\text{-}DomainTheory} \\
&\qquad\quad \wedge\ \mathbb{C}_{Abstract}.\mathit{Name} = \mathcal{C}.\mathit{Name} \\
&\qquad\quad \wedge\ \mathbb{C}_{Abstract}.\mathit{class\text{-}sort.classsort\text{-}id} = \mathcal{C}.\mathit{Name}
\end{aligned} \tag{OMT-2}$$

From this point forward in the translation rules, it is assumed that $\mathbb{C}$ is the O-SLANG class generated from $\mathcal{C}$ by Rules OMT-1 or OMT-2 and that $\mathbb{C}$ may represent either an abstract class or a concrete class.

7.2.1.2 Superclasses. The classes $\mathcal{C}$ inherits from are defined by a set of superclass names. In O-SLANG this requires importing each superclass and defining the class sort of $\mathbb{C}$ to be a subset of each of the superclass class sorts. This is accomplished by placing each superclass name in the import block of $\mathbb{C}$ and stating that the class sort of $\mathbb{C}$ is a subsort of each of its superclass class sorts as shown below (where $s_1 ... s_n$ are superclass names in $\mathcal{C}$.superclass).

$$\textit{class-sort } \mathcal{C}.\mathit{Name} < s_1 ... s_n$$

This translation is defined in Rules OMT-3 and OMT-4

$$s \in \mathcal{C}.\mathit{Superclass} \quad \Rightarrow \quad s \in \mathbb{C}.\mathit{Import} \tag{OMT-3}$$

$$s \in \mathcal{C}.\mathit{Superclass} \quad \Rightarrow \quad s \in \mathbb{C}.\textit{class-sort.inherited-sort-id} \tag{OMT-4}$$

7.2.1.3 Components. If $\mathcal{C}$ has component connections it is an aggregate object and requires the creation of an aggregate class. This aggregate class is then imported into $\mathbb{C}$ as defined in Section 7.2.3. A component of $\mathcal{C}$ consists of the following items.

- name
- qualifier (with a name and a datatype)
- role
- multiplicity

Besides generating an aggregate class, each component connection, c, in class $\mathcal{C}$ defines an attribute in $\mathbb{C}$ that takes the class sort of $\mathbb{C}$ as input and returns a set of component objects. As defined in Section 6.5, a component, c, becomes an object-valued attribute referencing a set of objects of class *c.Name.* If the *role* attribute is defined, then the name of the attribute becomes the role name given. If the *role* attribute is not defined, the component name appended with the string "-OBJ" is used to define the object-valued attribute.

$$\textit{c.Name-OBJ} : \mathcal{C}.\mathit{Name} \rightarrow \textit{c.Name-Class}$$

or

$$\textit{c.role} : \mathcal{C}.\mathit{Name} \rightarrow \textit{c.Name-Class}$$

The formal specification of this transformation is

$$c \in \mathcal{C}.Connection \Rightarrow \langle attr\text{-}name(c), [\mathcal{C}.Name], [c.Name\text{-}CLASS]\rangle \in \mathbb{C}.Attribute) \qquad \text{(OMT-5)}$$

where the function *attr-name* is defined below.

$$\begin{array}{l} defined?(c.role) \Rightarrow attr\text{-}name(c) = c.Role \\ undefined?(c.role) \Rightarrow attr\text{-}name(c) = c.Name\text{-}OBJ \end{array} \qquad (7.1)$$

A component qualifier is used to discriminate between components in a set. According to Section 6.5.2, a qualifier becomes an attribute of the component. This requirement is specified formally in Rule OMT-6. In this rule, c is a qualified component whose O-SLANG class specification is $\mathbb{C}_q$ and whose class set is $\mathbb{C}_{qs}$.

$$\begin{array}{l} c \in \mathcal{C}.Connection \wedge c.Name = \mathbb{C}_q.Name \wedge c.Name\text{-}CLASS = \mathbb{C}_{qs}.Name \\ \quad \wedge\ defined?(c.Qualifier) \Rightarrow \\ \qquad (\langle c.Qualifier.Name, [c.Name], [type(c.Qualifier)]\rangle \in \mathbb{C}_q.Attribute \\ \qquad \wedge\ \langle IMAGE, [\mathbb{C}_{qs}.Name, c.Qualifier.Name], [\mathbb{C}_{qs}.Name]\rangle \in \mathbb{C}_{qs}.Operation \\ \qquad \wedge\ \text{“}IMAGE(C,Q) = \{X \mid X \in C \wedge c.Qualifier.Name(C) = Q\}\text{”} \in \mathbb{C}_{qs}.Axiom) \end{array} \qquad \text{(OMT-6)}$$

where the function *type* is defined as

$$\begin{array}{l} defined?(c.datatype) \Rightarrow type(c) = c.Datatype \\ undefined?(c.datatype) \Rightarrow type(c) = c.Name \end{array} \qquad (7.2)$$

According to Section 6.5, a class, $\mathbb{C}$, with components also includes an attribute for each *association* whose components are all components (or subcomponents) of $\mathbb{C}$. Assuming there exists such an association, $\mathbb{A}$, the attribute takes the form show below. Because of the intricacies of determining exactly when this is applicable, the actual rule for this transformation is defined in Rule OMT-30.

$$\mathbb{A}.Name\text{-}ASSOC : \mathcal{C}.Name \rightarrow \mathbb{A}.Name$$

The component class named *c.Name* and its class set named *c.Name-CLASS* are both assumed to exist. As discussed above, an object with components generates an aggregate specification. This aggregate specification includes the definition of all lower-level components and must be imported into $\mathbb{C}$. This requirement is defined formally in Rule OMT-7.

$$(\exists\ (c)\ c \in \mathcal{C}.Connection) \Rightarrow \mathcal{C}.Name\text{-}AGGREGATE \in \mathbb{C}.Import \qquad \text{(OMT-7)}$$

The multiplicity of a component defines how many of each component may be part of the aggregate class $\mathbb{C}$. As defined by Rumbaugh, these multiplicities include:

- One
- Many
- Plus (with an integer)
- Optional
- Specified (with a set of Spec-Ranges which have one or two integers indicating the range of multiplicities)

where a multiplicity of *One* allows only one component, a multiplicity of *Many* allows zero or more components, *Plus* allows the user to specify a minimum number of components, *Optional* allows exactly zero or one component, and *Specified* allows the user to specify the exact number or range of components allowed to be part of the aggregate. Therefore, as defined in Section 6.4.1, the multiplicity of component c defines an O-SLANG axiom specifying the allowable number of components that may be part of $\mathbb{C}$. Generally, the axioms defined by the first four multiplicities (One, Many, Plus, and Optional) are simple and are given in O-SLANG syntax below.

$$\begin{aligned}
One &\mapsto SIZE(attr\text{-}name(c)(O)) = 1 \\
Many &\mapsto SIZE(attr\text{-}name(c)(O)) \geq 0 \\
Plus &\mapsto SIZE(attr\text{-}name(c)(O)) \geq c.Plus.integer \\
Optional &\mapsto SIZE(attr\text{-}name(c)(O)) = 0 \vee SIZE(attr\text{-}name(c)(O)) = 1
\end{aligned}$$

However, if the component is qualified, the multiplicity is defined on the *qualified image* operation defined in the component class by Rule OMT-6. The format of the qualified multiplicity axiom is show below.

$$\begin{aligned}
One &\mapsto SIZE(IMAGE(attr\text{-}name(c)(O), q)) = 1 \\
Many &\mapsto SIZE(IMAGE(attr\text{-}name(c)(O), q)) \geq 0 \\
Plus &\mapsto SIZE(IMAGE(attr\text{-}name(c)(O), q)) \geq c.Plus.integer \\
Optional &\mapsto SIZE(IMAGE(attr\text{-}name(c)(O), q)) = 0 \vee SIZE(IMAGE(attr\text{-}name(c)(O), q)) = 1
\end{aligned}$$

These requirements are formalized in the following axioms.

$$\begin{aligned}
&c \in \mathcal{C}.Connection \land c.Mult = One \Rightarrow \\
&\quad ((undefined?(c.Qualifier) \Rightarrow \text{“}SIZE(attr\text{-}name(c)(X)) = 1\text{”} \in \mathbb{C}.Axiom) \\
&\quad (defined?(c.Qualifier) \\
&\qquad \Rightarrow \text{“}SIZE(IMAGE(attr\text{-}name(c)(X), Q)) = 1\text{”} \in \mathbb{C}.Axiom))
\end{aligned} \tag{OMT-8}$$

$$\begin{aligned}
&c \in \mathcal{C}.Connection \land c.Mult = Many \Rightarrow \\
&\quad ((undefined?(c.Qualifier) \Rightarrow \text{“}SIZE(attr\text{-}name(c)(X)) \geq 0\text{”} \in \mathbb{C}.Axiom) \\
&\quad (defined?(c.Qualifier) \\
&\qquad \Rightarrow \text{“}SIZE(IMAGE(attr\text{-}name(c)(X), Q)) \geq 0\text{”} \in \mathbb{C}.Axiom))
\end{aligned} \tag{OMT-9}$$

$$\begin{aligned}
&c \in \mathcal{C}.Connection \land c.Mult = Plus \Rightarrow \\
&\quad ((undefined?(c.Qualifier) \\
&\qquad \Rightarrow \text{“}SIZE(attr\text{-}name(c)(X)) \geq c.Plus.integer\text{”} \in \mathbb{C}.Axiom) \\
&\quad (defined?(c.Qualifier) \\
&\qquad \Rightarrow \text{“}SIZE(IMAGE(attr\text{-}name(c)(X), Q)) \geq c.Plus.integer\text{”} \in \mathbb{C}.Axiom))
\end{aligned} \tag{OMT-10}$$

$$\begin{aligned}
&c \in \mathcal{C}.Connection \land c.Mult = Optional \Rightarrow \\
&\quad ((undefined?(c.Qualifier) \\
&\qquad \Rightarrow \text{“}SIZE(attr\text{-}name(c)(X)) = 0 \lor SIZE(attr\text{-}name(c)(X)) = 1\text{”} \in \mathbb{C}.Axiom) \\
&\quad (defined?(c.Qualifier) \\
&\qquad \Rightarrow \text{“}SIZE(IMAGE(attr\text{-}name(c)(X), Q)) = 0 \\
&\qquad\quad \lor SIZE(IMAGE(attr\text{-}name(c)(X), Q)) = 1\text{”} \in \mathbb{C}.Axiom))
\end{aligned} \tag{OMT-11}$$

Generating the axiom for a *Specified* multiplicity is more complex. It may be used to specify either an exact number, a range of numbers, or a combination of both. Each *Specified* multiplicity may have a number of *specified ranges.* For each specified range, s, if only one value ($value1$) is specified in an unqualified *Specified* multiplicity then the subaxiom

$$SIZE(attr\text{-}name(c)(O)) = s.Value1$$

is generated as part of the overall specified axiom. However, if two values ($value1$ and $value2$) are specified, then the multiplicity defines a range as shown in the following subaxiom.

$$SIZE(attr\text{-}name(c)(O)) \geq s.Value1 \land SIZE(attr\text{-}name(c)(O)) \leq s.Value2$$

Because a user may specify multiple *Specified* values or ranges, the axioms generated for each *Specified* multiplicity must be disjuncted to create a single axiom defining the possible multiplicities of a component c as shown below.

$$
\begin{array}{l}
c \in \mathcal{C}.Connection \wedge c.Mult = Specified \Rightarrow \\
\quad ((undefined?(c.Qualifier) \Rightarrow \\
\quad\quad OR(\{ax \mid s \in c.Mult \wedge \\
\quad\quad\quad (defined?(s.value2) \Rightarrow ax = \text{“}SIZE(attr\text{-}name(c)(X)) \geq s.value1 \\
\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ SIZE(attr\text{-}name(c)(X)) \leq s.value2\text{”}) \\
\quad\quad\quad \wedge\ (undefined?(s.value2) \Rightarrow ax = \text{“}SIZE(attr\text{-}name(c)(X)) = s.value1\text{”})\}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \in \mathbb{C}.Axiom) \\
\quad (defined?(c.Qualifier) \Rightarrow \\
\quad\quad OR(\{ax \mid s \in c.Mult \wedge \\
\quad\quad\quad (defined?(s.value2) \Rightarrow ax = \text{“}SIZE(IMAGE(attr\text{-}name(c)(X), Q)) \geq s.value1 \\
\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ SIZE(IMAGE(attr\text{-}name(c)(X), Q)) \leq s.value2\text{”}) \\
\quad\quad\quad \wedge\ (undefined?(s.value2) \\
\quad\quad\quad\quad \Rightarrow ax = \text{“}SIZE(IMAGE(attr\text{-}name(c)(X), Q)) = s.value1\text{”})\}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \in \mathbb{C}.Axiom))
\end{array}
\quad \text{(OMT-12)}
$$

where OR is a function that returns a single axiom that is the logical disjunction of all axioms in the input set.

7.2.1.4 Attributes. Each attribute, α, in $\mathcal{C}$ is either a normal or derived attribute. Each attribute in the GOMT AST consists of the following items.

- name
- datatype (optional)
- expression (optional)

Each normal attribute, α_{norm}, in $\mathcal{C}$ defines an attribute declaration in $\mathbb{C}$ of the form:

$$\alpha_{norm}.\text{Name: } \mathcal{C}.\text{Name} \rightarrow \text{type}(\alpha)$$

where *type* is the function defined in Equation 7.2.

An *expression* of a normal attribute is interpreted as the initial value computation for that the attribute. Therefore, each normal attribute expression defines an axiom in the axiomblock of $\mathbb{C}$ of the form

$$\alpha_{norm}.Name(CREATE\text{-}\mathcal{C}.Name(parameters)) = \alpha_{norm}.expression$$

where *CREATE-$\mathcal{C}$.Name(parameters)* is the create object function automatically created when $\mathbb{C}$ is defined. Parameters of the *create* function are defined by the dynamic or functional model as defined in Rule OMT-75 or OMT-85. If no dynamic model exists and *create* is not specified in the functional model then it is assumed there are no parameters to the *create* function. If the *create* function is specified in the dynamic model, then the parameter number and types defined in the dynamic model are used.

The formal transformations required for normal attributes are shown in Rule OMT-13.

$$\begin{aligned}
&\alpha \in \mathcal{C}.NormAttr \\
&\quad \Rightarrow (\langle \alpha.Name, [\mathcal{C}.Name], [type(\alpha)] \rangle \in \mathbb{C}.Attribute \\
&\qquad \wedge\ defined?(\alpha.expression) \Rightarrow \\
&\qquad\quad \text{“}\alpha.Name(CREATE\text{-}\mathcal{C}.Name(\mathcal{C}.Name, create\text{-}domain(\mathcal{C}))) \\
&\qquad\qquad\qquad = \alpha.expression\text{”} \in \mathbb{C}.Axiom)
\end{aligned} \qquad \text{(OMT-13)}$$

where the function *create-domain* is defined as

$$\begin{aligned}
&(\exists\ (\tau, a)\ \tau \in \mathcal{C}.Transition \wedge a \in \tau.Action \wedge a.Name = CREATE\text{-}\mathcal{C}.Name) \\
&\quad \Rightarrow create\text{-}domain(\mathcal{C}) = domain(a) \\
&\neg(\exists\ (\tau, a)\ \tau \in \mathcal{C}.Transition \wedge a \in \tau.Action \wedge a.Name = CREATE\text{-}\mathcal{C}.Name) \\
&\quad \Rightarrow create\text{-}domain(\mathcal{C}) = []
\end{aligned} \qquad (7.3)$$

and the function *domain* is defined as the following sequence.

$$domain(a) = [type(p)\ \mid\ p \in a.Parameter] \qquad (7.4)$$

Because a derived attribute calculates its valued based on normal attributes, each derived attribute, $\alpha_{derived}$, in $\mathcal{C}$ defines an operation declaration in $\mathbb{C}$ of the form

$$\alpha_{derived}.Name : \mathcal{C}.Name \rightarrow type(\alpha)$$

where again, *type* is defined in Equation 7.2. The user may define the value of a derived attribute in one of two ways: via a functional model decomposition or by providing an *expression* in the attribute definition. A functional model defines a set of axioms as described in Section 7.4. If the user provides an expression, the expression becomes an axiom in the axiom block of $\mathbb{C}$, . These expressions are assumed to have the correct O-SLANG syntax and semantics to compute the derived attribute value. The axiom generated is shown below.

$$\alpha_{derived}.Name(\mathcal{C}.Name) = \alpha_{derived}.expression$$

The formal transformations required for derived attributes are specified in Rule OMT-14 where *create-domain* is defined as in Equation 7.3.

$$\alpha \in \mathcal{C}.DerivedAttr \Rightarrow \langle \alpha.Name, [\mathcal{C}.Name], [type(\alpha)] \rangle \in \mathbb{C}.Operation$$
$$\wedge\ (defined?(\alpha.expression) \Rightarrow \text{``}\alpha.Name(\mathcal{C}.Name) = \alpha.expression\text{''} \in \mathbb{C}.Axiom) \quad \text{(OMT-14)}$$

7.2.1.5 Operations. Operations are transformed in the functional model as defined in Section 7.4 or via special operation definitions as defined in Section 7.5. The only exception to these rules is the operation *attr-equal*. The *attr-equal* operation determines if two objects of the same class have identical non-state attribute values. Therefore, if a class, $\mathcal{C}$, has *normal* attributes, it must have an *attr-equal* operation. The signature of the operation is shown below.

$$attr\text{-}equal : \mathcal{C}.Name, \mathcal{C}.Name \rightarrow Boolean$$

Assuming $\mathcal{C}$ has *normal* attributes $\alpha_1 ... \alpha_n$, the axiom defining the *attr-equal* operation takes the form

$$attr\text{-}equal(c_1, c_2) = \alpha_1(c_1) = \alpha_1(c_1) \wedge ... \wedge \alpha_n(c_1) = \alpha_n(c_1)$$

Formally, the transformations that create the *attr-equal* operation and its definition are

$$(\exists\ (\alpha)\ \alpha \in \mathcal{C}.NormAttr) \Rightarrow \langle ATTR\text{-}EQUAL, [\mathcal{C}.Name, \mathcal{C}.Name], [Boolean]\rangle \in \mathbb{C}.Operation$$
$$\wedge\ \text{``}ATTR\text{-}EQUAL(O_1, O_2) = attr\text{-}compare(\mathcal{C})\text{''} \in \mathbb{C}.Axiom \qquad \text{(OMT-15)}$$

where the function *attr-compare* is defined as the logical conjunction of equations between two attributes of two objects, O_1 and O_2 as defined below in Equation 7.5. The *AND* function is defined similar to the *OR* function as the conjunction of a set of axioms.

$$attr\text{-}compare(\mathcal{C}) = AND(\{\text{``}\alpha(O_1) = \alpha(O_2)\text{''} \mid \alpha \in \mathcal{C}.NormAttr\}) \qquad (7.5)$$

7.2.1.6 Methods. Because of my approach to using OMT in this research, methods are defined either in the dynamic or functional models; however, if there is no dynamic model and a *create* process is not defined in the functional model, a default *create* method must be defined. A default *create* method takes no inputs and produces a value of the class sort of of $\mathbb{C}$ as shown below.

$$CREATE\text{-}\mathcal{C}.Name :\rightarrow \mathcal{C}.Name$$

The default definition of the *create* method is shown below.

$$SIZE(\mathcal{C}.Transition) = 0 \wedge (\forall\ (p)\ p \in processes(\mathcal{C})\ p.Name \neq CREATE\text{-}\mathcal{C}.Name)$$
$$\Rightarrow \langle CREATE\text{-}\mathcal{C}.Name, [], [\mathcal{C}.Name]\rangle \in \mathbb{C}.Method \qquad \text{(OMT-16)}$$

In Rule OMT-16, the function *processes* returns the set of all subprocesses of a class or a process in a class as defined below.

$$\begin{aligned} processes(x) = \{p \mid\ & p \in Process(x) \\ & \vee\ p_1 \in processes(x) \wedge p \in Process(p) \end{aligned} \qquad (7.6)$$

The default values of normal attributes are used to create the definition of *create.* For each normal attribute, α, in $\mathcal{C}$ with a defined default value expression, the following axiom is generated as defined in Rule OMT-13.

$$\alpha.Name(CREATE\text{-}\mathcal{C}.Name()) = \alpha.expression$$

7.2.1.7 Events. When a default *create* process must be created as defined above, a corresponding *new* event must also be created to invoke the *create* method. This *new* event has the exact same domain and range as the *create* method

$$NEW\text{-}\mathcal{C}.Name :\rightarrow \mathcal{C}.Name$$

and its only axiom "invokes" the *create* method as shown below.

$$ATTR\text{-}EQUAL(NEW\text{-}\mathcal{C}.Name(), CREATE\text{-}\mathcal{C}.Name())$$

These two definitions are captured in the following transformation rules.

$$\begin{aligned}&SIZE(\mathcal{C}.Transition = 0) \wedge (\forall\ (p)\ p \in processes(\mathcal{C})\ p.Name \neq CREATE\text{-}\mathcal{C}.Name)\\&\quad \Rightarrow (\langle NEW\text{-}\mathcal{C}.Name, [], [\mathcal{C}.Name]\rangle \in \mathbb{C}.Event\end{aligned} \qquad \text{(OMT-17)}$$

$$\begin{aligned}&SIZE(\mathcal{C}.Transition = 0) \wedge (\forall\ (p)\ p \in processes(\mathcal{C})\ p.Name \neq CREATE\text{-}\mathcal{C}.Name)\\&\quad \Rightarrow \text{“}ATTR\text{-}EQUAL(NEW\text{-}\mathcal{C}.Name(), CREATE\text{-}\mathcal{C}.Name())\text{”} \in \mathbb{C}.Axiom)\end{aligned} \qquad \text{(OMT-18)}$$

where *processes* is defined in Equation 7.6.

7.2.2 Class Sets. Each class $\mathbb{C}$ generates a second specification called the *class set*, $\mathbb{C}_s$ that defines a set of objects of type $\mathbb{C}$. The name $\mathbb{C}$ defines the name of $\mathbb{C}_s$ as well as the name of the class sort of $\mathbb{C}_s$. The string "-CLASS" is simply appended to the name of the class set specification and class sort. To explicitly state that $\mathbb{C}_s$ defines a set of objects of type $\mathbb{C}$, a *contained-in* name is defined as the name of the defining class. The formal specification of the transformation is shown in Rule OMT-19.

$$\begin{aligned}&\mathcal{C} \in GOMT\text{-}DomainTheory.Class \Rightarrow \exists\ (\mathbb{C}_s)\ \mathbb{C}_s \in O\text{-}Slang\text{-}DomainTheory\\&\quad \wedge\ \mathbb{C}_s.Name = \mathbb{C}.Name\text{-}CLASS\\&\quad \wedge\ \mathbb{C}_s.class\text{-}sort.classsort\text{-}id = \mathbb{C}.Name\text{-}CLASS\\&\quad \wedge\ \mathbb{C}_s.contained\text{-}in = \mathbb{C}.Name\end{aligned} \qquad \text{(OMT-19)}$$

7.2.2.1 Class Set Superclasses. The set of superclass names of $\mathcal{C}$ defines the classes from which $\mathbb{C}$ inherits. As defined in Definition 6.2.1, the class set of the superclasses of $\mathcal{C}$ must also be imported into the class set of $\mathbb{C}$. This is accomplished by placing each superclass name in the import block of $\mathbb{C}_s$ as defined in Rule OMT-20.

$$s \in C.Superclass \quad \Rightarrow \quad s\text{-}CLASS \in \mathbb{C}_s.Import \tag{OMT-20}$$

7.2.2.2 Class Set Event. Each event in class $\mathbb{C}$ defines a *class event* in $\mathbb{C}_s$. These class events have the same signature as the events from class $\mathbb{C}$ with the class sort of $\mathbb{C}$ replaced by the class sort of $\mathbb{C}_s$. The formal transformation is shown below where the function *rest* returns all items in a sequence but the first item.

$$\begin{aligned} &e = \langle name, domain, range\rangle \in \mathbb{C}.Event \Rightarrow \\ &\quad \langle name, [\mathbb{C}.Name\text{-}CLASS] \,\|\, rest(domain), [\mathbb{C}.Name\text{-}CLASS] \,\|\, rest(range)\rangle \\ &\qquad\qquad \in \mathbb{C}_s.Event \end{aligned} \tag{OMT-21}$$

The purpose of class-level events are to distribute the object-level event to each object in the class set. Thus for each event in $\mathbb{C}_s$, an axiom is added via the following transformation.

$$\begin{aligned} &e = \langle name, domain, range\rangle \in \mathbb{C}.Event \Rightarrow \\ &\quad \text{“}\forall\ (C : c.Name, CC : c.Name\text{-}CLASS)\ C \in CC \Leftrightarrow \\ &e.Name(C, parameters(e)) \in e.Name(CC, parameters(e))\text{”} \in \mathbb{C}_s.Axiom \end{aligned} \tag{OMT-22}$$

where *parameters* is defined as

$$parameters(e) = [unique(x) \ \mid\ x \in rest(e.domain)] \tag{7.7}$$

and *unique* is a function that returns a unique symbol name based on the input symbol.

In Section 6.2.4, the Theory-Based Object Model requires each class to define a *new* event that causes the creation of a new object. Because the class sort of a class set is a set, a new class set object is simply an empty set. This requirements is captured by defining a *new* event

$$NEW\text{-}\mathbb{C}.Name\text{-}CLASS :\rightarrow \mathbb{C}_s.Name\text{-}CLASS$$

with the axiom defining a new class set to be empty.

$$NEW\text{-}\mathbb{C}.Name\text{-}CLASS() = EMPTY\text{-}SET$$

Formally, these definitions are captured by the following transformation rule.

$$\begin{aligned}&\mathbb{C}_s \in \textit{O-Slang-DomainTheory}\\ &\quad \langle \textit{NEW-}\mathbb{C}\textit{.Name-CLASS}, [], [\mathbb{C}\textit{.Name-CLASS}]\rangle \in \mathbb{C}_s\textit{.Event}\\ &\quad \wedge \text{“}\textit{NEW-}\mathbb{C}\textit{.Name-CLASS}() = \textit{EMPTY-SET}\text{”} \in \mathbb{C}_s\textit{.Axiom}\end{aligned} \qquad \text{(OMT-23)}$$

7.2.3 Aggregates. As discussed in Section 7.2.1, each class $\mathcal{C}$ with components defines an aggregate class $\mathbb{C}_{\mathbb{A}}$. This aggregate class has a special form that defines the colimit of a diagram. In this section $\mathcal{C}$ denotes the GOMT class that generates $\mathbb{C}$, the class specification, and $\mathbb{C}_{\mathbb{A}}$, the aggregate specification in O-SLANG. When generating the aggregate class, the name of $\mathbb{C}_{\mathbb{A}}$ is defined by simply appended the string "-AGGREGATE" to the name of the $\mathcal{C}$. Rule OMT-24 formally defines this translation requirement.

$$\begin{aligned}&c \in \mathcal{C}\textit{.Connection} \Rightarrow \exists\ (\mathbb{C}_{\mathbb{A}})\ \mathbb{C}_{\mathbb{A}} \in \textit{O-Slang-DomainTheory}\\ &\quad \wedge\ \mathbb{C}_{\mathbb{A}}\textit{.Name} = \mathcal{C}\textit{.Name-AGGREGATE}\end{aligned} \qquad \text{(OMT-24)}$$

The diagram of specifications and specification morphisms is defined by a set of nodes (specifications) and a set of arcs (specification morphisms). These nodes correspond to all classes, data types, associations, and event theories referenced by a class, or any of its superclasses or components. A specification is in the node set of $\mathbb{C}_{\mathbb{A}}$ if it is one of the following.

1. A specification imported in $\mathbb{C}$.
2. A component of $\mathcal{C}$.
3. The class set specification of a component of $\mathcal{C}$.
4. The aggregate specification of a component of $\mathcal{C}$.
5. Any specification imported by nodes in $\mathbb{C}_{\mathbb{A}}$.
6. An association specification whose connections are all nodes in $\mathbb{C}_{\mathbb{A}}$.
7. Any specification imported by components of two or more nodes of $\mathbb{C}_{\mathbb{A}}$.
8. A unique TRIV specification is added to the node set for each connection in an association specification $\mathbb{C}_{\mathbb{A}}$.Node.
9. An *event theory* defining the communication between nodes in $\mathbb{C}_{\mathbb{A}}$.
10. A *broadcast theory* defining the communication between multiple nodes in $\mathbb{C}_{\mathbb{A}}$.

The first seven of these transformations are defined next. Item 8, the creation of a unique TRIV specification, is created as part of Rule OMT-38 while the *event* and *broadcast* nodes and arcs are more complex and are defined in Section 7.2.3.1. The formal transformation rules for determining the set of nodes in an aggregate are expressed in Rules OMT-25 through OMT-39.

$$s \in \mathbb{C}.Import \Rightarrow \langle s, s\rangle \in \mathbb{C}_{\mathbb{A}}.Node \tag{OMT-25}$$

$$s \in \mathcal{C}.Connection \Rightarrow \langle s.Name, s.Name\rangle \in \mathbb{C}_{\mathbb{A}}.Node \tag{OMT-26}$$

$$s \in \mathcal{C}.Connection \Rightarrow \langle s.Name\text{-}CLASS, s.Name\text{-}CLASS\rangle \in \mathbb{C}_{\mathbb{A}}.Node \tag{OMT-27}$$

$$\begin{aligned} &s \in \mathcal{C}.Connection \ \wedge \ s.Name = \mathcal{C}'.Name \ \wedge \ (\exists\ (c)\ c \in \mathcal{C}'.Connection) \\ &\quad \Rightarrow \langle \mathcal{C}'.Name\text{-}AGGREGATE, \mathcal{C}'.Name\text{-}AGGREGATE\rangle \in \mathbb{C}_{\mathbb{A}}.Node \end{aligned} \tag{OMT-28}$$

$$\langle x, s.Name\rangle \in \mathbb{C}_{\mathbb{A}}.Node \ \wedge \ s1 \in s.Import \Rightarrow \langle s1, s1\rangle \in \mathbb{C}_{\mathbb{A}}.Node \tag{OMT-29}$$

$$\begin{aligned} &\mathcal{A} \in GOMT\text{-}DomainTheory.Assoc \wedge (\forall\ (c)\ c \in \mathcal{A}.Connection\ \langle x, c.Name\rangle \in \mathbb{C}_{\mathbb{A}}.Node \\ &\quad \Rightarrow ((\langle \mathcal{A}.Name, \mathcal{A}.Name\rangle \in \mathbb{C}_{\mathbb{A}}.Node) \\ &\qquad \langle \mathcal{A}.Name\text{-}ASSOC, [\mathbb{C}.Name], [\mathcal{A}.Name]\rangle \in \mathbb{C}.Attribute) \end{aligned} \tag{OMT-30}$$

$$\begin{aligned} &\langle z, s_1.Name\rangle, \langle y, s_2.Name\rangle \in \mathbb{C}_{\mathbb{A}}.Node \wedge x \in imports(s_1) \wedge x \in imports(s_2) \\ &\quad \Rightarrow \langle x, x\rangle \in \mathbb{C}_{\mathbb{A}}.Node \end{aligned} \tag{OMT-31}$$

where the function *imports* recursively defines the set of all imports of a given class as shown below.

$$\begin{aligned} &defined?(c.Node) \Rightarrow imports(c) = \{s \ \mid \ s \in imports(x) \vee \langle x, s.Name\rangle \in c.Node\} \\ &defined?(c.Import) \Rightarrow imports(c) = \{s \ \mid \ s \in c.Import \vee (s \in imports(x) \wedge x \in c.Import)\} \end{aligned} \tag{7.8}$$

The morphisms between nodes of $\mathbb{C}$ define the set of arcs in $\mathbb{C}$ and are critical to correctly defining the colimit of the diagram. An arc is in the arc set of $\mathbb{C}$ if for n_1, n_2 in the node set of $\mathbb{C}_{\mathbb{A}}$ one of the following holds.

1. n_1 is directly or indirectly imported by n_2
2. n_1 is a component of n_2
3. n_1 is superclass of n_2
4. $n_2 = n_1$-CLASS
5. $n_1 = n_2$-AGGREGATE
6. $n_1 = n_2$-LINK

7. n_2 is a connection of n_1, an association
8. n_1 is an event theory received by n_2 (or a subcomponent of n_2)
9. n_1 is an event theory sent by a component of n_2

Once the nodes of an aggregate are known, the arcs may be computed. Once again, the first seven items above are straightforward and discussed below, while defining the arcs between event and broadcast theories is more complicated and is discussed separately in Section 7.2.3.1. The first seven transformations may be expressed formally as shown below. In these transformations, n_1 and n_2 are classes in the O-SLANG Domain Theory whose names, $n_1.Name$ and $n_2.Name$, are in the node set of aggregate $\mathbb{C}_\mathbb{A}$ and $\mathcal{C}_{n1}$ and $\mathcal{C}_{n2}$ represent the GOMT classes that generated classes n_1 and n_2. The transformations to define the set of arcs for an aggregate $\mathbb{C}_\mathbb{A}$ are shown below.

$$\begin{aligned}&\langle x, n_1.Name\rangle, \langle y, n_2.Name\rangle \in \mathbb{C}_\mathbb{A}.Node\\ &\quad \wedge\ n_1.Name \in imports(n_2) \Rightarrow \langle\langle x, n_1.Name\rangle, \langle y, n_2.Name\rangle, \{\}\rangle \in \mathbb{C}_\mathbb{A}.Arc\end{aligned} \qquad \text{(OMT-32)}$$

$$\begin{aligned}&n_1.Name \in imports(n_2)\\ &\quad \Rightarrow \langle\langle n_1.Name, n_1.Name\rangle, \langle n_2.Name, n_2.Name\rangle, \{\}\rangle \in \mathbb{C}_\mathbb{A}.Arc\end{aligned} \qquad \text{(OMT-33)}$$

$$\begin{aligned}&n_1.Name \in n_2.ClassSort.Inherited\text{-}Sort\text{-}Id\\ &\quad \Rightarrow \langle\langle n_1.Name, n_1.Name\rangle, \langle n_2.Name, n_2.Name\rangle, \{\}\rangle \in \mathbb{C}_\mathbb{A}.Arc\\ &\qquad \wedge\ \langle\langle n_1.Name\text{-}CLASS, n_1.Name\text{-}CLASS\rangle,\\ &\qquad\quad \langle n_2.Name\text{-}CLASS, n_2.Name\text{-}CLASS\rangle, \{\}\rangle \in \mathbb{C}_\mathbb{A}.Arc\end{aligned} \qquad \text{(OMT-34)}$$

$$\begin{aligned}&n_2.Name = n_1.Name\text{-}CLASS\\ &\Rightarrow \langle\langle n_1.Name, n_1.Name\rangle, \langle n_2.Name, n_2.Name\rangle, \{\}\rangle \in \mathbb{C}_\mathbb{A}.Arc\end{aligned} \qquad \text{(OMT-35)}$$

$$\begin{aligned}&n_1.Name = n_2.Name\text{-}AGGREGATE\\ &\quad \Rightarrow \langle\langle n_1.Name, n_1.Name\rangle, \langle n_2.Name, n_2.Name\rangle, \{\}\rangle \in \mathbb{C}_\mathbb{A}.Arc\end{aligned} \qquad \text{(OMT-36)}$$

$$\begin{aligned}&n_1.Name = n_2.Name\text{-}LINK\\ &\Rightarrow \langle\langle n_1.Name, n_1.Name\rangle, \langle n_2.Name, n_2.Name\rangle, \{\}\rangle \in \mathbb{C}_\mathbb{A}.Arc\end{aligned} \qquad \text{(OMT-37)}$$

$$\begin{aligned}&c \in \mathcal{C}_{n1}.Connection \wedge c.Name = n_2.Name \wedge TRIV_x = \langle unique(TRIV), TRIV\rangle \Rightarrow\\ &\quad ((defined?(n_2.role)\\ &\qquad \Rightarrow \langle TRIV_x, \langle n_1.Name\text{-}LINK, n_1.Name\text{-}LINK\rangle, \{E \rightarrow n_2.role\}\rangle \in \mathbb{C}_\mathbb{A}.Arc)\\ &\quad \wedge\ (undefined?(n_2.role)\\ &\qquad \Rightarrow \langle TRIV_x, \langle n_1.Name\text{-}LINK, n_1.Name\text{-}LINK\rangle, \{E \rightarrow attr\text{-}name(n_2)\}\rangle \in \mathbb{C}_\mathbb{A}.Arc)\\ &\quad \wedge\ \langle TRIV_x, \langle n_2.Name, n_2.Name\rangle, \{E \rightarrow n_2.Name\}\rangle \in \mathbb{C}_\mathbb{A}.Arc\\ &\quad \wedge\ TRIV_x \in \mathbb{C}_\mathbb{A}.Node)\end{aligned} \qquad \text{(OMT-38)}$$

7.2.3.1 Communication Theories in Aggregates. When discussing *event* and *broadcast* theories, it is often difficult to determine exactly when to include them in aggregate nodes and arcs. If there is only one class that receives an event, the problem is not difficult. Event theories become nodes in an aggregate when (1) a class which is a node in the aggregate imports the event theory (the class sends the event), or (2) when the nodes of an aggregate include both a sending class and a receiving class. The formal transformation rules of this simple case are shown below

$$\begin{aligned}
& e \in \textit{O-Slang-DomainTheory.Event} \;\wedge\; \textit{size}(\textit{receives}(e)) = 1 \\
& \quad \wedge\; \langle n_1, n_1\rangle, \langle n_2, n_2\rangle \in \mathbb{C}_{\mathbf{A}}.\textit{Node} \;\wedge\; n_1 \in \textit{comp-sends}(e) \wedge\; n_2 \in \textit{comp-receives}(e) \\
& \qquad \Rightarrow (\langle e.\textit{Name}, e.\textit{Name}\rangle \in \mathbb{C}_{\mathbf{A}}.\textit{Node} \\
& \qquad\quad \wedge\; \langle\langle e.\textit{Name}, e.\textit{Name}\rangle, \langle n_1, n_1\rangle, \{\}\rangle \in \mathbb{C}_{\mathbf{A}}.\textit{Arc} \\
& \qquad\quad \wedge\; \langle\langle e.\textit{Name}, e.\textit{Name}\rangle, \langle n_2, n_2\rangle, \{\textit{domain-map}(e, n_2)\}\rangle \in \mathbb{C}_{\mathbf{A}}.\textit{Arc})
\end{aligned} \tag{OMT-39}$$

where *domain-map* is the mapping of sorts in the domain of the equivalent event signatures in c_1 to the sorts of the domain of event signature in c_2 as defined below (the function *index* is the index of the sort symbol within a sequence – the *domain-ident* of the event)

$$\begin{aligned}
\textit{domain-map}(c_1, c_2) \;=\; & \{\text{“}a_1 \rightarrow a_2\text{”} \mid a_1 \in e_1.\textit{domain-ident} \wedge a_2 \in e_2.\textit{domain-ident} \\
& \wedge\; \textit{index}(a_1, e_1.\textit{domain-ident}) = \textit{index}(a_2, e_2.\textit{domain-ident}) \\
& \wedge (e_1 \in c_1.\textit{Event} \wedge e_2 \in c_2.\textit{Event} \wedge e_1.\textit{Name} = e_2.\textit{Name})\}
\end{aligned} \tag{7.9}$$

and *receives*, *sends*, *comp-receives*, and *comp-sends* are functions that define a set of classes who send/receive a given event or who have components who send/receive a given event.

$$\begin{aligned}
\textit{receives}(e) = \{ & c.\textit{Name} \mid c \in \textit{GOMT-DomainTheory.Class} \\
& \wedge\; t \in c.\textit{Transition} \wedge e.\textit{Name} = t.\textit{Name}\}
\end{aligned} \tag{7.10}$$

$$\begin{aligned}
\textit{sends}(e) = \{ & c.\textit{Name} \mid c \in \textit{GOMT-DomainTheory.Class} \\
& \wedge\; t \in c.\textit{Transition} \wedge a \in t.\textit{Action} \wedge e.\textit{Name} = a.\textit{Action}.\textit{Name}\}
\end{aligned} \tag{7.11}$$

$$\textit{comp-receives}(e) = \{c.\textit{Name} \mid x \in \textit{receives}(e) \wedge (x \in \textit{components-of}(c) \vee x = c.\textit{Name})\} \tag{7.12}$$

$$\textit{comp-sends}(e) = \{c.\textit{Name} \mid x \in \textit{sends}(e) \wedge (x \in \textit{components-of}(c) \vee x = c.\textit{Name})\} \tag{7.13}$$

The function *components-of* defines a set consisting of the names of all classes which are components or sub-components of a given class as defined below.

$$\begin{aligned} \textit{components-of}(c) = \{&x.Name \mid x \in c.Connection \\ &\vee (x \in \textit{components-of}(y.Name) \wedge y \in \textit{components-of}(c))\} \end{aligned} \tag{7.14}$$

When there is more than a single class that receives an event, the computation of aggregate nodes and arcs becomes more difficult. As described in Section 6.6, a *broadcast* theory is defined to send an event to multiple classes. Determining where to place this broadcast theory is the problem. According to the Theory-Based Object Model, the broadcast theory should be placed in the lowest-level aggregate in the domain theory that includes (possibly as components or subcomponents of nodes in the aggregate) all of the sending classes and all of the receiving classes.

When the appropriate aggregate has been found, a broadcast theory is created as defined in Section 6.6. The broadcast theory imports the appropriate event theory and creates a unique event theory for each receiving class. An import arc is defined between each sending class and a single event theory, which in turn has an import arc between it and the broadcast theory. While the sending classes may share a single copy of an event theory specification in order to link to the broadcast theory, each receiving class must have a unique event theory specification that is mapped to the event in the broadcast theory for that particular receiving class. The transformations are defined formally below. Rule OMT-40 defines the broadcast theory while Rule OMT-41 defines the appropriate nodes and arcs in the aggregate.

$$\begin{array}{l}
e \in \mathit{O\text{-}Slang\text{-}DomainTheory.Event} \;\wedge\; \mathit{size}(\mathit{receives}(e)) > 1 \\
\quad \wedge\; \mathit{event} \in \mathit{e.Event} \\
\quad \wedge\; (c \in \mathit{receives}(e) \Rightarrow c \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}})) \\
\quad \wedge\; (c \in \mathit{sends}(e) \Rightarrow c \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}})) \\
\quad \wedge\; \neg(\exists\ (a)\ a \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}}) \wedge \mathit{defined?}(\mathit{a.Node}) \\
\qquad \wedge\; (c \in \mathit{receives}(e) \Rightarrow c \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}})) \\
\qquad \wedge\; (c \in \mathit{sends}(e) \Rightarrow c \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}}))) \\
\Rightarrow \mathbb{C}_{\mathbb{E}} \in \mathit{O\text{-}Slang\text{-}DomainTheory} \\
\quad \wedge\; \mathbb{C}_{\mathbb{E}}.\mathit{Name} = \mathit{e.Name\text{-}MULT} \\
\quad \wedge\; \mathbb{C}_{\mathbb{E}}.\mathit{Class\text{-}Sort.Class\text{-}Sort\text{-}Id} = \mathit{e.Name\text{-}SORT} \\
\quad \wedge\; \mathit{e.Name} \in \mathbb{C}_{\mathbb{E}}.\mathit{Import} \\
\quad \wedge\; (c \in \mathit{receives}(e) \Rightarrow \\
\qquad (\langle \mathit{e.Name}, [\mathit{c.Name}] \,\|\, \mathit{domain}(\mathit{event}), [\mathit{c.Name}] \rangle \in \mathbb{C}_{\mathbb{E}}.\mathit{Event} \\
\qquad \wedge\; \langle \mathit{c.Name\text{-}OBJ}, [\mathit{e.Name}], [\mathit{c.Name}] \rangle \in \mathbb{C}_{\mathbb{E}}.\mathit{Attribute} \\
\qquad \wedge\; \text{“}\mathit{c.Name\text{-}OBJ}(\mathit{e.Name}(\mathit{e.Class\text{-}Sort.Class\text{-}Sort\text{-}Id}, \mathit{domain}(\mathit{event})) \\
\qquad\quad = \mathit{e.Name}(\mathit{C.Name\text{-}OBJ}(\mathit{e.Class\text{-}Sort.Class\text{-}Sort\text{-}Id}), \mathit{domain}(\mathit{event}))) \\
\hfill \in \mathbb{C}_{\mathbb{E}}.\mathit{Axiom})) \qquad \text{(OMT-40)}
\end{array}$$

$$\begin{array}{l}
e \in \mathit{O\text{-}Slang\text{-}DomainTheory.Event} \;\wedge\; \mathit{size}(\mathit{receives}(e)) > 1 \\
\quad \wedge\; \mathit{emult} \in \mathit{O\text{-}Slang\text{-}DomainTheory.Event} \;\wedge\; \mathit{e.Name\text{-}MULT} = \mathit{emult.Name} \\
\quad \wedge\; (c \in \mathit{receives}(e) \Rightarrow c \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}})) \\
\quad \wedge\; (c \in \mathit{sends}(e) \Rightarrow c \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}})) \\
\quad \wedge\; \neg(\exists\ (a)\ a \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}}) \wedge \mathit{defined?}(\mathit{a.Node}) \\
\qquad \wedge\; (c \in \mathit{receives}(e) \Rightarrow c \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}})) \\
\qquad \wedge\; (c \in \mathit{sends}(e) \Rightarrow c \in \mathit{components\text{-}of}(\mathbb{C}_{\mathbf{A}}))) \\
\Rightarrow \langle \mathit{emult.Name}, \mathit{emult.Name} \rangle \in \mathbb{C}_{\mathbf{A}}.\mathit{Node} \\
\quad \wedge\; (c \in (\mathit{comp\text{-}receives}(e) \cap \{n \mid \langle n, x \rangle \in \mathbb{C}_{\mathbf{A}}.\mathit{Node}\}) \\
\qquad \wedge\; \mathit{ev} \in \mathit{c.Event} \wedge \mathit{ev.Name} = \mathit{e.Name} \\
\qquad \Rightarrow\; (x = \mathit{unique}(\mathit{e.Name}) \wedge \langle x, \mathit{e.Name} \rangle \in \mathbb{C}_{\mathbf{A}}.\mathit{Node} \\
\qquad\quad \wedge\; \langle \langle x, \mathit{e.Name} \rangle, \langle \mathit{c.Name}, \mathit{c.Name} \rangle, \mathit{domain\text{-}map}(e, c) \rangle \in \mathbb{C}_{\mathbf{A}}.\mathit{Arc} \\
\qquad\quad \wedge\; \langle \langle x, \mathit{e.Name} \rangle, \langle \mathit{emult.Name}, \mathit{emult.Name} \rangle, \mathit{domain\text{-}map}(e, \mathit{emult}) \rangle \\
\hfill \in \mathbb{C}_{\mathbf{A}}.\mathit{Arc})) \qquad \text{(OMT-41)}
\end{array}$$

7.2.3.2 Domain Theory Aggregate. To ensure that all classes, associations, and events are unified in the domain theory, an overall domain theory aggregate is created that combines all top-level classes into a single specification. This is accomplished by creating a top-level aggregate with all top-level classes as nodes. Then, Rules OMT-25 through OMT-41 add additional nodes (associations, common imports, and event and broadcast theories) to the aggregate and compute the necessary arcs. The top-level classes are those classes that are not components of any other class as defined by Equation 7.15. The function *top-level* takes a GOMT domain theory and returns a set of nodes.

$$\textit{top-level}(DT) = \{\langle c, c\rangle \mid c \in DT.Class\} \setminus \{\langle c, c\rangle \mid c.Name \in k.Connection \wedge k \in DT.Class\} \quad (7.15)$$

Thus, for each GOMT domain theory, a top-level aggregate is automatically created with an initial set of top-level nodes as defined in Rule OMT-42.

$$\begin{aligned} &\exists\, (\mathbb{C}_{\mathbb{A}})\ \mathbb{C}_{\mathbb{A}} \in \textit{O-Slang-DomainTheory} \\ &\quad \wedge\ \mathbb{C}_{\mathbb{A}}.Name = \textit{DOMAIN-THEORY} \\ &\quad \wedge\ \mathbb{C}_{\mathbb{A}}.Node = \textit{top-level(GOMT-DomainTheory)} \end{aligned} \qquad \text{(OMT-42)}$$

7.2.4 Association Translation. A GOMT association, $\mathcal{A}$, consists of the following items

- name
- set of class connections
- set of attributes
- set of operation definitions

Each GOMT association, $\mathcal{A}$, defines two O-SLANG specifications, a *link* specification, $\mathbb{A}_{\mathbb{L}}$, and an *association* specification $\mathbb{A}$ where the association specification defines a set of link objects similar to a class set specification. The name of $\mathcal{A}$ defines the names and class sorts of $\mathbb{A}$ and $\mathbb{A}_{\mathbb{L}}$. The formal rules for this transformation are shown below.

$$\begin{aligned} &\mathcal{A} \in \textit{GOMT-DomainTheory.Assoc} \Rightarrow \mathbb{A}_{\mathbb{L}} \in \textit{O-Slang-DomainTheory} \\ &\quad \wedge\ \mathbb{A}_{\mathbb{L}}.Name = \mathcal{A}.\textit{Name-LINK} \\ &\quad \wedge\ \mathbb{A}_{\mathbb{L}}.\textit{Class-Sort.ClassSort-Id} = \mathcal{A}.\textit{Name-LINK} \end{aligned} \qquad \text{(OMT-43)}$$

$$\begin{aligned} &\mathcal{A} \in \textit{GOMT-DomainTheory.Assoc} \Rightarrow \mathbb{A} \in \textit{O-Slang-DomainTheory} \\ &\quad \wedge\ \mathbb{A}.Name = \mathcal{A}.Name \\ &\quad \wedge\ \mathbb{A}.\textit{Class-Sort.ClassSort-Id} = \mathcal{A}.Name \\ &\quad \wedge\ \mathbb{A}.\textit{Link-Class} = \mathcal{A}.\textit{Name-LINK} \end{aligned} \qquad \text{(OMT-44)}$$

The association specification defines a set of links along with the association multiplicity defined by the association connections. Associations do not have attributes or operations. GOMT operations and attributes are used to define the link attributes and operations only. Therefore, the definition of the association relies solely on its set of class connections. These connections have four components.

- name
- qualifier (with a name and a datatype)
- role
- multiplicity

In order to reference sets of objects from the associated classes, each connection, c, in class $\mathcal{A}$ defines a sort in $\mathbb{A}$ that is unified with the class set sort of the associated class. The formal transformation of a connection to a sort in an association is defined in Rule OMT-45.

$$c \in \mathcal{A}.Connection \Rightarrow c.Name\text{-}CLASS \in \mathbb{A}.Sort \tag{OMT-45}$$

In order to constrain the multiplicity of the objects based the number of links in the association, there must be a way to determine the number of links a given object participates in. As defined in Section 6.4.1, this corresponds to the *image* operation. Assuming the association is a simple (no qualifier) binary association, the signature for the two *image* operations defined by components, c_1 and c_2, is shown below.

$$\begin{aligned} &IMAGE : \mathcal{A}.Name, c_1.Name \rightarrow c_2.Name\text{-}CLASS \\ &IMAGE : \mathcal{A}.Name, c_2.Name \rightarrow c_1.Name\text{-}CLASS \end{aligned}$$

Both operations take an association object (a set of links) and an object from one of the two associated classes and returns the set of objects that are associated with the input object by links in the association. Assuming two classes c_1 and c_2 are in a simple binary association *Assoc*, the *image* operation is defined as follows.

$$\begin{aligned} &\forall (S : Assoc, x : c_1.Name, y : c_2.Name) \\ &\quad l \in S \wedge attr\text{-}name(c_1)(l) = x \wedge attr\text{-}name(c_2)(l) = y \\ &\qquad \Leftrightarrow y \in image(S, x) \end{aligned}$$

The formal translations are defined below

$$\begin{aligned}
&c_1, c_2 \in \mathcal{A}.Connection \wedge undefined?(c_1.Qualifier) \wedge undefined?(c_2.Qualifier) \Rightarrow \\
&\quad \wedge \langle IMAGE, [\mathbb{A}.Name, c_1.Name], [c_2.Name\text{-}CLASS]\rangle \in \mathbb{A}.Operation \\
&\quad \langle IMAGE, [\mathbb{A}.Name, c_2.Name], [c_1.Name\text{-}CLASS]\rangle \in \mathbb{A}.Operation \\
&\quad \wedge \text{“}\forall(ASSOC : \mathbb{A}.Name)\ (X \in c_1.Name \wedge Y \in c_2.Name \\
&\qquad \wedge LINK \in ASSOC \wedge attr\text{-}name(c_1)(LINK) = X \\
&\qquad \wedge attr\text{-}name(c_2)(LINK) = Y) \\
&\quad\ \Leftrightarrow Y \in IMAGE(ASSOC, X) \wedge X \in IMAGE(ASSOC, Y)\text{”} \in \mathbb{A}.Axiom \qquad \text{(OMT-46)}
\end{aligned}$$

In order to reference objects from the associated classes, each connection, c, in class $\mathcal{A}$ defines a sort in $\mathbb{A}_{\mathbb{L}}$ that is unified with the class sort of the associated class. The formal transformation of a connection to a sort in an association is defined in Rule OMT-47.

$$c \in \mathcal{A}.Connection \Rightarrow c.Name \in \mathbb{A}_{\mathbb{L}}.Sort \qquad \text{(OMT-47)}$$

A connection qualifier is used to discriminate between links in an association. According to Section 6.4.2, a qualifier becomes an attribute of the link class. This requirement is specified formally in Rule OMT-48.

$$\begin{aligned}
&c_1, c_2 \in \mathcal{A}.Connection \wedge defined?(c_1.Qualifier) \Rightarrow \\
&\quad \langle c_1.Qualifier.Name, [c_2.Name], [type(c_1.Qualifier)]\rangle \in \mathbb{A}_{\mathbb{L}}.Attribute \qquad \text{(OMT-48)}
\end{aligned}$$

This qualified attribute is used to define a different image operation in the association specification. In this image operation, the qualifier is used as an additional parameter. Assuming component c_1 had a qualifier attached to it (which becomes an attribute of component c_2 by Rule OMT-48), the following *image* operation would be generated.

$$IMAGE : \mathcal{A}.Name, c_1.Name, type(c_1.Qualifier) \rightarrow c_2.Name\text{-}CLASS$$

This image operation is defined formally below.

$$\begin{aligned}
&c_1, c_2 \in \mathcal{A}.Connection \wedge defined?(c_1.Qualifier) \wedge undefined?(c_2.Qualifier) \Rightarrow \\
&\quad \wedge \langle IMAGE, [\mathbb{A}.Name, c_1.Name, type(c_1.Qualifier)], [c_2.Name\text{-}CLASS] \rangle \in \mathbb{A}.Operation \\
&\quad \langle IMAGE, [\mathbb{A}.Name, c_2.Name], [c_1.Name\text{-}CLASS] \rangle \in \mathbb{A}.Operation \\
&\quad \wedge \text{``}\forall (ASSOC : \mathbb{A}.Name)(X \in c_1.Name \wedge Y \in c_2.Name \wedge Z \in type(c_1.Qualifier) \\
&\qquad (\wedge\ LINK \in ASSOC \wedge attr\text{-}name(c_1)(LINK) = X \\
&\qquad \wedge\ attr\text{-}name(c_2)(LINK) = Y) \wedge c_1.Qualifier.Name(LINK) = Z \\
&\quad \Leftrightarrow Y \in IMAGE(ASSOC, X, Z) \wedge X \in IMAGE(ASSOC, Y)\text{''} \in \mathbb{A}.Axiom \qquad \text{(OMT-49)}
\end{aligned}$$

The multiplicity of a component defines how many of each component may be referred to by links in the association class $\mathbb{A}$ and thus generates multiplicity axioms over the *image* operations defined above. These multiplicity axioms are very similar to the aggregate multiplicity axioms defined in Section 7.2.1. According to the Theory-Based Object Model, for binary associations there are five types of multiplicities: one, many, optional, plus, or numerically specified. The axioms generated by these multiplicities for simple binary associations are shown below.

$$\begin{aligned}
One &\mapsto SIZE(IMAGE(A,O)) = 1 \\
Many &\mapsto SIZE(IMAGE(A,O)) \geq 0 \\
Plus &\mapsto SIZE(IMAGE(A,O)) \geq c.Plus.integer \\
Optional &\mapsto SIZE(IMAGE(A,O)) = 0 \vee SIZE(IMAGE(A,O)) = 1 \\
Specified &\mapsto SIZE(IMAGE(A,O)) = c.Specified.Spec\text{-}Range.value1 \\
Specified &\mapsto SIZE(IMAGE(A,O)) \geq c.Specified.Spec\text{-}Range.value1 \\
&\qquad \wedge SIZE(IMAGE(A,O)) \leq c.Specified.Spec\text{-}Range.value2
\end{aligned}$$

The formal association multiplicity transformations for simple associations are defined below.

$$c \in \mathcal{A}.Connection \wedge undefined?(c.Qualifier) \wedge c.Mult = One$$
$$\Rightarrow \text{“}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) = 1\text{”} \in \mathbb{A}.Axiom \qquad \text{(OMT-50)}$$

$$c \in \mathcal{A}.Connection \wedge undefined?(c.Qualifier) \wedge c.Mult = Many$$
$$\Rightarrow \text{“}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) \geq 0\text{”} \in \mathbb{A}.Axiom \qquad \text{(OMT-51)}$$

$$c \in \mathcal{A}.Connection \wedge undefined?(c.Qualifier) \wedge c.Mult = Plus \Rightarrow$$
$$\text{“}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) \geq c.Plus.integer\text{”} \in \mathbb{A}.Axiom \qquad \text{(OMT-52)}$$

$$c \in \mathcal{A}.Connection \wedge undefined?(c.Qualifier) \wedge c.Mult = Optional \Rightarrow$$
$$\text{“}X \in c.Name$$
$$\Rightarrow (SIZE(IMAGE(A,X)) = 0 \vee SIZE(IMAGE(A,X)) = 1)\text{”} \in \mathbb{A}.Axiom \qquad \text{(OMT-53)}$$

$$c \in \mathcal{A}.Connection \wedge undefined?(c.Qualifier) \wedge c.Mult = Specified \Rightarrow$$
$$OR(\{ax \mid s \in c.Mult \wedge$$
$$(defined?(s.value2) \Rightarrow ax = \text{“}X \in c.Name \Rightarrow (SIZE(IMAGE(A,X)) \geq s.value1$$
$$\wedge\ SIZE(IMAGE(A,X)) \leq s.value2)\text{”})$$
$$\wedge\ (undefined?(s.value2)$$
$$\Rightarrow ax = \text{“}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) = s.value1\text{”})\})$$
$$\in \mathbb{A}.Axiom \qquad \text{(OMT-54)}$$

The formal association multiplicity transformations for qualified associations are defined below.

$$c \in \mathcal{A}.Connection \wedge defined?(c.Qualifier) \wedge c.Mult = One$$
$$\Rightarrow \text{“}\forall\ (X : c.Name, Z : type(c.Qualifier))\ \ SIZE(IMAGE(A, X, Z)) = 1\text{”}$$
$$\in \mathbb{A}.Axiom \qquad \text{(OMT-55)}$$

$$c \in \mathcal{A}.Connection \wedge defined?(c.Qualifier) \wedge c.Mult = Many$$
$$\Rightarrow \text{“}\forall\ (X : c.Name, Z : type(c.Qualifier))\ \ SIZE(IMAGE(A, X, Z)) \geq 0\text{”}$$
$$\in \mathbb{A}.Axiom \qquad \text{(OMT-56)}$$

$$c \in \mathcal{A}.Connection \wedge defined?(c.Qualifier) \wedge c.Mult = Plus \ \Rightarrow$$
$$\text{“}\forall\ (X : c.Name, Z : type(c.Qualifier))$$
$$SIZE(IMAGE(A, X, Z)) \geq c.Plus.integer\text{”} \in \mathbb{A}.Axiom \qquad \text{(OMT-57)}$$

$$c \in \mathcal{A}.Connection \wedge defined?(c.Qualifier) \wedge c.Mult = Optional \ \Rightarrow$$
$$\text{“}\forall\ (X : c.Name, Z : type(c.Qualifier))$$
$$SIZE(IMAGE(A, X, Z)) = 0 \vee SIZE(IMAGE(A, X, Z)) = 1\text{”} \in \mathbb{A}.Axiom \qquad \text{(OMT-58)}$$

$$c \in \mathcal{A}.Connection \wedge defined?(c.Qualifier) \wedge c.Mult = Specified \Rightarrow$$
$$\text{“}\forall\ (X : c.Name, Z : type(c.Qualifier))\text{”} \parallel OR(\{ax \mid s \in c.Mult$$
$$\wedge\ (defined?(s.value2) \Rightarrow ax =$$
$$\text{“}SIZE(IMAGE(A, X, Z)) \geq s.value1 \wedge\ SIZE(IMAGE(A, X, Z)) \leq s.value2\text{”})$$
$$\wedge\ (undefined?(s.value2) \Rightarrow ax =$$
$$\text{“}SIZE(IMAGE(A, X, Z)) = s.value1\text{”})\}) \in \mathbb{A}.Axiom \qquad \text{(OMT-59)}$$

where OR is a function that returns the logical disjunction of all axioms in the input set.

Section 6.4 requires each association to define a *new* event to create a new association. There is no *create* method in an association since an association has no attributes. Because the class sort of an association is a set, the *new* event returns an empty set. This requirement is captured by defining a *new* event

$$NEW\text{-}\mathbb{A}.Name : \rightarrow \mathbb{A}.Name$$

with the axiom defining a new class set to be empty.

$$NEW\text{-}\mathbb{A}.Name() = EMPTY\text{-}SET$$

Formally, these definitions are captured by the following transformation rule.

$$\begin{aligned}&\mathbb{A} \in \textit{O-Slang-DomainTheory}\\ &\quad \langle \textit{NEW-}\mathbb{A}.\textit{Name}, [], [\mathbb{A}.\textit{Name}]\rangle \in \mathbb{A}.\textit{Event}\\ &\quad \wedge \text{“}\textit{NEW-}\mathbb{A}.\textit{Name}() = \textit{EMPTY-SET}\text{”} \in \mathbb{A}.\textit{Axiom}\end{aligned} \qquad \text{(OMT-60)}$$

7.2.4.1 Link Classes. A *link* class, $\mathbb{A}_{\mathbb{L}}$, defines an object with object valued attributes referencing each object in a given link. A link may also contain additional attributes, operations, methods, and events. The formal transformation for each item in $\mathcal{A}$ is defined below.

Connections. Because an association relates two or more classes, $\mathcal{A}$ must have at least two connections. These connections define which classes belong to the association. A connection consists of the following items:

- name
- qualifier (with a name and a datatype)
- role
- multiplicity

The multiplicity of a component defines axioms in the association class $\mathbb{A}$ and are not used in the link class definition. To reference objects from the associated classes, each connection, c, in class $\mathcal{A}$ defines an object valued attribute in $\mathbb{A}_{\mathbb{L}}$ that takes the class sort of $\mathbb{A}_{\mathbb{L}}$ as input and returns the reference to an object from class *c.Name*. This attribute declaration is generally of the form:

$$\textit{c.Name-OBJ} : \mathbb{A}_{\mathbb{L}}.\textit{Name} \rightarrow \textit{c.Name}$$

However, if the user has defined a role name for the connection, the role name is used as the attribute name as shown below.

$$\textit{c.role} : \mathbb{A}_{\mathbb{L}}.\textit{Name} \rightarrow \textit{c.Name}$$

Using the *attr-name* function defined in Equation 7.1, the formal transformation of a connection to an object valued attribute in a link is defined in Rule OMT-61.

$$\alpha \in \mathcal{A}.\textit{Connection} \Rightarrow \langle \textit{attr-name}(\alpha), [\mathcal{A}.\textit{Name-LINK}], [\alpha]\rangle \in \mathbb{A}_{\mathbb{L}}.\textit{Attribute}) \qquad \text{(OMT-61)}$$

Attributes. Additional link attributes may be entered directly into the object model and are transformed exactly like class attributes as defined in Rules OMT-13 and OMT-14 except that $\mathcal{C} \mapsto \mathcal{A}$ and $\mathbb{C} \mapsto \mathbb{A}_{\mathbb{L}}$.

Operations. User-defined link operations may also be entered directly into the object model. These operations are transformed exactly like class operations as defined in Section 7.5 (Rules OMT-89 – OMT-91) except that $\mathcal{C} \mapsto \mathcal{A}$ and $\mathbb{C} \mapsto \mathbb{A}_{\mathbb{L}}$.

Create Method/New Event. The only method created automatically for a link is the *create* method similar to the *create* method defined for classes in Rule OMT-16. However, to create a link, all object references must be provided to the *create* method. Therefore, the link *create* method has the signature

$$CREATE\text{-}linkname : component_1, ..., component_n \rightarrow linkname$$

while the *new* event has a similar signature.

$$NEW\text{-}linkname : component_1, ..., component_n \rightarrow linkname$$

Then, for each component in the link, an axiom of the form

$$component_i(CREATE\text{-}linkname(x_1, ..., x_n)) = x_i$$

is added to the axiom block of the link class, while the axiom causing the *new* event to invoke the *create* method

$$ATTR\text{-}EQUAL(NEW\text{-}\mathcal{A}.Name(x_1...x_n), CREATE\text{-}\mathcal{A}.Name(x_1...x_n)$$

is also generated and placed in the link class attribute block. The formal definition of the link *create* operation and *new* event is shown below, followed by the definition for the *attr-equal* operation.

$$\begin{aligned}&\mathcal{A} \in \textit{GOMT-DomainTheory} \Rightarrow \\ &\quad \langle \textit{CREATE-}\mathcal{C}.\textit{Name}, \textit{link-domain}, [\mathcal{A}.\textit{Name}]\rangle \in \mathbb{A}_{\mathbb{L}}.\textit{Method} \\ &\quad \wedge \langle \textit{NEW-}\mathcal{C}.\textit{Name}, \textit{link-domain}, [\mathcal{A}.\textit{Name}]\rangle \in \mathbb{A}_{\mathbb{L}}.\textit{Event}\end{aligned} \qquad \text{(OMT-62)}$$

$$\begin{aligned}&\mathcal{A} \in \textit{GOMT-DomainTheory} \Rightarrow \\ &\quad \text{``}\textit{ATTR-EQUAL}(\textit{CREATE-}\mathcal{C}.\textit{Name}(\textit{link-domain}), \textit{CREATE-}\mathcal{C}.\textit{Name}(\textit{link-domain}))\text{''} \\ &\qquad\qquad \in \mathbb{A}_{\mathbb{L}}.\textit{Axiom}\end{aligned} \qquad \text{(OMT-63)}$$

$$\begin{aligned}&\mathcal{A} \in \textit{GOMT-DomainTheory} \Rightarrow \\ &\quad \langle \textit{ATTR-EQUAL}, [\mathcal{A}.\textit{Name}, \mathcal{A}.\textit{Name}], [\textit{Boolean}]\rangle \in \mathbb{A}_{\mathbb{L}}.\textit{Operation} \\ &\qquad \wedge \text{``}\textit{ATTR-EQUAL}(O_1, O_2) = \textit{link-compare}(\mathcal{A})\text{''} \in \mathbb{A}_{\mathbb{L}}.\textit{Axiom}\end{aligned} \qquad \text{(OMT-64)}$$

where *link-domain* is defined as the sequence of parameters

$$\begin{aligned}\textit{link-domain}(c) \;=\; &[d \mid c \in \mathcal{A}.\textit{Connection} \\ &\wedge (\textit{defined?}(c.\textit{role}) \Rightarrow d = c.\textit{role}) \\ &\wedge (\textit{undefined?}(c.\textit{role}) \Rightarrow d = c.\textit{name})]\end{aligned} \qquad (7.16)$$

and *link-compare* is defined as the pairwise comparison of all link attributes and component attributes of $\mathbb{A}_{\mathbb{L}}$ as shown below.

$$\begin{aligned}\textit{link-compare}(\mathcal{C}) \;=\; &\textit{AND}(\{\text{``}\alpha(O_1) = \alpha(O_2)\text{''} \mid \alpha \in \mathcal{C}.\textit{NormAttr}\} \\ &\cup \{\text{``}\textit{attr-name}(c)(O_1) = \textit{attr-name}(c)(O_2)\text{''} \mid c \in \mathcal{C}.\textit{Connection}\})\end{aligned} \qquad (7.17)$$

The axiom that defines a link *create* method is simple since object references for each component must be provided. This axiom takes the form

$$\textit{attr-name}(x_i)(\textit{CREATE-}\mathcal{A}.\textit{Name}(x_1, \ldots x_n)) = x_i$$

where x_i represents a parameter in the parameter string of *create* and *attr-name*(x_i) is the object valued attribute name in the link class corresponding to that component. Formally, the creation of the set of axioms is given by Rule OMT-65.

$$\begin{aligned}&c \in \mathcal{A}.\textit{Connection} \wedge \textit{index}(\textit{create-domain}(\mathcal{C}), z) = \textit{index}(\mathcal{A}.\textit{Connection}, c.\textit{Name}) \\ &\quad \Rightarrow \text{``}c.\textit{Name}(\textit{CREATE-}\mathcal{C}.\textit{Name}(\textit{link-create-domain}(\mathcal{C}))) = z\text{''} \in \mathbb{C}.\textit{Axiom}\end{aligned} \qquad \text{(OMT-65)}$$

The function *index* is the index of a symbol within a sequence (the domain-ident of the event) and the function *link-create-domain* is defined as

$$\textit{create-domain}(a) = [\textit{unique}(x.\textit{Name}) \mid x \in a.\textit{Connection}] \tag{7.18}$$

and *unique* is a function that returns a unique symbol name based on the input symbol.

7.3 Dynamic Model Translations

The dynamic model of a GOMT Class C defines the dynamic behavior of a class in the form of a statechart. The dynamic model defines the allowable states that an object may be in and its behavior while in that state. Objects transition from one state to the next based on the receipt of events from external objects. After receiving an event, an object may react by changing state, invoking a method, or sending additional events. The dynamic model consists of the following items in the GOMT AST.

- set of states
- set of transitions

The dynamic model may or may not exist for a given class. The transformations defined in the dynamic model are assumed to occur after C is converted to an O-SLANG specification, $\mathbb{C}$, based on the object model. The transformations for states and transitions are defined below.

7.3.1 States. Each state in C has three possible attributes:

- name
- invariant axioms
- set of substates

If the dynamic model is defined in a class C then four types of declarations are added to $\mathbb{C}$. However, before adding these declarations, the set of states must be partitioned into n partitions such that 1) there is exactly one initial state in each partition, 2) each state is reachable from the start state, and 3) there are no transitions between partitions. If $n > 2$ then there are n distinct concurrent subdiagrams for class C.

1. First, for each partition $i \in 1...n$, a sort declaration as shown below is created.

$$\mathcal{C}.Name\text{-}STATE\text{-}i$$

2. Second a state attribute of the form

$$\mathcal{C}.Name\text{-}STATE\text{-}i : \mathcal{C}.Name \rightarrow \mathcal{C}.Name\text{-}STATE\text{-}i$$

   is created for each partition $i \in 1...n$.

3. Then, each state in partition $i \in 1...n$, σ generates a nullary operation (constant) in the states block of $\mathbb{C}$ of the form

$$\sigma.Name :\rightarrow \mathcal{C}.Name\text{-}STATE\text{-}i$$

4. Finally, for each pair of states in partition $i \in 1...n$, σ_1 and σ_2, the axiom

$$\sigma_1.Name \neq \sigma_2.Name$$

   is added to the axioms of $\mathcal{C}$.

Each state, σ, may also contain substates. If a state contains substates then four additional declarations, similar to the ones defined above, are generated for state σ. Again however, before the declarations can be added, the concurrent state partitions for the subdiagram must be computed. To determine these concurrent partitions, the set of substates of σ is separated into n partitions such that 1) there is exactly one initial state each each partition, 2) each state is reachable from the start state, and 3) there are no transitions between partitions. Again, $n > 2$ indicates that there are n distinct concurrent substate diagrams for state σ.

1. First, for each partition $i \in 1...n$, a sort declaration as shown below is created.

$$\sigma.Name\text{-}SUBSTATE\text{-}i$$

2. Second, for each partition $i \in 1...n$, a state attribute of the form

$$\sigma.Name\text{-}SUBSTATE\text{-}i : \mathcal{C}.Name \rightarrow \sigma.Name\text{-}SUBSTATE\text{-}i$$

   is created.

3. Then, each state ψ, in each partition $i \in 1...n$ of state σ, generates a nullary operation (constant) in the states block of $\mathbb{C}$.

$$\psi.Name :\rightarrow \sigma.Name\text{-}SUBSTATE\text{-}i$$

4. Finally, for each pair of states in partition $i \in 1...n$, ψ_1 and ψ_2, the axiom

$$\psi_1.Name \neq \psi_2.Name$$

is added to the axioms of $\mathcal{C}$.

Formal definition of these transformations starts with the computation of the set of concurrent state partitions. For a given state, s, its partition of concurrent state sets, π_s, is given by the following recursive definition.

$$\begin{array}{rcl} base(s) & = & \{\langle s_1, s_2\rangle \mid (\exists\,(\tau, s_1, s_2)\ \tau \in \mathcal{C}.Transition \wedge (s = s_1 \vee s = s_2) \wedge \tau.fromstate = s_1 \\ & & \wedge \tau.fromstate \neq \textit{Initial-State-Marker} \wedge \tau.ToState = s_2) \\ & & \vee\ (\langle s_1, s_m\rangle \in base \wedge \langle s_m, s_2\rangle \in base)\} \\ \pi_s & = & \{s \mid \langle s, s_m\rangle \in base \vee \langle s_m, s\rangle \in base\} \end{array} \tag{7.19}$$

Then the set of all partitions, $\Pi_{\mathbb{C}}$, is defined as below. Since $\Pi_{\mathbb{C}}$ is defined as a set, there is only one partition, $\pi_{\mathbb{C}}$ for each partition of concurrent states in $\mathbb{C}$. (Π is also computable on states that have substates.)

$$\Pi_{\mathbb{C}} = \{\pi_s \mid s \in \mathcal{C}.State\} \tag{7.20}$$

Once the partitioning of states is complete, the four O-SLANG declarations described above are defined. The first transformations create declarations for a sort and state attribute for each partition as shown in Rule OMT-66 and OMT-67.

$$i \in \Pi_{\mathbb{C}} \Rightarrow \mathcal{C}.Name\text{-}STATE\text{-}i \in \mathbb{C}.Sort \tag{OMT-66}$$

$$i \in \Pi_{\mathbb{C}} \Rightarrow \langle \mathcal{C}.Name\text{-}STATE\text{-}i, [\mathcal{C}.Name], [\mathcal{C}.Name\text{-}STATE\text{-}i]\rangle \in \mathbb{C}.StateAttr \tag{OMT-67}$$

The third transformation creates a nullary operation declaration for each state in the partition as shown in Rule OMT-68.

$$i \in \Pi_{\mathbb{C}} \Rightarrow (s \in \pi_i \Rightarrow \langle s.Name, [], [\mathcal{C}.Name\text{-}STATE\text{-}i]\rangle \in \mathbb{C}.State) \tag{OMT-68}$$

And finally, to ensure each state created by Rule OMT-68 is unique, Rule OMT-69 defines the appropriate state uniqueness axioms.

$$i \in \Pi_C \land s_1, s_2 \in \pi_i \land s_1 \neq s_2 \Rightarrow \text{“}s_1 \neq s_2\text{”} \in \mathbb{C}.Axiom \qquad \text{(OMT-69)}$$

Once a class's base states are defined, the substates, as discussed above, can be translated. Therefore, for any given superstate, σ, with Π_σ as defined in Equation 7.20, the following rules translate the substates of σ.

$$\begin{aligned}
&\sigma \in states(C) \land (\exists\ (s)\ s \in \sigma.State) \Rightarrow \\
&\quad i \in \Pi_\sigma \Rightarrow \\
&\qquad \sigma.Name\text{-}SUBSTATE\text{-}i \in \mathbb{C}.Sort && \text{(OMT-70)} \\
&\qquad \land \langle \sigma.Name\text{-}SUBSTATE\text{-}i, [C.Name], [\sigma.Name\text{-}SUBSTATE\text{-}i] \rangle \in \mathbb{C}.StateAttr && \text{(OMT-71)} \\
&\qquad \land\ s \in \pi_i \Rightarrow \langle s.Name, [], [\sigma.Name\text{-}SUBSTATE\text{-}i] \rangle \in \mathbb{C}.State && \text{(OMT-72)} \\
&\qquad \land\ s_1, s_2 \in \pi_i \land s_1 \neq s_2 \Rightarrow \text{“}s_1 \neq s_2\text{”} \in \mathbb{C}.Axiom && \text{(OMT-73)}
\end{aligned}$$

where *states* is the set of all states of C as defined below.

$$states(C) = \{s \mid s \in C.State \lor (s \in s_1.State \land s_1 \in states(C))\} \qquad (7.21)$$

7.3.2 Transitions. In the dynamic model, transitions are used to represent incoming events and actions taken by the object. Transitions translate into events, methods, axioms, and event theories in the O-SLANG AST and consist of six components.

- name
- set of parameters (name and datatype)
- set of guard conditions defined via axioms
- set of actions
- from-state
- to-state

As defined in Section 6.2.3, actions are used to specify methods or events sent to other objects. In the GOMT AST, actions are decomposed into three components.

- name
- sequence of parameters (each with a name and datatype)

- set of actions

If the action name is *SEND* the object is to send the parameterized event specified by the sub-action. This is the only valid use of sub-actions. If the action name is not *SEND* then the action defines a method in $\mathbb{C}$. Therefore, a transition defines a receive event along with possibly multiple methods and send events. The relationship between these events and methods are defined by a transition axiom.

Each transition, τ, in class $\mathcal{C}$ defines an incoming event signature in the event block of $\mathbb{C}$. Each parameter, p, in τ.parameter becomes a parameter in the event as defined below.

$$\tau.Name : \mathcal{C}.Name, p_1...p_n \rightarrow \mathcal{C}.Name$$

The translation into the O-SLANG AST is

$$\tau \in \mathcal{C}.Transition \Rightarrow \langle \tau.Name, [\mathcal{C}.Name] \parallel domain(\tau), [\mathcal{C}.Name] \rangle \in \mathbb{C}.Event \qquad \text{(OMT-74)}$$

where *domain* is defined as in Equation 7.4.

Each non-SEND action, s, in τ defines a signature in the method block of $\mathbb{C}$ and each parameter, p, in *s.parameter* becomes a parameter of the method as shown below.

$$s.Name : \mathcal{C}.Name, p_1...p_n \rightarrow \mathcal{C}.Name$$

This transformation is captured formally in Rule OMT-75.

$$\begin{aligned} &\tau \in \mathcal{C}.Transition \wedge s \in \tau.Action \wedge \tau.Name \neq SEND \\ &\quad \Rightarrow \langle s.Name, [\mathcal{C}.Name] \parallel domain(s.Parameter), [\mathcal{C}.Name] \rangle \in \mathbb{C}.Method \qquad \text{(OMT-75)} \end{aligned}$$

Each subaction, s, in τ defines an outgoing event in $\mathbb{C}$ as well as an event theory specification that is used later in an aggregate specification to unify $\mathbb{C}$ with the receiving object's class specification. The event theory defines an event sort and signature and is imported into $\mathbb{C}$; therefore, instead of creating a signature for the outgoing event, the event theory name (which is the event name, s.Name) is added to the import block of $\mathbb{C}$. An event theory is shown below

event s.Action.Name is
  class-sort s.Action.Name-SORT
  sorts $p_1, ..., p_n$
  events s.Action.Name : s.Action.Name-SORT, p1...pn $\rightarrow$ *s.Action.Name-SORT*
end-event

where $p_1...p_n$ denote the parameters of the outgoing event. Formally, this transformation is

$$\tau \in \mathcal{C}.Transition \wedge s \in \tau.Action \wedge s.Name = SEND$$
$$\Rightarrow \mathbb{C}_\mathbb{E} \in \textit{O-Slang-DomainTheory} \quad \text{(OMT-76)}$$

$$\wedge\ \mathbb{C}_\mathbb{E}.Name = s.Action.Name \quad \text{(OMT-77)}$$

$$\wedge\ \mathbb{C}_\mathbb{E}.Classsort.\textit{Class-Sort-Id} = s.Action.\textit{Name-SORT} \quad \text{(OMT-78)}$$

$$\wedge\ \langle s.Action.Name, [s.Action.\textit{Name-SORT}] \,\|\, domain(s.Action.Parameter), [s.Action.\textit{Name-SORT}]\rangle \in \mathbb{C}_\mathbb{E}.Event \quad \text{(OMT-79)}$$

$$\wedge\ s.Action.Name \in \mathbb{C}.Import \quad \text{(OMT-80)}$$

where *domain* is again defined in Equation 7.4.

Before sending an event, the sending object must know where to send it. Section 6.6 requires each event have an object-valued attribute of the form

$$s.\textit{Name-OBJ} : \mathcal{C}.Name \rightarrow s.\textit{Name-SORT}$$

to define where an event is sent. This declaration is generated and placed in the attribute block of $\mathbb{C}$. Formally, this translation is shown in Rule OMT-81 (the sort *s.Name-SORT* is defined in the imported event theory in Rule OMT-78 above).

$$\tau \in \mathcal{C}.Transition \wedge s \in \tau.Action \wedge s.Name = SEND$$
$$\Rightarrow \langle s.Action.\textit{Name-OBJ}, [\mathcal{C}.Name], [s.Action.\textit{Name-SORT}]\rangle \in \mathbb{C}.Attribute \quad \text{(OMT-81)}$$

Each transition defines an axiom that causes the object, upon receipt of an incoming event in the appropriate state, to state change as well as invoke methods and send events. This axiom has five parts: current state, guard condition, new state, method invocations, and sending of events. These parts are merged into a single axiom of the form:

$$\textit{old-state} \wedge \textit{guard-condition} \Rightarrow \textit{new-state} \wedge \textit{method-invocations} \wedge \textit{event-sends}$$

Using functions to define the individual aspects of the axiom (i.e., old-state, guard-condition, new-state, method-invocations, and event-sends), Rule OMT-82 defines the axiom for each event except the initial *new* event.

$$\begin{aligned} &\tau \in \mathcal{C}.\textit{Transition} \wedge \tau.\textit{FromState} \neq \textit{Initial-State-Marker} \\ &\quad \Rightarrow \text{“}\textit{old-state}(\tau.\textit{FromState}) \wedge \textit{guard-condition}(\tau) \Rightarrow \textit{new-state}(\tau) \\ &\quad\quad \wedge\ \textit{method-invocations}(\tau) \wedge \textit{event-sends}(\tau)\text{”} \in \mathbb{C}.\textit{Axiom} \end{aligned} \qquad \text{(OMT-82)}$$

$$\begin{aligned} &\tau \in \mathcal{C}.\textit{Transition} \wedge \tau.\textit{FromState} = \textit{Initial-State-Marker} \Rightarrow \text{“}\textit{new-state}(\tau) \\ &\quad \wedge\ \textit{method-invocations}(\tau) \wedge \textit{event-sends}(\tau)\text{”} \in \mathbb{C}.\textit{Axiom} \end{aligned} \qquad \text{(OMT-83)}$$

The functions *old-state*, *guard-condition*, *new-state*, *method-invocations* and *event-sends* are defined below.

Old-State Because the *from-state* is a mandatory part of a transition, an *old-state* is always generated. If the $\tau.\textit{from-state}$ is a top-level state then the old-state part of the axiom is simply

$$\mathcal{C}.\textit{Name-STATE}(o) = \tau.\textit{from-state}$$

However, if the from-state is a substate of another state then the superstate must be in the correct state as well, as shown below.

$$\begin{aligned} &\mathcal{C}.\textit{Name-STATE}(o) = \textit{superstate}(\tau, \textit{from-state}) \\ &\quad \wedge\ \textit{superstate}(\tau, \textit{from-state})\textit{-SUBSTATE}(o) = \tau.\textit{from-state} \end{aligned}$$

In this example, *superstate* is a function that determines the superstate of a substate. Obviously, there can be many levels of substates so that an arbitrary number of superstates may be included in the old state part of the axiom.

Formally, the definition of the *old-state* function is shown in Rule 7.22. The definition of *old-state* is recursive and relies on the partitioning of states defined above in Equation 7.19.

$$\begin{array}{l}(\exists\ (s, s_1)\ s \in \mathit{states}(\mathcal{C}) \land \sigma = s_1.\mathit{Name} \land s_1 \in s.\mathit{State}) \land \sigma \in \pi_i \\ \quad \Rightarrow \mathit{old\text{-}state}(\sigma) = \text{``}\mathit{old\text{-}state}(s) \land s.\mathit{Name\text{-}SUBSTATE\text{-}i}(\mathcal{C}.\mathit{Name}) = \sigma\text{''} \\ (s \in \mathit{states}(\mathcal{C}) \Rightarrow \sigma = s_1.\mathit{Name} \land s_1 \notin s.\mathit{State}) \land \sigma \in \pi_i \\ \quad \Rightarrow \mathit{old\text{-}state}(\sigma) = \mathcal{C}.\mathit{Name\text{-}STATE}(\mathcal{C}.\mathit{Name}) = \sigma\text{''}\end{array} \tag{7.22}$$

where *states* is defined in Equation 7.21 above.

Guard-Condition The guard condition part of the transition axiom is optional. If the guard condition does exists, it is assumed that the guard condition is an axiom written in O-SLANG syntax based on the object's attribute values and incoming parameter values only. Thus if the guard condition exists, it requires no translation.

$$\begin{array}{l}\mathit{defined?}(\tau.\mathit{Axiom}) \Rightarrow \mathit{guard\text{-}condition}(\tau) = \tau.\mathit{Axiom} \\ \mathit{undefined?}(\tau.\mathit{Axiom}) \Rightarrow \mathit{guard\text{-}condition}(\tau) = \text{``}\mathit{true}\text{''}\end{array} \tag{7.23}$$

New-State Because a transition always has a *to-state*, the new state part of the transition axiom is defined by *τ.to-state* and takes the form

$$\mathcal{C}.\mathit{Name\text{-}STATE}(\tau.\mathit{Name}(o, p_1...p_n)) = \tau.\mathit{tostate}$$

or, if the state is a substate,

$$\mathit{superstate}(\tau, \mathit{tostate})\text{-}\mathit{SUBSTATE}(\tau.\mathit{Name}(o, p_1...p_n)) = \tau.\mathit{tostate}$$

where $p_1...p_n$ denote the parameters of the incoming event. If the transition occurs in a substate diagram, the values of superstate attributes do not change. Formally, the definition of *new-state* is

$$\begin{array}{l}(\exists\ (s, s_1)\ s \in \mathit{states}(\mathcal{C}) \land \tau.\mathit{ToState} = s_1.\mathit{Name} \land s_1 \in s.\mathit{State}) \land \tau.\mathit{ToState} \in \pi_i \\ \quad \Rightarrow \mathit{new\text{-}state}(\tau) = \text{``}s.\mathit{Name\text{-}SUBSTATE\text{-}i}(\tau.\mathit{Name}(\mathcal{C}.\mathit{Name}, \mathit{domain}(\tau)) = \tau.\mathit{ToState}\text{''} \\ (s \in \mathit{states}(\mathcal{C}) \Rightarrow \tau.\mathit{ToState} = s_1.\mathit{Name} \land s_1 \notin s.\mathit{State}) \land \tau.\mathit{ToState} \in \pi_i \\ \quad \Rightarrow \mathit{new\text{-}state}(\tau) = \mathcal{C}.\mathit{Name\text{-}STATE}(\mathcal{C}.\mathit{Name}, \mathit{domain}(\tau)) = \tau.\mathit{ToState}\text{''}\end{array} \tag{7.24}$$

where *domain* is defined in Equation 7.4.

Method-Invocations A non-SEND action, a, specifies that a method is invoked as the result of the event receipt. As defined in the Section 6.2.5, the form used to specify method invocation is

$$\mathit{attr\text{-}equal}(\tau.\mathit{Name}(o, p_1...p_n), \tau.\mathit{Action}.\mathit{Name}(o, p_{a1}...p_{a2}))$$

where $p_1...p_n$ denote the parameters of the incoming event and $p_{a1}...p_{a2}$ denote the parameters of the method. The formal definition of the *method-invocations* function is shown below.

$$\begin{aligned}\mathit{method\text{-}invocations}(\tau) \quad = \quad & \mathit{AND}(\{\mathit{inv} \mid s \in \tau.\mathit{Action} \land s.\mathit{Name} \neq \mathit{SEND} \\ & \land\ \mathit{inv} = \text{``}\mathit{ATTR\text{-}EQUAL}(\tau.\mathit{Name}(\mathcal{C}.\mathit{Name}, \mathit{domain}(\tau)), \\ & \quad s.\mathit{Name}(\mathcal{C}.\mathit{Name}, \mathit{domain}(s)))\text{''}\} \cup \{\text{``}\mathit{true}\text{''}\})\end{aligned} \tag{7.25}$$

The *AND* function in Equation 7.25 denotes the logical conjunction of all axioms in the input set. The *true* axiom ensures that there is at least one axiom returned from *method-invocations* thus ensuring that Rule OMT-82 is well formed.

Event-Sends SEND actions represent the sending of a subaction event to the object whose reference is stored in the appropriate object-valued attribute. Therefore, a SEND action with sub-action, s, generates the axiom

$$s.\mathit{Name\text{-}OBJ}(\tau.\mathit{Name}(o, p_1...p_n)) = s.\mathit{Name}(s.\mathit{Name\text{-}OBJ}(o), p_{s1}...p_{s2})$$

where $p_1...p_n$ denote the parameters of the incoming event and $p_{a1}...p_{a2}$ denote the parameters of the outgoing event. *Event-Sends* is formally defined in Equation 7.26.

$$\begin{aligned}\mathit{event\text{-}sends}(\tau) = \mathit{AND}(\{\mathit{snd} \mid\ & s \in \tau.\mathit{Action} \land s.\mathit{Name} = \mathit{SEND} \\ \land\ \mathit{snd} = \text{``}& s.\mathit{Action}.\mathit{Name\text{-}OBJ}(\tau.\mathit{Name}(\mathcal{C}.\mathit{Name}, \mathit{domain}(\tau))) = \\ & s.\mathit{Action}.\mathit{Name}(s.\mathit{Action}.\mathit{Name\text{-}OBJ}(X), \mathit{domain}(s.\mathit{Action})))\text{''}\} \\ & \cup \{\text{``}\mathit{true}\text{''}\})\end{aligned} \tag{7.26}$$

Once the valid transitions have been transformed and all incoming events and states defined in $\mathbb{C}$, invalid transitions can be computed. Because the theory-based object model assumes there is no reaction to an event that occurs in a state with no explicitly defined transition for that event, axioms explicitly stating this assumption must be generated. These axioms are of the form

$$\mathit{old\text{-}state} \Rightarrow \mathit{same\text{-}state}$$

which is formally generated by Rule OMT-84.

$$\begin{aligned}&s \in \mathbb{C}.State \wedge e \in \mathbb{C}.Event\\&\quad \wedge \neg(\exists\ (\tau)\ \tau \in \mathcal{C}.Transition \wedge \tau.Name = e.Name \wedge \tau.FromState = s.Name)\\&\qquad \Rightarrow \text{“}old\text{-}state(\sigma) \Rightarrow same\text{-}state(e, s)\text{”} \in \mathbb{C}.Axiom\end{aligned} \tag{OMT-84}$$

where *old-state* is defined as before in Equation 7.22 and *same-state* is defined below in Equation 7.27.

$$\begin{aligned}&(\exists\ (s, s_1)\ s \in states(\mathcal{C}) \wedge \sigma = s_1.Name \wedge s_1 \in s.State) \wedge \sigma \in \pi_i\\&\quad \Rightarrow same\text{-}state(e, \sigma) = \text{“}s.Name\text{-}SUBSTATE\text{-}i(e.Name(event\text{-}domain(e)) = \sigma\text{”}\\&(s \in states(\mathcal{C}) \rightarrow \sigma = s_1.Name \wedge s_1 \notin s.State) \wedge \sigma \in \pi_i\\&\quad \Rightarrow same\text{-}state(e, \sigma) = \mathcal{C}.Name\text{-}STATE(event\text{-}domain(e)) = \sigma\text{”}\end{aligned} \tag{7.27}$$

and *event-domain* is defined as

$$event\text{-}domain(e) = [unique(x) \mid x \in e.Domain\text{-}Ident]$$

and *unique* is a function that returns a unique symbol name.

7.4 Functional Model Translation

The OMT functional model is depicted as a basic dataflow diagram and defines a set of processes and the dataflow between them. In the GOMT AST, a class $\mathcal{C}$ functional model consists of the following components.

- set of processes (with subprocesses)
- set of dataflows
- set of datastores

The functional model may or may not exist in a given class. If it does, it is assumed that the following transformations are performed after $\mathcal{C}$ is converted to an O-SLANG specification, $\mathbb{C}$, based on the object model. The translation of each of the above components is discussed below.

7.4.1 Processes. As defined in the GOMT AST, processes have the following components.

- name
- set of input data flows (name and datatype)
- set of output data flows (name and datatype)

- set of subprocesses

As interpretted by the theory-based object model, processes define purely functional methods or operations that take data of the type defined by the input dataflows and produce data of the type of defined by the output data flows as defined in Section 5.5. If the output dataflow, o, of a process, p, or any of its subprocesses, is to a datastore, then the process modifies the object, or subobjects, of which it is a part and defines a method. If a dataflow is to/from a datastore, the name of the object-valued attribute referencing the datastore becomes the parameter name (i.e., the datastore is input to the method) instead of the dataflow datatype. If any subprocesses of p have dataflows that are input from datastores, those datastores must be also be part of the method input parameters. A method signature for a process p is defined as

$$p.Name \;:\; \mathcal{C}.Name, flowtype(i_1)...flowtype(i_n) \rightarrow \mathcal{C}.Name$$

where $i_1...i_n$ are the input dataflows and *flowtype* is a function that returns the datastore object-valued attribute or dataflow type depending on whether the dataflow is from a datastore. Formally, the transformation of processes that represent methods is defined in Rule OMT-85.

$$\begin{aligned}&(p \in processes(\mathcal{C}) \wedge size(datastores\text{-}modified(\mathcal{C},p)) > 1\\&\quad \Rightarrow \langle p.Name, [\mathcal{C}.Name] \,\|\, dataflow\text{-}domain(p), [\mathcal{C}.Name]\rangle \in \mathbb{C}.Method)\\&\wedge\ (p \in processes(\mathcal{C}) \wedge size(datastores\text{-}modified(\mathcal{C},p)) = 1\\&\quad \Rightarrow \langle p.Name, [datastore\text{-}sort(\mathcal{C},p)] \,\|\, dataflow\text{-}domain(p),\\&\qquad\qquad [datastore\text{-}sort(\mathcal{C},p)]\rangle \in \mathbb{C}.Method)\end{aligned} \tag{OMT-85}$$

where *processes* is defined in Equation 7.6 and the function *datastores-modified* defines the set of all datastores modified by a process and is defined as

$$\begin{aligned}&p \in processes(\mathcal{C})\\&\quad \Rightarrow datastores\text{-}modified(\mathcal{C},p)\\&\qquad = \{d \mid d \in \mathcal{C}.Datastore \wedge f \in d.InFlows \wedge o \in all\text{-}outflows(p) \wedge f = o\}\end{aligned} \tag{7.28}$$

where *all-outflows* is a function that produces the set of all output dataflows from p or any of its subprocesses

$$\textit{all-outflows}(p) = \{f | f \in p.OutFlows\} \cup \{\textit{all-outflows}(p_1) | p_1 \in processes(p)\} \tag{7.29}$$

and *dataflow-domain* is a function that returns the sequence of input dataflow types as define below.

$$\textit{dataflow-domain}(p) = [flowtype(f) \mid f \in p.InFlows] \tag{7.30}$$

The function *flowtype* used in Equation 7.30 returns the datastore name or the dataflow type as defined below.

$$\begin{aligned}
&(d \in \mathcal{C}.Datastore \wedge f_1 \in d.InFlows \wedge f_1 = f \Rightarrow flowtype(f) = d.Name) \\
&\quad \wedge ((\neg\exists\ (d, f, f_1)\ d \in \mathcal{C}.Datastore \wedge f_1 \in d.InFlows \wedge f_1 = f) \Rightarrow \\
&\qquad (defined?(f.type) \Rightarrow flowtype(f) = f.type \\
&\qquad \wedge undefined?(f.type) \Rightarrow type(f) = f.Name))
\end{aligned} \tag{7.31}$$

The function *datastores-sort* returns either the class sort of $\mathbb{C}$, the name of the datastore accessed by p, or an empty string if no datastores are accessed as defined below.

$$\begin{aligned}
&(size(\textit{datastores-modified}(\mathcal{C},p)) = 1 \wedge d \in \textit{datastores-modified}(\mathcal{C},p) \\
&\Rightarrow \textit{datastore-sort}(\mathcal{C},p) = d.Name) \\
&\wedge\ (size(\textit{datastores-modified}(\mathcal{C},p)) > 1 \\
&\quad \Rightarrow \textit{datastore-sort}(\mathcal{C},p) = \mathcal{C}.Name) \\
&\wedge\ (size(\textit{datastores-modified}(\mathcal{C},p)) = 0 \Rightarrow \\
&\quad (size(\textit{datastores-accessed}(\mathcal{C},p)) > 1 \\
&\quad \Rightarrow \textit{datastore-sort}(\mathcal{C},p) = \mathcal{C}.Name) \\
&\quad \wedge\ (size(\textit{datastores-accessed}(\mathcal{C},p)) = 1 \wedge d \in \textit{datastores-accessed}(\mathcal{C},p) \\
&\qquad \Rightarrow \textit{datastore-sort}(\mathcal{C},p) = d.Name) \\
&\quad \wedge\ (size(\textit{datastores-accessed}(\mathcal{C},p)) = 0 \Rightarrow \textit{datastore-sort}(\mathcal{C},p) = \text{``''}))
\end{aligned} \tag{7.32}$$

where *datastores-accessed* is a function that returns all datastores accessed by a process and is defined as

$$\begin{aligned}
&p \in processes(\mathcal{C}) \Rightarrow \\
&\quad \textit{datastores-accessed}(\mathcal{C},p) \\
&\qquad = \{d \mid d \in \mathcal{C}.Datastore \wedge f \in d.OutFlows \wedge o \in \textit{all-inflows}(p) \wedge f = o\}
\end{aligned} \tag{7.33}$$

where the function *all-inflows* is defined in Equation 7.34.

$$\textit{all-inflows}(p) = \{f | f \in p.InFlows \vee (f \in \textit{all-inflows}(p_1) \wedge p_1 \in processes(p))\} \tag{7.34}$$

If there are no output dataflows from process p or its subprocesses that are sent to datastores, then process p defines an operation signature. Again, if any input dataflows of any subprocesses of p is from a datastore, that datastore must also be included in the operation input parameters. If there are multiple subprocess dataflow that access datastores, then the format of the operation signature is shown below.

$$p.Name : \mathcal{C}.Name, type(i_1)...type(i_n) \rightarrow o_1.type...o_m.type$$

If there is only one subprocess dataflow, o_{sub} that accesses a datastore then the following operation signature is used.

$$p.Name : datasore\text{-}sort(\mathcal{C}, p), type(i_1)...type(i_n) \rightarrow o_1.type...o_m.type$$

where *datastore-sort* is defined in Equation 7.32 and $o_1.type...o_m.type$ are the output dataflow datatypes. The formal transformation is shown below.

$$\begin{aligned} & p \in processes(\mathcal{C}) \wedge size(datastores\text{-}modified(\mathcal{C}, p)) = 0 \\ & \quad \Rightarrow \langle p.Name, [datastore\text{-}sort(\mathcal{C}, p)] \parallel dataflow\text{-}domain(p), [\mathcal{C}.Name] \rangle \\ & \qquad\qquad \in \mathbb{C}.Operation \end{aligned} \qquad \text{(OMT-86)}$$

If a process has subprocesses, then those subprocesses and their associated dataflows define the composition of the process. This composition is defined axiomatically for each process, p, with subprocesses the following axiom is created.

$$p.Name(proc\text{-}domain(p)) = proc\text{-}range(p) \wedge implementing\text{-}axioms(p)$$

where *proc-domain(p)* and *proc-range(p)* are functions defining domain and range variables while *implementing-axioms(p)* is a function that creates a sub-axiom for each subprocess, p_1, of p as shown below.

$$proc\text{-}range(p1) = p.Name(proc\text{-}domain(p))$$

The functions *proc-domain(p)* and *proc-range(p)* generate signatures that are compatible with the operation and method signatures defined above; however, they insert variable names instead of datatypes according to the following rules.

$$
\begin{array}{l}
(\mathit{size}(\mathit{datastores\text{-}modified}(\mathcal{C},p)) > 0 \Rightarrow \mathit{proc\text{-}domain}(p) = \mathcal{C}.\mathit{Name}, i_1.\mathit{Name}...i_n.\mathit{Name}) \\
\wedge\ (\mathit{size}(\mathit{datastores\text{-}modified}(\mathcal{C},p)) = 0 \Rightarrow \\
\quad (\mathit{size}(\mathit{datastores\text{-}accessed}(\mathcal{C},p)) > 1 \Rightarrow \mathit{proc\text{-}domain}(p) = \mathcal{C}.\mathit{Name}, i_1.\mathit{Name}...i_n.\mathit{Name}) \\
\quad \wedge\ (\mathit{size}(\mathit{datastores\text{-}accessed}(\mathcal{C},p)) = 1 \\
\qquad \Rightarrow \mathit{proc\text{-}domain}(p) = \mathit{datasore\text{-}sort}(\mathcal{C},p), i_1.\mathit{Name}...i_n.\mathit{Name}))
\end{array}
\tag{7.35}
$$

where $i_1...i_n$ is the set of all input flow names in $p.InFlows$.

$$
\begin{array}{l}
(\mathit{size}(\mathit{datastores\text{-}modified}(\mathcal{C},p)) = 1 \Rightarrow \mathit{proc\text{-}range}(p) = \mathit{datastore\text{-}sort}(\mathcal{C},p)) \\
\wedge\ (\mathit{size}(\mathit{datastores\text{-}modified}(\mathcal{C},p)) > 0 \Rightarrow \mathit{proc\text{-}range}(p) = \mathcal{C}.\mathit{Name}) \\
\wedge\ (\mathit{size}(\mathit{datastores\text{-}modified}(\mathcal{C},p)) = 0 \Rightarrow \mathit{proc\text{-}range}(p) = o_1.\mathit{Name}...o_n.\mathit{Name})
\end{array}
\tag{7.36}
$$

where $o_1...o_n$ is the set of all input flow names in $p.OutFlows$.

Therefore, The formal transformation of the axiomatic definition of a process p with subprocesses is

$$
\begin{array}{l}
p \in \mathit{processes}(\mathcal{C}) \wedge \mathit{defined?}(p.\mathit{Process}) \\
\quad \Rightarrow \text{“}p.\mathit{Name}(\mathit{proc\text{-}domain}(\mathcal{C},p)) = \mathit{proc\text{-}range}(\mathcal{C},p) \wedge\ \mathit{implementing\text{-}axioms}(p)\text{”} \\
\qquad\qquad \in \mathbb{C}.\mathit{Axiom}
\end{array}
\tag{OMT-87}
$$

where *implementing-axioms* is defined as

$$
\begin{array}{l}
\mathit{implementing\text{-}axioms}(p) = \mathit{AND}(\{\text{“}\mathit{proc\text{-}range}(p_1) = p_1.\mathit{Name}(\mathit{proc\text{-}domain}(p_1))\text{”} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid\ p_1 \in \mathit{processes}(p)\}
\end{array}
\tag{7.37}
$$

where AND denotes the logical conjunction of all input axioms.

7.4.2 Dataflows. Dataflows are not translated directly into components of an O-SLANG specification. They are used in defining the method/operation signature and subprocess axioms for GOMT processes. Dataflows correspond to the inputs and outputs of methods and operations.

7.4.3 Datastores. Datastores are not translated directly into components of an O-SLANG specification. They are used in defining the method/operation signature and subprocess axioms for GOMT processes. Datastores represent object classes and associations within an aggregate specification and are accessed via object-valued attributes referencing those classes and associations.

7.5 Additional Translations

Additional information may supplied as part of the GOMT Class, $\mathcal{C}$. In this research this additional information consists of the following items:

- set of axiomatic constraints
- set of operation definitions

Each of these items may or may not exist in a given class. These transformations are performed after $\mathcal{C}$ is converted to an O-SLANG specification, $\mathbb{C}$, based on the object model, functional model, and dynamic model.

7.5.1 Constraints.

Constraints are user supplied O-SLANG axioms that constrain the behavior of various components of the class. Therefore, each constraint in $\mathcal{C}$ is translated directly to axioms in the axiom block of $\mathbb{C}$ as shown below.

$$c \in \mathcal{C}.Axiom \Rightarrow c \in \mathbb{C}.Axiom \qquad \text{(OMT-88)}$$

7.5.2 Operations.

Operations in the GOMT AST can represent three different O-SLANG constructs: 1) a newly defined O-SLANG method, 2) a newly defined O-SLANG operation, or 3) the method definition of an action in the dynamic model. A GOMT Operation consists of the following items.

- name
- set of parameters (with a name and datatype)
- result
- definition
- boolean denoting whether it is abstract

An operation *result* is its output datatype. If the operation has no defined result, the operation is a method and the output datatype is the class sort of $\mathbb{C}$. Each GOMT operation, o, defines a signature. If the result is the class sort of $\mathbb{C}$ or *result* is not defined, then the operation defines a method signature, as shown below, which is added to the method block of $\mathbb{C}$.

$$o.Name : \mathcal{C}.Name, p_1...p_n \rightarrow \mathcal{C}.Name$$

where $p_1...p_n$ are parameter datatypes from the parameter set of o. The formal definition of these translations is given in Rule OMT-89 below.

$$\begin{aligned} & o \in \mathcal{C}.GOMT\text{-}Op \wedge (undefined?(o.Result) \vee o.Result = \mathcal{C}.Name) \\ & \quad \Rightarrow \langle o.Name, [\mathcal{C}.Name] \parallel domain(o), [\mathcal{C}.Name] \rangle \in \mathbb{C}.Method \end{aligned} \tag{OMT-89}$$

where *domain* is defined in Equation 7.4.

If the result of o is a datatype other than the class sort of $\mathbb{C}$ then o defines an operation whose signature is placed in the operations block of $\mathbb{C}$.

$$o.Name : \mathcal{C}.Name, p_1...p_n \rightarrow o.result$$

This translation is shown formally in Rule OMT-90.

$$\begin{aligned} & o \in \mathcal{C}.GOMT\text{-}Op \wedge defined?(o.Result) \wedge o.Result \neq \mathcal{C}.Name \\ & \quad \Rightarrow \langle o.Name, [\mathcal{C}.Name] \parallel domain(o), [o.Result] \rangle \in \mathbb{C}.Operation \end{aligned} \tag{OMT-90}$$

The definition of an operation, o, is a set of user supplied axioms in O-SLANG syntax. It is assumed that the axioms are syntactically valid and correctly define the operation. The axioms in the definition of o translate directly into axioms in the axiom block of $\mathbb{C}$ as shown below.

$$c \in \mathcal{C}.GOMT\text{-}Op.Definition \Rightarrow c \in \mathbb{C}.Axiom \tag{OMT-91}$$

7.5.3 Imports. The specification import statements are used to include external specifications in the definition of the current specification. These imported specifications may either be aggregate specifications, superclass specifications, or specifications defining the sorts used for attributes, parameters, qualifiers, or operation results. The following rule defines exactly which specifications must be included in a class specification.

$$\begin{aligned}&\mathbb{C} \in \mathit{O\text{-}Slang\text{-}DomainTheory} \Rightarrow \\ &\quad \mathbb{C}.\mathit{Import} = \mathbb{C}.\mathit{Import} \cup (\mathit{class\text{-}imports}(\mathbb{C}) \setminus \mathit{imports}(\mathbb{C}))\end{aligned} \qquad \text{(OMT-92)}$$

where *class-imports* is a function that collects the references to all external specifications within $\mathbb{C}$. The function *imports* determines which specifications are already imported into $\mathbb{C}$ through specifications already in $\mathbb{C}.\mathit{Import}$ by Rules OMT-3, OMT-7, OMT-20, and OMT-80. Therefore, the set difference between *class-imports(*$\mathbb{C}$*)* and *imports(*$\mathbb{C}$*)* is a set of specification names that must be included in $\mathbb{C}.\mathit{Import}$. The definition of *class-imports* and *imports* are given below in Equations 7.38 and 7.8.

$$\begin{aligned}\mathit{class\text{-}imports}(c) \quad = \quad & \{s \mid s \in c.\mathit{Operation.Domain} - \mathit{Ident} \\ & \| \; c.\mathit{Operation.Range\text{-}Ident} \\ & \| \; c.\mathit{Attribute.Range\text{-}Ident} \\ & \| \; c.\mathit{Method.Domain\text{-}Ident} \\ & \| \; c.\mathit{Method.Range\text{-}Ident} \\ & \| \; c.\mathit{Event.Domain\text{-}Ident} \\ & \| \; c.\mathit{Event.Range\text{-}Ident}\}\end{aligned} \qquad (7.38)$$

7.6 Translation Correctness

In this section, I show that the translation of the object, functional, and dynamic models as defined in Section 7.2, Section 7.3, and Section 7.4 are correct with respect to the formal semantics defined in Chapter V. I prove this by showing that the diagram in Figure 7.1 composes. That is that for each OMT model, the mapping defined by the formal semantics, φ, is equivalent to the composition of the translation of the GOMT model to O-SLANG by τ followed by the mapping from O-SLANG to the formal semantics, ω as defined in Equation 7.39.

$$\forall(D : \mathit{GOMT\text{-}DomainTheory}) \;\; \varphi(D) = \omega(\tau(D)) \qquad (7.39)$$

A second useful property to prove about τ would be to show that the inverse transformation τ^{-1}, results in a domain theory equivalent to the original (i.e. $D \equiv \tau^{-1}(\tau(D))$). While this would

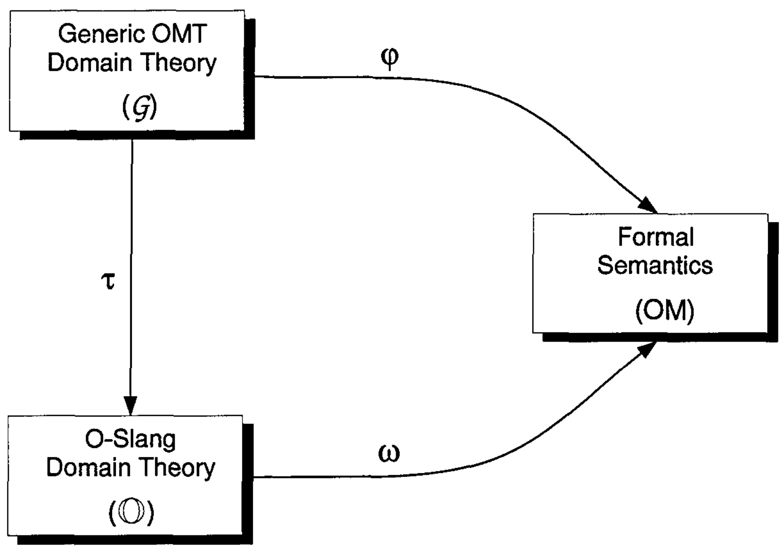

Figure 7.1 OMT Translation Composition

not help prove correctness, it would show the bijective nature of τ as well as its completeness. Unfortunately, the early design decision to have only a single axiom block per O-SLANG specification eliminates this possibility. Because a user may enter free-form axioms to define operations and class constraints, the ability to determine where O-SLANG axioms were generated (i.e., manually or automatically from the functional or dynamic model) is impaired. A fairly simple modification to the O-SLANG syntax and AST (along with the associated transformations) could solve this problem allowing τ to be bijective. However, due to time constraints, these modifications were not implemented in this research.

7.6.1 Object Model Correctness. In this section, I show that the translation of the GOMT object model as defined in Section 7.2 is correct with respect to the object model semantics defined in Section 5.3.3. I start by defining a mapping φ from the GOMT object model to the formal semantics of the restricted OMT object model as defined in Section 5.3.

Given the formal semantics defined in Section 5.3, an object model O consists of a set of specifications, S where a specification may consist of five items:

1. **Name**. The *name* of S – a single symbol.

2. **Imports.** The *imports* of S are a set of specification names.

3. **Sorts.** The *sorts* of S are a set of symbols.

4. **Operations.** *Operations* of S are defined as a tuple, $\langle name, domain, range \rangle$, where *domain* and *range* are a sequence of sorts.

5. **Axioms.** The *axioms* are a set of first-order logical statements defined over the sorts and operations defined in S, or in the specifications imported by S. I assume all axioms are in O-SLANG syntax.

For the definition of φ, τ, and ω, I assume that only the object model is defined and that no methods or operations are defined manually by the user. I also assume that all associations are binary and that there are no qualified associations or aggregates. The binary association assumption is made to simplify the mappings and proofs although the results can be extended to include higher-order association as well. Qualified associations and aggregations are not included since they were not included in the object model semantics as defined by Bourdeau and Cheng (14) and, while interesting, add little to the semantics. Finally, I assume that all attributes are defined with explicit datatypes. While not necessary, it simplifies the mapping definitions and the proof.

Definition 7.6.1 φ. *The mapping φ from the object model of a class, C, in a GOMT domain theory to an object model, OM, is defined in Equations 7.40 through 7.55. In these equations the function* comp-pred *(Equation 7.56) defines the the predicate name by returning either $HAS\text{-}PART$ or c.Role depending on whether c.Role is defined and* mult-subaxiom *(Equation 7.57) defines the sub-axioms of each range in a* Specified *multiplicity.*

$$\begin{aligned}&C \in GOMT\text{-}DomainTheory.GOMT\text{-}Class \ \wedge\ C.Name = n\\ &\quad \Leftrightarrow S_C \in OM \ \wedge\ S_C.Name = n \ \wedge\ n \in S_C.Sorts\end{aligned} \tag{7.40}$$

$$\begin{aligned}&a \in C.Attribute \Leftrightarrow\\ &\quad \langle a.Name, [C.Name], [a.datatype]\rangle \in S_C.Operations \ \wedge\ a.datatype \in S_C.Imports\end{aligned} \tag{7.41}$$

$$c \in C.Connection \Leftrightarrow \langle comp\text{-}pred(c), [C.Name, c.Name], [Boolean]\rangle \in S_C.Operations \tag{7.42}$$

$$c \in \mathcal{C}.Connection \;\wedge\; c.Mult = One \;\Leftrightarrow \\ \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = 1\text{”} \in S_C.Axioms) \tag{7.43}$$

$$c \in \mathcal{C}.Connection \;\wedge\; c.Mult = Many \Leftrightarrow \\ \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) \geq 0\text{”} \in S_C.Axioms) \tag{7.44}$$

$$c \in \mathcal{C}.Connection \;\wedge\; c.Mult = Plus \;\Leftrightarrow \\ \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) \geq c.Plus.integer\text{”} \in S_C.Axioms) \tag{7.45}$$

$$c \in \mathcal{C}.Connection \;\wedge\; c.Mult = Optional \;\Leftrightarrow \\ \text{“}X \in c.Name \Rightarrow (SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = 0 \\ \vee\; SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = 1)\text{”} \in S_C.Axioms) \tag{7.46}$$

$$c \in \mathcal{C}.Connection \;\wedge\; c.Mult = Specified \;\wedge\; sort = c.Name \;\wedge\; pred = comp\text{-}pred(c) \\ \wedge\; s \in c.Mult \;\wedge\; v_1 = s.value1 \;\wedge\; v_2 = s.value2 \Leftrightarrow \\ (ax \in S_C.Axioms \wedge\; mult\text{-}subaxiom(sort, pred, v_1, v_2) \sqsubset ax \\ \wedge\; ax = OR(\{mult\text{-}subaxiom(sort, pred, x_1, x_2) \mid \; c \in \mathcal{C}.Connection \\ \wedge\; c.Mult = Specified \;\wedge\; sort = c.Name \;\wedge\; pred = comp\text{-}pred(c) \\ \wedge\; s \in c.Mult \;\wedge\; v_1 = s.value1 \;\wedge\; v_2 = s.value2 \tag{7.47}$$

$$c \in \mathcal{C}.Superclass \Leftrightarrow \langle simulates, [\mathcal{C}.Name], [c]\rangle \in S_C.Operations \;\wedge\; c \in S_C.Imports \tag{7.48}$$

$$\mathcal{A} \in GOMT\text{-}DomainTheory.Assoc \;\wedge\; \mathcal{A}.Name = n \Leftrightarrow \\ S_A \in OM \;\wedge\; S_A.Name = n \;\wedge\; n \in S_A.Sorts \tag{7.49}$$

$$dom = [c.Name \mid c \in \mathcal{A}.Connection] \Leftrightarrow \\ (\langle \mathcal{C}.Name, dom, [Boolean]\rangle \in S_A.Operations \;\wedge\; c \in dom \Rightarrow c \in S_A.Imports) \tag{7.50}$$

$$c \in \mathcal{A}.Connection \;\wedge\; c.Mult = One \Leftrightarrow \\ \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) = 1\text{”} \in S_A.Axioms \tag{7.51}$$

$$c \in \mathcal{A}.Connection \;\wedge\; c.Mult = Many \Leftrightarrow \\ \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq 0\text{”} \in S_A.Axioms \tag{7.52}$$

$$c \in \mathcal{A}.Connection \;\wedge\; c.Mult = Plus \Leftrightarrow \\ \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq c.Plus.integer\text{”} \in S_A.Axioms \tag{7.53}$$

$$c \in \mathcal{A}.Connection \;\wedge\; c.Mult = Optional \Leftrightarrow \\ \text{“}X \in c.Name \Rightarrow (SIZE(\{Y \mid S_A.Name(X,Y)\}) = 0 \\ \vee\; SIZE(\{Y \mid S_A.Name(X,Y)\}) = 1)\text{”} \in S_A.Axioms \tag{7.54}$$

$$c \in \mathcal{C}.Connection \;\wedge\; c.Mult = Specified \;\wedge\; sort = c.Name \;\wedge\; pred = \mathcal{C}.Name \\ \wedge\; s \in c.Mult \;\wedge\; v_1 = s.value1 \;\wedge\; v_2 = s.value2 \Leftrightarrow \\ (ax \in S_C.Axioms \wedge\; mult\text{-}subaxiom(sort, predv_1, v_2) \sqsubset ax \\ \wedge\; ax = OR(\{mult\text{-}subaxiom(sort, pred, x_1, x_2) \mid \; c \in \mathcal{C}.Connection \\ \wedge\; c.Mult = Specified \;\wedge\; sort = c.Name \;\wedge\; pred = \mathcal{C}.Name \\ \wedge\; s \in c.Mult \;\wedge\; v_1 = s.value1 \;\wedge\; v_2 = s.value2 \tag{7.55}$$

$$defined?(c.Role) \Rightarrow comp\text{-}pred(c) = c.Role \\ undefined?(c.Role) \Rightarrow comp\text{-}pred(c) = HAS\text{-}PART \tag{7.56}$$

$$mult\text{-}subaxiom(sort, pred, v_1, v_2) = ax \\ \wedge\; (defined?(v_2) \Leftrightarrow ax = \text{“}X \in sort \Rightarrow (SIZE(\{Y \mid pred(X,Y)\}) \geq v_1 \\ \wedge\; SIZE(\{T \mid pred(X,Y)\}) \leq v_2)\text{”}) \\ \wedge\; (undefined?(v_2) \Leftrightarrow ax = \text{“}X \in sort \Rightarrow (SIZE(\{Y \mid pred(X,Y)\}) = v_1)\text{”}) \tag{7.57}$$

Definition 7.6.2 ω. *The mapping ω from an* O-SLANG *class, $\mathbb{C}$, to an object model, OM', is defined in Equations 7.58 through 7.74. In these equations the function* om-pred *(Equation 7.75) converts the component attribute name in* O-SLANG *to the appropriate predicate name in OM and the function* sort-of *(Equation 7.76) finds the class sort of the class referenced by an object-valued attribute.*

$$\begin{aligned}&\mathbb{C} \in \textit{O-Slang-DomainTheory.Class} \ \wedge \ \mathbb{C}.\textit{Name} = n \Leftrightarrow \\ &\quad S_C \in OM' \ \wedge \ S_C.\textit{Name} = n \ \wedge \ n \in S_C.\textit{Sorts}\end{aligned} \tag{7.58}$$

$$\begin{aligned}&a \in \mathbb{C}.\textit{Attribute} \ \wedge \ a.\textit{Range-Ident}(1) \neq \textit{x-CLASS} \ \wedge \ a.\textit{Range-Ident}(1) \neq \textit{x-ASSOC} \Leftrightarrow \\ &\quad a \in S_C.\textit{Operations} \ \wedge \ a.\textit{Range-Ident}(1) \in S_C.\textit{Imports}\end{aligned} \tag{7.59}$$

$$\begin{aligned}&a \in \mathbb{C}.\textit{Operation} \ \wedge \ a.\textit{Range-Ident} = [\mathbb{C}.\textit{Name}] \ \wedge \ a.\textit{Name} \neq \textit{ATTR-EQUAL} \Leftrightarrow \\ &\quad a \in S_C.\textit{Operations} \ \wedge \ (i \in a.\textit{Range-Ident} \Rightarrow i \in S_C.\textit{Imports})\end{aligned} \tag{7.60}$$

$$\begin{aligned}&c \in \mathbb{C}.\textit{attribute} \wedge c.\textit{Range-Ident} = [\textit{name-CLASS}] \Leftrightarrow \\ &\quad \langle \textit{om-pred}(c.\textit{Name}), [\mathbb{C}.\textit{Name}, \textit{name}], [\textit{Boolean}] \rangle \in S_C.\textit{Operations}\end{aligned} \tag{7.61}$$

$$\begin{aligned}&\text{“}SIZE(\textit{name}(X)) = 1\text{”} \in \mathbb{C}.\textit{Axiom} \Leftrightarrow \\ &\quad \text{“}X \in \textit{sort-of}(\textit{name}) \Rightarrow SIZE(\{Y \mid \textit{om-pred}(\textit{name})(X,Y)\}) = 1\text{”} \in S_C.\textit{Axioms})\end{aligned} \tag{7.62}$$

$$\begin{aligned}&\text{“}SIZE(\textit{name}(X)) \geq 0\text{”} \in \mathbb{C}.\textit{Axiom} \Leftrightarrow \\ &\quad \text{“}X \in \textit{sort-of}(\textit{name}) \Rightarrow SIZE(\{Y \mid \textit{om-pred}(\textit{name})(X,Y)\}) \geq 0\text{”} \in S_C.\textit{Axioms})\end{aligned} \tag{7.63}$$

$$\begin{aligned}&\text{“}SIZE(\textit{name}(X)) \geq x\text{”} \in \mathbb{C}.\textit{Axiom} \ \wedge \ x \neq 0 \Leftrightarrow \\ &\quad \text{“}X \in \textit{sort-of}(\textit{name}) \Rightarrow SIZE(\{Y \mid \textit{om-pred}(\textit{name})(X,Y)\}) \geq x\text{”} \in S_C.\textit{Axioms})\end{aligned} \tag{7.64}$$

$$\begin{aligned}&\text{“}SIZE(\textit{name}(X)) = 1 \vee SIZE(\textit{name}(X)) = 0\text{”} \in \mathbb{C}.\textit{Axiom} \Leftrightarrow \\ &\quad \text{“}X \in \textit{sort-of}(\textit{name}) \Rightarrow \text{“}(SIZE(\{Y \mid \textit{om-pred}(\textit{name})(X,Y)\}) = 0 \\ &\qquad \vee \ SIZE(\{Y \mid \textit{om-pred}(\textit{name})(X,Y)\}) = 1)\text{”} \in S_C.\textit{Axioms})\end{aligned} \tag{7.65}$$

$$\begin{aligned}&a \in \mathbb{A}.\textit{Axioms} \\ &\wedge \ (\text{“}(SIZE(\textit{name}(X)) \geq n \wedge SIZE(\textit{name}(X)) \leq m)\text{”} \sqsubset a \\ &\quad \vee \ (\text{“}SIZE(\textit{name}(X)) = n\text{”} \sqsubset a \ \wedge \ n > 1)) \\ &\Leftrightarrow (ax \in S_A.\textit{Axioms} \\ &\quad \wedge \ ((\text{“}(SIZE(\textit{name}(X)) \geq n \wedge SIZE(\textit{name}(X)) \leq m)\text{”} \sqsubset a) \\ &\qquad \Leftrightarrow (\text{“}X \in \textit{sort-of}(\textit{name}) \Rightarrow (SIZE(\{Y \mid \textit{om-pred}(\textit{name})(X,Y)\}) \geq n \\ &\qquad\qquad \wedge \ SIZE(\{Y \mid \textit{om-pred}(\textit{name})(X,Y)\}) \leq m)\text{”} \sqsubset ax)) \\ &\quad \wedge \ (\text{“}SIZE(\textit{name}(X)) = n\text{”} \sqsubset a) \\ &\qquad \Leftrightarrow \text{“}X \in \textit{sort-of}(\textit{name}) \\ &\qquad\quad \Rightarrow (SIZE(\{Y \mid \textit{om-pred}(\textit{name})(X,Y)\})) = n\text{”} \sqsubset ax)) \in S_A.\textit{Axioms}\end{aligned} \tag{7.66}$$

$$\begin{aligned}&c \in \mathbb{C}.\textit{Class-Sort.Inherited-Sort-Id} \\ &\quad \Leftrightarrow \langle \textit{simulates}, [\mathbb{C}.\textit{Name}], [c] \rangle \in S_C.\textit{Operations} \ \wedge \ c \in S_C.\textit{Imports}\end{aligned} \tag{7.67}$$

$$\begin{aligned}&\mathbb{A} \in \textit{O-Slang-DomainTheory.Association} \ \wedge \ \mathbb{A}.\textit{Name} = n \Leftrightarrow \\ &\quad S_A \in OM \ \wedge \ S_A.\textit{Name} = n \ \wedge \ n \in S_A.\textit{Sorts}\end{aligned} \tag{7.68}$$

$$\begin{aligned}&\textit{dom} = [a.\textit{Domain-Ident}(2) \mid a \in \mathbb{A}.\textit{Operation} \ \wedge \ a.\textit{Name} = \textit{IMAGE}] \Leftrightarrow \\ &\quad (\langle \mathbb{C}.\textit{Name}, \textit{dom}, [\textit{Boolean}] \rangle \in S_A.\textit{Operations} \ \wedge \ c \in \textit{dom} \Rightarrow c \in S_A.\textit{Imports})\end{aligned} \tag{7.69}$$

$$\begin{aligned}&\text{“}X \in \textit{sort} \Rightarrow SIZE(IMAGE(A,X)) = 1\text{”} \in \mathbb{A}.\textit{Axioms} \Leftrightarrow \\ &\quad \text{“}X \in \textit{sort} \Rightarrow SIZE(\{Y \mid S_A.\textit{Name}(X,Y)\}) = 1\text{”} \in S_A.\textit{Axioms}\end{aligned} \tag{7.70}$$

$$\begin{aligned}&\text{“}X \in sort \Rightarrow SIZE(IMAGE(A,X)) \geq 0\text{”} \in \mathbb{A}.Axioms \Leftrightarrow\\&\quad \text{“}X \in sort \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq 0\text{”} \in S_A.Axioms\end{aligned} \tag{7.71}$$

$$\begin{aligned}&\text{“}X \in sort \Rightarrow SIZE(IMAGE(A,X)) \geq x\text{”} \in \mathbb{A}.Axioms \Leftrightarrow\\&\quad \text{“}X \in sort \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq x\text{”} \in S_A.Axioms\end{aligned} \tag{7.72}$$

$$\begin{aligned}&\text{“}X \in sort \Rightarrow (SIZE(IMAGE(A,X)) = 0 \vee SIZE(IMAGE(A,X)) = 1)\text{”} \in \mathbb{A}.Axioms \Leftrightarrow\\&\quad \text{“}X \in sort \Rightarrow (SIZE(\{Y \mid S_A.Name(X,Y)\}) = 0\\&\qquad \vee\ SIZE(\{Y \mid S_A.Name(X,Y)\}) = 1)\text{”} \in S_A.Axioms\end{aligned} \tag{7.73}$$

$$\begin{aligned}&a \in \mathbb{A}.Axioms\\&\wedge\ (\text{“}X \in sort \wedge (SIZE(IMAGE(A,X)) \geq n \wedge SIZE(IMAGE(A,X)) \leq m)\text{”} \sqsubset a\\&\quad \vee\ (\text{“}X \in sort \wedge SIZE(IMAGE(A,X)) = n\text{”} \sqsubset a\ \wedge\ n > 1))\\&\Leftrightarrow (ax \in S_A.Axioms\\&\quad \wedge\ ((\text{“}X \in sort \wedge (SIZE(IMAGE(A,X)) \geq n \wedge SIZE(IMAGE(A,X)) \leq m)\text{”} \sqsubset a)\\&\qquad \Leftrightarrow (\text{“}X \in sort \wedge (SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq n\\&\qquad\qquad \wedge\ SIZE(\{Y \mid S_A.Name(X,Y)\}) \leq m)\text{”} \sqsubset ax))\\&\quad \wedge\ (\text{“}X \in sort \wedge (SIZE(IMAGE(A,X)) = n\text{”} \sqsubset a)\\&\qquad \Leftrightarrow \text{“}X \in sort \wedge (SIZE(\{Y \mid S_A.Name(X,Y)\})) = n\text{”} \sqsubset ax)) \in S_A.Axioms\end{aligned} \tag{7.74}$$

The function *om-pred* is defined in Equation 7.75 below. Basically, if the component attribute name has a *-OBJ* ending, there was no role name assigned to the component and thus the default *HAS-PART* predicate name is used. If the attribute name does not have an *-OBJ* ending, then the attribute name is the role name and no transformation is made.

$$\begin{aligned}&c = component\text{-}OBJ \Leftrightarrow om\text{-}pred(c) = HAS\text{-}PART\\&c \neq component\text{-}OBJ \Leftrightarrow om\text{-}pred(c) = c\end{aligned} \tag{7.75}$$

The function *sort-of* finds the class sort of the class referenced by an object-valued attribute since by OMT-5, aggregate components in a GOMT class generate object-valued attributes that are named either by the class name or role. The definition of *sort-of* is given in Equation 7.76.

$$a \in \mathbb{C}.Attribute\ \wedge\ a.Name = c\ \wedge\ [sort\text{-}CLASS] = a.Range\text{-}Ident \Rightarrow sort\text{-}of(c) = sort \tag{7.76}$$

7.6.2 Object Model Correctness Theorem. In this section, Theorem VII.1 establishes the correctness of the object model translation with respect to the object model semantics established in Section 5.3.3.

Theorem VII.1 *Given a valid GOMT domain theory class C with a well defined object model, the translation to* O-SLANG *as defined by τ in Section 7.2 preserves the semantics of the object model as defined in Definition 5.3.1.*

Proof. See Appendix F

7.6.3 Dynamic Model Correctness. In this section, I show that the translation of the GOMT dynamic model as defined in Section 7.3 is correct with respect to the dynamic model semantics defined in Section 5.4. I start by defining a mapping φ from the GOMT dynamic model to the formal semantics of the restricted OMT dynamic model.

For the definition of φ, τ, and ω, I assume that substates and concurrent states have already been translated into this simple automata as discussed in Section 5.4.4. With this assumption, φ is defined as

Definition 7.6.3 φ. *The mapping φ from the dynamic model of a class, C, in a GOMT domain theory to a statechart, $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ is defined as*

$$Q = \{s.Name \mid s \in C.State\}$$

$$\Sigma = \{t.Name \mid t \in C.Transition\}$$

$$\begin{aligned}\Delta = &\{msig(t.Action) \mid t \in C.Transition \land t.Action.Name \neq SEND\}\\ &\cup \{esig(t.Action.Action) \mid t \in C.Transition \land t.Action.Name = SEND\}\end{aligned}$$

$$\delta(q, \sigma) = \begin{cases} t.ToState & if\ \exists\ (t \in C.Transition)\ such\ that\ t.FromState = q \\ & \qquad \land\ t.Name = \sigma \land t.Axiom\ holds \\ q & otherwise \end{cases}$$

$$\lambda(q, \sigma) = \begin{cases} sig(t.Action) & if\ \exists\ (t \in C.Transition)\ such\ that\ defined?(t.Action) \\ & \qquad \land\ t.Name = \sigma \land t.Axiom\ holds \\ & \qquad \land\ \land t.FromState = q \\ \{\} & otherwise \end{cases}$$

$$q_0 = Initial\text{-}State$$

where the function *msig* defines the functional signature of a method action as defined in Equation 7.77.

$$msig(a) = \langle a.Name, [\mathcal{C}.Name] \parallel domain(a.Parameter), [\mathcal{C}.Name]\rangle \tag{7.77}$$

The function *esig* defines the functional signature of an event send action as defined in Equation 7.78.

$$esig(a) = \langle a.Name, [a.Name\text{-}SORT] \parallel domain(a.Parameter), [a.Name\text{-}SORT]\rangle \tag{7.78}$$

The function *sig* defines the signature of each method or event send action as defined in Equation 7.78.

$$\begin{aligned} sig(a) = \{s \mid\ & t \in a \wedge undefined?(t.Action) \Rightarrow s = msig(t) \\ & \wedge\ defined?(t.Action) \Rightarrow s = esig(t.Action)\} \end{aligned} \tag{7.79}$$

Definition 7.6.4 ω. *The mapping ω from an* O-SLANG *class,* $\mathbb{C}$, *to a statechart,* $M' = (Q', \Sigma', \Delta', \delta', \lambda', q_0')$ *is defined as*

$$Q' = \{s.Name \mid s \in \mathbb{C}.State\} \cup \{Initial\text{-}State\text{-}Marker\}$$

$$\Sigma' = \{e.Name \mid e \in \mathbb{C}.Event\}$$

$$\begin{aligned} \Delta' = \{m \mid\ & m \in \mathbb{C}.Method \wedge e \in \mathbb{C}.Event \wedge a \in \mathbb{C}.Axiom \\ & \wedge\ a = \text{“}\mathbb{C}.Name\text{-}STATE(x) = q... \Rightarrow \mathbb{C}.Name\text{-}STATE(\sigma(...)) = q_2\ ...\text{”} \\ & \wedge\ \text{“}ATTR\text{-}EQUAL(e.Name(...), m.Name(...))\text{”} \sqsubset a\} \\ \cup \{e \mid\ & e.Name \in \mathbb{C}.Import \wedge \mathbb{C}_{\mathbb{E}} \in O\text{-}Slang\text{-}DomainTheory \\ & \wedge\ \mathbb{C}_{\mathbb{E}}.Name = e.Name \wedge e \in \mathbb{C}_{\mathbb{E}}.Event\} \end{aligned}$$

$$\delta'(q,\sigma) = \begin{cases} q_2 & if\ ax \in \mathbb{C}.Axiom \\ & \quad \wedge\ ax = \text{“}\mathbb{C}.Name\text{-}STATE(x) = q \wedge guard \\ & \qquad \Rightarrow \mathbb{C}.Name\text{-}STATE(\sigma(...)) = q_2\ ...\text{”} \\ & \quad \wedge\ guard\ holds \\ q_2 & if\ q = Initial\text{-}State\text{-}Marker \wedge ax \in \mathbb{C}.Axiom \\ & \quad \wedge\ ax = \text{“}\mathbb{C}.Name\text{-}STATE(NEW\text{-}\mathbb{C}.Name(...)) = q_2\ ...\text{”} \\ & \quad \wedge\ guard\ holds \\ q & otherwise \end{cases}$$

$$\lambda'(q,\sigma) = \begin{cases} action\text{-}set(ax) & if\ ax \in \mathbb{C}.Axiom \\ & \quad \wedge\ ax = \text{“}\mathbb{C}.Name\text{-}STATE(x) = q \\ & \quad \wedge\ guard \Rightarrow \mathbb{C}.Name\text{-}STATE(\sigma(...)) = q_2\ ...\text{”} \\ & \quad \wedge\ guard\ holds \\ action\text{-}set(ax) & if\ q = Initial\text{-}State\text{-}Marker \wedge ax \in \mathbb{C}.Axiom \\ & \quad \wedge\ ax = \text{“}\mathbb{C}.Name\text{-}STATE(NEW\text{-}\mathbb{C}.Name(...)) = q_2\ ...\text{”} \\ & \quad \wedge\ guard\ holds \\ \{\} & otherwise \end{cases}$$

$$q'_0 = Initial\text{-}State$$

where $\sqsubset$ is the boolean-valued subsequence operator and ... matches zero or more characters in axiom.

$$\begin{aligned} action\text{-}set(a) = \{m \mid\ & m \in \mathbb{C}.Method \wedge\ \text{“}ATTR\text{-}EQUAL(\sigma(...), m.Name(...))\text{”} \sqsubset a\} \\ \cup \{e \mid\ & e.Name \in \mathbb{C}.Import \wedge \mathbb{C}_{\mathbb{E}} \in O\text{-}Slang\text{-}DomainTheory \\ & \wedge\ \mathbb{C}_{\mathbb{E}}.Name = e.Name \wedge e \in \mathbb{C}_{\mathbb{E}}.Event\} \\ & \text{“}e.Name\text{-}OBJ(\sigma(...)) = e.Name(e.Name\text{-}OBJ(x)...)\text{”} \sqsubset a\} \end{aligned} \tag{7.80}$$

7.6.4 Dynamic Model Correctness Theorem. In this section, Theorem VII.2 establishes the correctness of the dynamic model translation with respect to the dynamic model semantics established in Section 5.4.4.

Theorem VII.2 *Given a valid GOMT domain theory class C with a defined dynamic model, the translation to* O-SLANG *as defined by τ in Section 7.3 preserves the semantics of the dynamic model as defined in Definition 5.4.2.*

Proof. See Appendix F

7.6.5 Functional Model Correctness. In this section, I show that the translation of the GOMT functional model, as defined in Section 7.4, is correct with respect to the functional model semantics defined in Section 5.5.4. I start by defining a mapping φ from the GOMT functional model to the formal semantics of the restricted OMT functional model.

For the definition of φ, τ, and ω, I assume that all methods and operations are defined either as 1) actions from the GOMT dynamic model, or 2) processes from the OMT functional model. Furthermore, I assume that all actions defined in the dynamic model have a process definition in the functional model and that derived attributes (which result in operations) are not defined. While these assumptions are not critical to the result of the proof, it eliminates clutter caused by the definition of "default" functional models for methods and operations without explicit functional models. These assumptions imply that the required object "create" process is included as part of the functional model and is not created automatically by the default rule OMT-16.

Definition 7.6.5 φ. *The mapping φ from the functional model of a class, C, in a GOMT domain theory to a dataflow diagram, $D = (C, F, K, R)$ is defined as*

$$C = \{c.Name \mid c \in (proc(\mathcal{C}) \cup \mathcal{C}.DataStore \cup \{Extern\})\}$$

$$F = dfmerge(\mathcal{C}.DataFlow)$$

$$K = \{c.Name \mid c \in (proc(\mathcal{C}) \cup \mathcal{C}.DataStore)\}$$

$$\begin{aligned} R = \{\langle x, y\rangle \mid\ & (x, y \in dfmerge(\mathcal{C}.DataFlow) \wedge x.Target = y.Source \wedge x.Target \neq Extern) \\ & \vee (\langle x, z\rangle \in R \wedge \langle z, y\rangle \in R)\} \end{aligned}$$

where the function *proc* defines the set of all processes and subprocesses within the class $\mathcal{C}$ as defined below.

$$proc(c) = \{p \mid p \in c.Process \vee (p_1 \in proc(c) \wedge p \in p_1.Process)\} \qquad (7.81)$$

and the function *dfmerge* modifies all subprocess dataflows where it appears in the subdiagram that the source or target of the dataflow is external, when in fact the source or target is a process from a higher level diagram. In the functional model of Figure 7.2 the output c from process $P14$ defines a dataflow with an external target; however, in actuality, $P14$ produces output c for the higher-level process $P1$ whose target is process $P3$. The function *dfmerge* is defined below.

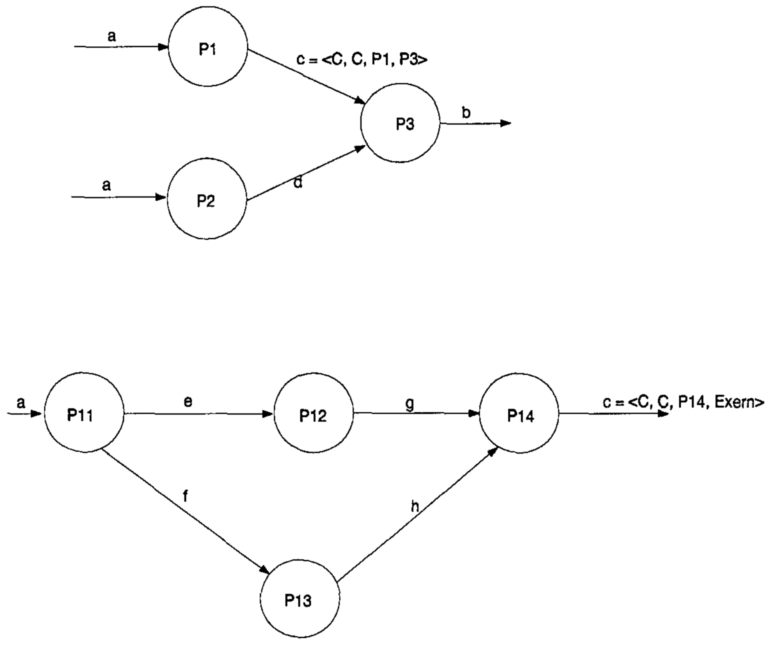

Figure 7.2 Dataflow Definitions

$$
\begin{aligned}
& d_1 \in d \wedge d_1.Target \neq Extern \wedge d_1.Source \neq Extern \Rightarrow d_1 \in dfmerge(d) \\
& d_1 \in d \wedge d_1.Target = Extern \wedge \neg(\exists\ (d_2)\ d_2 \in d \wedge \langle d_1.Name, d_1.Type\rangle = \langle d_2.Name, d_2.Type\rangle \\
& \quad \wedge d_2.Target \neq Extern) \Rightarrow d_1 \in dfmerge(d) \\
& d_1 \in d \wedge d_1.Source = Extern \wedge \neg(\exists\ (d_2)\ d_2 \in d \wedge \langle d_1.Name, d_1.Type\rangle = \langle d_2.Name, d_2.Type\rangle \\
& \quad \wedge d_2.Source \neq Extern) \Rightarrow d_1 \in dfmerge(d) \\
& d_1, d_2 \in d \wedge \langle d_1.Name, d_1.Type\rangle = \langle d_2.Name, d_2.Type\rangle \wedge d_1.Source = Extern \wedge d_2.Source \neq Extern \\
& \quad \Rightarrow \langle d_1.Name, d_1.Type, d_2.Source, d_1.Target\rangle \in dfmerge(d) \\
& d_1, d_2 \in d \wedge \langle d_1.Name, d_1.Type\rangle = \langle d_2.Name, d_2.Type\rangle \wedge d_1.Target = Extern \wedge d_2.Target \neq Extern \\
& \quad \Rightarrow \langle d_1.Name, d_1.Type, d_1.Source, d_2.Target\rangle \in dfmerge(d) \qquad (7.82)
\end{aligned}
$$

Definition 7.6.6 *ω. The mapping ω from an* O-SLANG *class, $\mathbb{C}$, to a dataflow diagram, $D' = (C', F', K', R')$ is defined as*

$$C' = \{c.Name \mid c \in \mathbb{C}.Method \vee c.Name \in datastores(\mathbb{C}) \\ \vee (c \in \mathbb{C}.Operation \wedge c.Name \neq IMAGE \wedge c.Name \neq ATTR\text{-}EQUAL)\} \\ \cup \{Extern\}$$

$$F' = dfmerge(\{f \mid f \in dataflows\text{-}of(a) \wedge a \in \mathbb{C}.Axiom\})$$

$$K' = C' \setminus \{Extern\}$$

$$R' = \{\langle x, y\rangle \mid (x, y \in dfmerge(\{f \mid a \in \mathbb{C}.Axiom \wedge f \in dataflows\text{-}of(a)\}) \\ \wedge x.Target = y.Source \wedge x.Target \neq Extern) \\ \vee (\langle x, z\rangle \in R' \wedge \langle z, y\rangle \in R')\}$$

$$\begin{aligned} datastores(c) = \{d \mid\ & ax \in c.Axiom \\ & \wedge (ax = \text{``}m(x, ...) = x_1 \wedge ...d(x_1) = m_2(d(x), ...)...\text{''} \\ & \quad \vee ax = \text{``}m(d(x), ...) = d(x_1) \wedge ...d(x_1) = m_2(d(x), ...)...\text{''} \\ & \quad \vee ax = \text{``}m(d(x), ...) = ... \wedge ... = o(d(x), ...)...\text{''})\} \end{aligned} \tag{7.83}$$

The function *dataflows-of* defines the seven valid mappings from O-SLANG functional axioms generated by Rule OMT-87 as shown in Table 7.1. In the *comment* region of Table 7.1, *illegal* means that a dataflow from the listed source to the listed target is illegal in standard dataflow diagrams. The comment *constrained* means that the particular combination of source and target are illegal by restriction of the functional model as described in Section 5.5.3.

$$\begin{aligned} & dataflows\text{-}of(a) = \\ & \quad \{a \neq \text{``}m(i_1...i_m) = o_1...o_n... \wedge r_1...r_n = sp(d_1...d_n)...\text{''} \Rightarrow dataflows\text{-}of(a) = \{\} \\ & \quad\ a = \text{``}m(i_1...i_m) = o_1...o_n... \wedge r_1...r_n = sp(d_1...d_n)...\text{''} \Rightarrow dataflows\text{-}of(a) = \\ & \qquad \{x \mid t \in top\text{-}level(a) \wedge op_1, op_2 \in operations(a) \\ & \qquad \wedge p_1 \in op_1.dom \wedge p_2 \in op_2.dom \wedge o \in op_1.ran \\ & \qquad \wedge (p_3 \in t.dom \Rightarrow x = \langle p_3, itype(p_3, t), Extern, t.Name\rangle) \\ & \qquad \wedge (p_3 \in t.ran \Rightarrow x = \langle p_3, otype(p_3, t), t.Name, Extern\rangle) \\ & \qquad \wedge (p_1 \notin datastores(\mathbb{C}) \wedge p_1 \in t.dom \Rightarrow x = \langle p_1, itype(p_1, op_1), Extern, op_1.Name\rangle) \\ & \qquad \wedge (p_1 \notin datastores(\mathbb{C}) \wedge p_1 \in p_2.ran \Rightarrow x = \langle p_1, itype(p_1, op_1), op_2.Name, op_1.Name\rangle) \\ & \qquad \wedge (o \notin datastores(\mathbb{C}) \wedge o \in t.ran \Rightarrow x = \langle o, otype(o, op_1), op_1.Name, Extern\rangle) \\ & \qquad \wedge (p_1 \in datastores(\mathbb{C}) \Rightarrow x = \langle dsname(p_1), dstype(p_1), p_1, op_1.Name\rangle) \\ & \qquad \wedge (o \in datastores(\mathbb{C}) \Rightarrow x = \langle dstype(o), dstype(o), op_1.Name, o\rangle)\} \end{aligned} \tag{7.84}$$

Table 7.1 Valid Dataflows

	Source	Target	Comment
1	Extern	Top Level Process	
2	Extern	Subprocess	
3	Extern	Datastore	Illegal
4	Extern	Extern	Illegal
5	Top Level Process	Top Level Process	Illegal
6	Top Level Process	Subprocess	Illegal
7	Top Level Process	Datastore	Constrained
8	Top Level Process	Extern	
9	Subprocess	Top Level Process	Illegal
10	Subprocess	Subprocess	
11	Subprocess	Datastore	
12	Subprocess	Extern	
13	Datastore	Top Level Process	Constrained
14	Datastore	Subprocess	
15	Datastore	Datastore	Illegal
16	Datastore	Extern	Illegal

where the functions *top-level* and *operations* return the tuple $\langle name, dom, ran \rangle$ for each method or operation used in the axiom. The function *top-level* returns only the signature of the method being defined while the function *operations* returns a set consisting of the signature of each of the methods/operations used to define the "top-level" operation.

The function *itype* takes an input parameter and operation/method name and returns the appropriate domain sort from the operation or method signature defined in $\mathbb{C}$. The *otype* function provides the same functionality for an output parameter.

$$\begin{aligned}&(\exists\ (op)\ op \in (\mathbb{C}.Operation \cup \mathbb{C}.Method) \land op.Name = f.Name \\ &\quad \land\ x = op.Domain\text{-}Ident(index(p_1, f.dom))) \\ &\Leftrightarrow itype(p_1, f) = x\end{aligned} \tag{7.85}$$

$$\begin{aligned}&(\exists\ (op)\ op \in (\mathbb{C}.Operation \cup \mathbb{C}.Method) \land op.Name = f.Name \\ &\quad \land\ y = op.Range\text{-}Ident(index(p_2, f.ran))) \\ &\Leftrightarrow otype(p_2, f) = y\end{aligned} \tag{7.86}$$

The function *dsname* returns the name of the class or association associated with the datastore. It is performed by finding the appropriate object-valued attribute declaration that references the class set or association as defined below.

$$\begin{aligned}&(\exists\ (a)\ a \in \mathbb{C}.\mathit{Attribute} \land a.\mathit{Name} = d\\&\ \land (a.\mathit{Range\text{-}Ident}(1) = \mathit{name\text{-}CLASS} \lor a.\mathit{Range\text{-}Ident}(1)\text{-}\mathit{ASSOC} = \mathit{name}))\\&\Leftrightarrow \mathit{dsname}(d) = \mathit{name}\end{aligned} \tag{7.87}$$

The function *dstype* returns the class set or association class sort. It is performed by finding the appropriate object-valued attribute declaration that references the class set or association as defined below.

$$\begin{aligned}&(\exists\ (a)\ a \in \mathbb{C}.\mathit{Attribute} \land a.\mathit{Name} = d \land \mathit{type} = a.\mathit{Range\text{-}Ident}(1)\\&\ \land (\mathit{type} = x\text{-}\mathit{CLASS} \lor \mathit{type}\text{-}\mathit{ASSOC} = a.\mathit{Name}))\\&\Leftrightarrow \mathit{dstype}(d) = \mathit{type}\end{aligned} \tag{7.88}$$

7.6.6 Functional Model Correctness Theorem. In this section, Theorem VII.3 establishes the correctness of the functional model translation with respect to the functional model semantics established in Section 5.5.4.

Theorem VII.3 *Given a valid GOMT domain theory class C with a defined functional model, the translation to* O-SLANG *as defined by τ in Section 7.4 preserves the semantics of the functional model as defined in Definition 5.5.1.*

Proof. See Appendix F

7.6.7 Communication Correctness Theorem. In this section, Theorem VII.4 establishes the correctness of the use of event theories and broadcast theories to establish the global broadcast communications model used by Rumbaugh. In essence, the broadcast communications model assumes that all events are broadcast to the system and are received by all objects whose dynamic model have the capability of receiving that event. Events are distinguished by their names. If class A sends an event E and class B receives an event E, by definition, there must be a communications path from A to B for event E.

Section 6.6 describes the use of *event* and *broadcast* theories to implement the global broadcast communications used in OMT. These theories are created by Rules OMT-40, OMT-76, OMT-77, OMT-78, and OMT-79 while their integration into aggregate diagrams is defined by Rules OMT-41 and OMT-80. The validity of these transformations with respect to the global broadcast communications model is shown in Theorem VII.4.

Theorem VII.4 *For each event E in a GOMT domain theory, there exists a valid communication path from each sending class A to each receiving class B in the* O-SLANG *domain theory aggregate.*

Proof. To be a valid communications path from class A to class B for event E, there must be an operation e_A in A and e_B in B such that e_A and e_B are in the same equivalence class in the domain theory aggregate which implies that the associated sorts in the domain and range of e_A and e_B are also in equivalence classes. These requirements are satisfied by unifying the operation signatures and sorts via the aggregate colimit operation. There are two unique cases to consider: a single receiving class and multiple receiving classes. Each of these will be handled separately.

- If a set of classes A_1 ... A_n sends event E to a class B then Rules OMT-76 – OMT-79 create an event theory ET which is imported into each class $A \in A_1$ ... A_n by Rule OMT-80. This importation of ET defines an identity morphism mapping each operation and sort of ET into A (n_1 in Rule OMT-39). If there is only one sending class (i.e., $n = 1$) then the morphism from ET to B (n_2 in Rule OMT-39), as defined by Rule OMT-39, causes the operations and sorts in ET, A, and B to be unified via the colimit operation in the aggregate class.

  If, however, there are more than a single sending class (i.e., $n > 1$), then by the definition of *comp-sends* and *comp-receives* in Equations 7.12 and 7.13, Rule OMT-39 defines morphisms from ET to each object in the aggregate that sends (A_1 ... A_n) or receives (B), or whose *components* sends or receives, event E. Since these morphisms are formed at each appropriate level of aggregation, the required unification of the event operation and sorts is accomplished as described above in some aggregate. In addition, since Rule OMT-42 creates a top-level domain theory aggregate for each domain, the events in all sending classes of E are eventually unified with the appropriate operations and sorts of all other sending and receiving classes in the domain theory.

- If a set of classes A_1 ... A_n sends event E to a set of classes B_1 ... B_m then Rules OMT-76 – OMT-79 create an event theory ET which is imported into each class $A \in A_1$ ... A_n by Rule OMT-80. When the lowest-level aggregate class is found whose components or subcomponents contain all sending and receiving classes (i.e., A_1 ... A_n plus B_1 ... B_m), Rule OMT-40 creates a broadcast theory BT and Rule OMT-41 defines the appropriate morphisms in the aggregate diagram such that the E event signatures in sending classes B_1 ... B_m are mapped to the appropriate operation in BT and all receive event signatures in receiving classes A_1 ... A_n are mapped to their unique event in BT defined by Rule OMT-40. Then the axioms defined in the broadcast theory ensure any that event received from a sending class B_i is translated into events received by each receiving class A_1 ... A_n. Finally, the colimit operation creates operation and sort equivalence classes based on the defined morphisms completing the unification process.

□

7.7 Summary

This chapter presented the transformation rules necessary to translate a GOMT AST into a valid O-SLANG AST. The rules were specified in accordance with the various OMT models: the object model, the dynamic model, and the functional model. Theorems were presented that show that the transformation rules defined in this chapter preserve the semantics of the restricted OMT models as defined in Chapter V. These rules are the basis of an automated transformation system described in Appendix D and are used in the next chapter to transform two graphically-based OMT specifications into formal theory-based domain models.

VIII. Feasibility Demonstration

8.1 Overview

This chapter demonstrates the feasibility of automatically translating graphically-based object-oriented specifications into theory-based domain specifications. The theories in this chapter were generated automatically by the prototype demonstration system described in Appendix D. An overview of each specification is given, followed by a description of its translation into theories.

8.2 Pump Domain

The *Pump* domain defines a specification for a simple gasoline pump and is a modified version of the case study found in (21). Each pump may have multiple hand guns, pump motors, and displays. Since I am modeling a domain, and not a system specification, the exact number of each item is not important. The domain object model is presented in Section 8.2.1, the dynamic model is discussed in Section 8.2.2, and the additional textual input is described in Section 8.2.3. There is no functional model in the pump domain. The complete O-SLANG specification of the domain model is shown in Appendix G.

8.2.1 Pump Domain Object Model. The *Pump* domain object model is shown in Figure 8.1. Basically, the specification models a type of pump, or more precisely, two types of pumps – *sophisticated* and *regular* – that are subtypes of a basic *pump* class. Each object in a pump class has a *Pump-ID* attribute and consists of zero or more *Gun-Holster-Assemblies*, *Clutch-Motor-Assemblies*, and *Displays.* Each *Gun-Holster-Assembly* object consists of a *Gun* object and a *Holster* object. While both the *Gun* and *Holster* classes appear to be simple classes with no attributes, a peek forward at their dynamic models (Figures 8.5 and 8.6) shows that their dynamic models are non-trivial and thus they are valid objects in the domain (i.e., they have *state* attributes). Likewise, a *Clutch-Motor-Assembly* object has exactly two components, *Motor* and *Clutch*, which also are defined by their dynamic models.

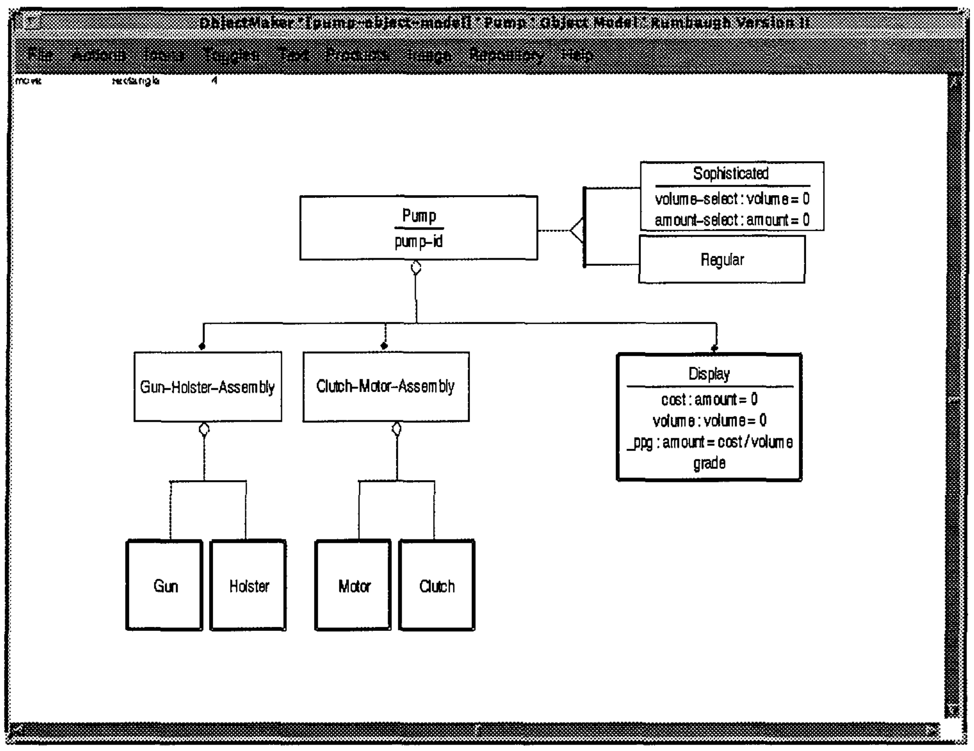

Figure 8.1 Pump Domain Object Model

The *Display* class has a more typical appearance than the *Gun*, *Holster*, *Motor*, or *Clutch* classes in that it has four attributes: *cost*, *volume*, *ppg*, and *grade.* The underscore in front of the *ppg* attribute denotes that it is a derived attribute (the usual derived attribute character "/" could not be captured appropriately by the tool) and its value for a given object is equal to the object's *cost* attribute value divided by the object's *volume* attribute value.

Each class in the object model generates a class and class set specification. Since this translation is the same for each class, I only present one such translation in detail, the *Display* class. The O-SLANG translations for the remainder of the classes are shown in Appendix G. The *Display* class specification, as generated strictly from the object model, is shown below. The specification itself, along with the class sort, is created by Rule OMT-1. The normal attributes *cost*, *volume*, and *grade* are translated into functions over the class sort as defined by Rule OMT-13 while the derived attribute *ppg* is defined by the user supplied axiom and is generated by Rule OMT-14. Since there are normal attributes defined for the *Display* class, Rule OMT-15 requires that an *attr-equal* operation be defined. The axiom is also automatically generated based on the normal attributes of the specification. Finally, the specification names *grade*, *amount*, and *volume* are included in the imports block of the specification by Rule OMT-92.

```
class DISPLAY is
    class-sort DISPLAY
    import GRADE, AMOUNT, VOLUME
    operations ATTR-EQUAL : DISPLAY, DISPLAY → BOOLEAN
    attributes
        COST : DISPLAY → AMOUNT
        VOLUME : DISPLAY → VOLUME
        PPG : DISPLAY → AMOUNT
        GRADE : DISPLAY → GRADE
    axioms
        ATTR-EQUAL(D1, D2) <=> (GRADE(D1) = GRADE(D2)
                & VOLUME(D1) = VOLUME(D2)
                & COST(D1) = COST(D2));
        PPG(D) = COST(D)/VOLUME(D)
end-class
```

Rule OMT-19 requires there be a class set specification for each class specification in a domain theory. The class set for the *Display* class is shown below. The class name, class sort name, and contained class name are all defined by Rule OMT-19 based on the *Display* class. Although not defined in the object model, Rule OMT-21 requires that each event defined in the basic class be defined in the class set over the class set sort. This class set event distributes the event to all objects in the class set as defined by the axiom generated by Rule OMT-22. Since the class set is simply a set of objects, the *new* event is defined by Rule OMT-23 to be simply an empty set.

```
class DISPLAY-CLASS is
    class-sort DISPLAY-CLASS
    contained-class DISPLAY
    events
        RESET-DISPLAY : DISPLAY-CLASS → DISPLAY-CLASS
        PULSE : DISPLAY-CLASS → DISPLAY-CLASS
        NEW-DISPLAY-CLASS : → DISPLAY-CLASS
    axioms
        NEW-DISPLAY-CLASS() = EMPTY-SET;
        fa (D : DISPLAY, DC : DISPLAY-CLASS) in(D, DC)  <=>  in(PULSE(D), PULSE(DC));
        fa (D : DISPLAY, DC : DISPLAY-CLASS) in(D, DC)
                <=>  in(RESET-DISPLAY(D), RESET-DISPLAY(DC));
end-class
```

The *Pump*, *Clutch-Motor-Assembly*, and *Gun-Holster-Assembly* classes are aggregate classes, and as such have a slightly different translation. Again I only show the translation details of one class, the *Gun-Holster-Assembly* aggregate class, while the rest are documented in Appendix G. Because the *Gun-Holster-Assembly* class is an aggregate class, Rule OMT-24 requires the creation of an "aggregate" specification which defines a diagram in the category **Spec**. This diagram specification, *Gun-Holster-Assembly-Aggregate*, consists of a set of nodes (specifications) defined by Rules OMT-25 through OMT-39. In this case, the nodes consist of the event theories sent or received by components of the *Gun-Holster-Assembly* as well as the component specifications themselves. The arcs are based on the nodes in the diagram and are defined by Rules OMT-32 through OMT-38.

```
aggregate GUN-HOLSTER-ASSEMBLY-AGGREGATE is
   nodes RELEASE-HOLSTER-SWITCH, FREE-CLUTCH, ENGAGE-CLUTCH,
      CLOSE-HOLSTER-SWITCH, DISABLE-PUMP, START-TIMER, GUN-CLASS,
      GUN, HOLSTER-CLASS, HOLSTER
   arcs RELEASE-HOLSTER-SWITCH → HOLSTER :
            {RELEASE-HOLSTER-SWITCH-SORT → HOLSTER},
      CLOSE-HOLSTER-SWITCH → HOLSTER :
            {CLOSE-HOLSTER-SWITCH-SORT → HOLSTER},
      GUN → GUN-CLASS : {},
      HOLSTER → HOLSTER-CLASS : {},
      RELEASE-HOLSTER-SWITCH → GUN : {},
      FREE-CLUTCH → GUN : {},
      ENGAGE-CLUTCH → GUN : {},
      CLOSE-HOLSTER-SWITCH → GUN : {},
      DISABLE-PUMP → GUN : {},
      START-TIMER → GUN : {}
end-aggregate
```

Once the aggregate specification is complete, it is imported into the *Gun-Holster-Assembly* specification as defined by Rule OMT-7. This effectively imports every specification that is part of the diagram defined by the aggregate specification into the *Gun-Holster-Assembly*. This specification is the same as a normal class specification as defined above for the *Display* class with a couple of extensions. First, in order to reference its components, an object-valued attribute is created for each aggregate component (or set of components) as defined by Rule OMT-5. These object-valued attributes are the *Gun-Obj* and *Holster-Obj* attributes. Second, the multiplicities of the component must be axiomatized as defined by Rules OMT-8 through OMT-12. In this case, there is exactly one of each component and thus the axiom is of the form $SIZE(HOLSTER\text{-}OBJ(G)) = 1$.

```
class GUN-HOLSTER-ASSEMBLY is
    class-sort GUN-HOLSTER-ASSEMBLY
    import GUN-HOLSTER-ASSEMBLY-AGGREGATE
    attributes
        GUN-OBJ : GUN-HOLSTER-ASSEMBLY → GUN-CLASS
        HOLSTER-OBJ : GUN-HOLSTER-ASSEMBLY → HOLSTER-CLASS
    methodsCREATE-GUN-HOLSTER-ASSEMBLY : → GUN-HOLSTER-ASSEMBLY
    eventsNEW-GUN-HOLSTER-ASSEMBLY : → GUN-HOLSTER-ASSEMBLY
    axioms
        ATTR-EQUAL(G1, G2) <=> (HOLSTER-OBJ(G1) = HOLSTER-OBJ(G2)
            & GUN-OBJ(G1) = GUN-OBJ(G2));
        ATTR-EQUAL(NEW-GUN-HOLSTER-ASSEMBLY(), CREATE-GUN-HOLSTER-ASSEMBLY());
        SIZE(HOLSTER-OBJ(G)) = 1;
        SIZE(GUN-OBJ(G)) = 1
end-class
```

There are two cases of inheritance in the *Pump* domain object model. I concentrate on the *Sophisticated* pump class as it is the most interesting. The *Sophisticated* pump class specification is shown below and looks exactly like a typical class specification with a couple of extensions. First, the subclass specification (*Sophisticated*) imports the superclass specification (*Pump*) as defined by Rule OMT-3 while the class sort of the subclass (*Sophisticate*) is defined as a subsort of the superclass class sort as defined by Rule OMT-4.

```
class SOPHISTICATED is
    class-sort SOPHISTICATED < PUMP
    import PUMP
    operations ATTR-EQUAL : SOPHISTICATED, SOPHISTICATED → BOOLEAN
    attributes
        VOLUME-SELECT : SOPHISTICATED → VOLUME
        AMOUNT-SELECT : SOPHISTICATED → AMOUNT
    methods
        CREATE-SOPHISTICATED : → SOPHISTICATED
    events
        NEW-SOPHISTICATED : → SOPHISTICATED
    axioms
        ATTR-EQUAL(S1, S2) <=> (PUMP.ATTR-EQUAL(S1, S2)
            & AMOUNT-SELECT(S1) = AMOUNT-SELECT(S2)
            & VOLUME-SELECT(S1) = VOLUME-SELECT(S2));
        AMOUNT-SELECT(CREATE-SOPHISTICATED(S)) = 0;
        VOLUME-SELECT(CREATE-SOPHISTICATED(S)) = 0;
        ATTR-EQUAL(NEW-SOPHISTICATED(), CREATE-SOPHISTICATED())
end-class
```

The only other extension required for a subclass is that its class set must import the class set of the superclass as defined by Rule OMT-20 as shown below.

```
class SOPHISTICATED-CLASS is
    class-sort SOPHISTICATED-CLASS
    contained-class SOPHISTICATED
    import PUMP-CLASS
    events NEW-SOPHISTICATED-CLASS : → SOPHISTICATED-CLASS
    axioms NEW-SOPHISTICATED-CLASS() = EMPTY-SET
end-class
```

8.2.2 Pump Domain Dynamic Model. The unique aspects of the dynamic model for each class are described below, starting with those of the *Display* class as shown in Figure 8.2. Besides the initial state, there are exactly two states: *zero-display* and *increment-display.* The transition *new-display/create-display* corresponds to the "new" event and "create" method discussed in Section 6.2.4. The transitions between states are relatively simple and each consists of an event with an associated action.

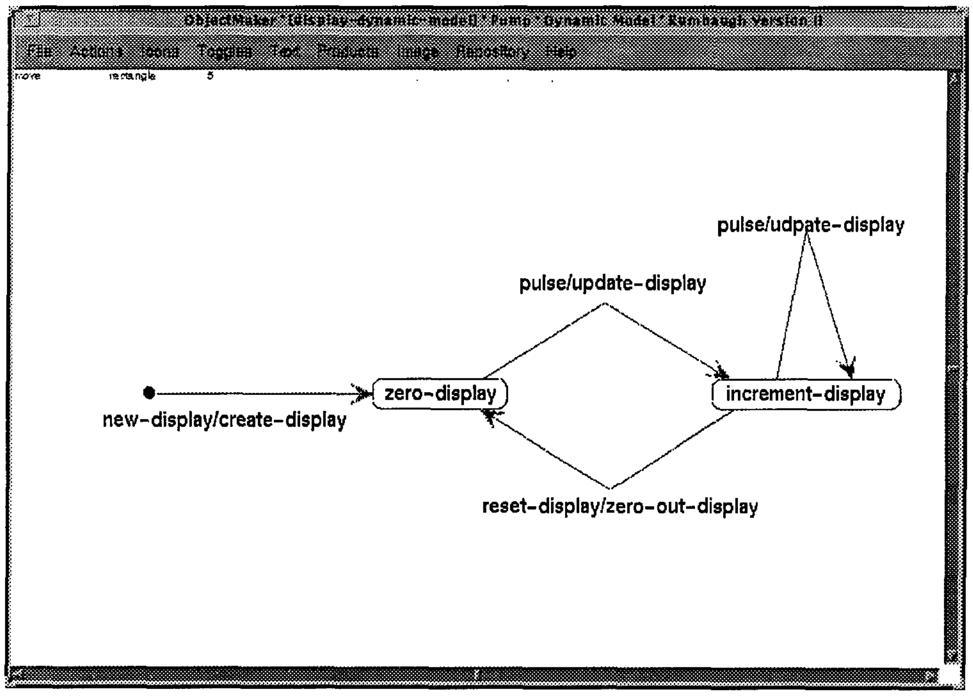

Figure 8.2 Display Class Dynamic Model

The axioms derived from the *Display* class dynamic model are shown below.

$$\mathit{ZERO\text{-}DISPLAY} <> \mathit{INCREMENT\text{-}DISPLAY}; \tag{8.1}$$

$$\begin{aligned}&(\mathit{DISPLAY\text{-}STATE}(\mathit{NEW\text{-}DISPLAY}(D)) = \mathit{ZERO\text{-}DISPLAY}\\&\quad \& \ \mathit{ATTR\text{-}EQUAL}(\mathit{NEW\text{-}DISPLAY}(D), \mathit{CREATE\text{-}DISPLAY}(D)));\end{aligned} \tag{8.2}$$

$$\begin{aligned}&(\mathit{DISPLAY\text{-}STATE}(D) = \mathit{ZERO\text{-}DISPLAY}) => \\&\quad (\mathit{DISPLAY\text{-}STATE}(\mathit{PULSE}(D)) = \mathit{INCREMENT\text{-}DISPLAY}\\&\qquad \& \ \mathit{ATTR\text{-}EQUAL}(\mathit{PULSE}(D), \mathit{UPDATE\text{-}DISPLAY}(D)));\end{aligned} \tag{8.3}$$

$$\begin{aligned}&(\mathit{DISPLAY\text{-}STATE}(D) = \mathit{INCREMENT\text{-}DISPLAY}) => \\&\quad (\mathit{DISPLAY\text{-}STATE}(\mathit{RESET\text{-}DISPLAY}(D)) = \mathit{ZERO\text{-}DISPLAY}\\&\qquad \& \ \mathit{ATTR\text{-}EQUAL}(\mathit{RESET\text{-}DISPLAY}(D), \mathit{ZERO\text{-}OUT\text{-}DISPLAY}(D)));\end{aligned} \tag{8.4}$$

$$\begin{aligned}&(\mathit{DISPLAY\text{-}STATE}(D) = \mathit{INCREMENT\text{-}DISPLAY}) => \\&\quad (\mathit{DISPLAY\text{-}STATE}(\mathit{PULSE}(D)) = \mathit{INCREMENT\text{-}DISPLAY}\\&\qquad \& \ \mathit{ATTR\text{-}EQUAL}(\mathit{PULSE}(D), \mathit{UPDATE\text{-}DISPLAY}(D)));\end{aligned} \tag{8.5}$$

$$\begin{aligned}&\mathit{DISPLAY\text{-}STATE}(D) = \mathit{ZERO\text{-}DISPLAY} => \\&\quad \mathit{DISPLAY\text{-}STATE}(\mathit{RESET\text{-}DISPLAY}(D)) = \mathit{ZERO\text{-}DISPLAY};\end{aligned} \tag{8.6}$$

$$\begin{aligned}&\mathit{DISPLAY\text{-}STATE}(D) = \mathit{ZERO\text{-}DISPLAY} => \\&\quad \mathit{DISPLAY\text{-}STATE}(\mathit{PULSE}(D)) = \mathit{ZERO\text{-}DISPLAY};\end{aligned} \tag{8.7}$$

$$\begin{aligned}&\mathit{DISPLAY\text{-}STATE}(D) = \mathit{INCREMENT\text{-}DISPLAY} => \\&\quad \mathit{DISPLAY\text{-}STATE}(\mathit{PULSE}(D)) = \mathit{INCREMENT\text{-}DISPLAY};\end{aligned} \tag{8.8}$$

Equation 8.1 ensures that the two states defined in Figure 8.2 are unique, and is created by Rule OMT-69. Equations 8.3, 8.4, and 8.5 are defined by the three transition arrows in the graphical representation of the dynamic model as shown in Figure 8.2. These three axioms are created by Rule OMT-82 where the *guard-condition* and *event-sends* parts of the axiom are trivially true since they are not defined in Figure 8.2. The final three axioms, Equations 8.6, 8.7, and 8.8 are created by Rule OMT-84 that ensures no other transitions may be added to the model.

The *Clutch* dynamic model is shown in Figure 8.3. The creation of the O-SLANG axioms that define *Clutch* dynamic model is very similar to the *Display* dynamic model with the exception of a *send* action on the transition from *clutch-free* to *clutch-engaged.* The axiom generated by this transition is shown in Equation 8.9 while the full definition of the *Clutch* dynamic model is shown in Appendix G.

$$\begin{aligned}&(\mathit{CLUTCH\text{-}STATE}(C) = \mathit{CLUTCH\text{-}FREE}) => \\&\quad (\mathit{CLUTCH\text{-}STATE}(\mathit{ENGAGE\text{-}CLUTCH}(C)) = \mathit{CLUTCH\text{-}ENGAGED}\\&\quad \& \ \mathit{START\text{-}FUEL\text{-}OBJ}(\mathit{ENGAGE\text{-}CLUTCH}(C))\\&\qquad = \mathit{START\text{-}FUEL}(\mathit{START\text{-}FUEL\text{-}OBJ}(C)));\end{aligned} \tag{8.9}$$

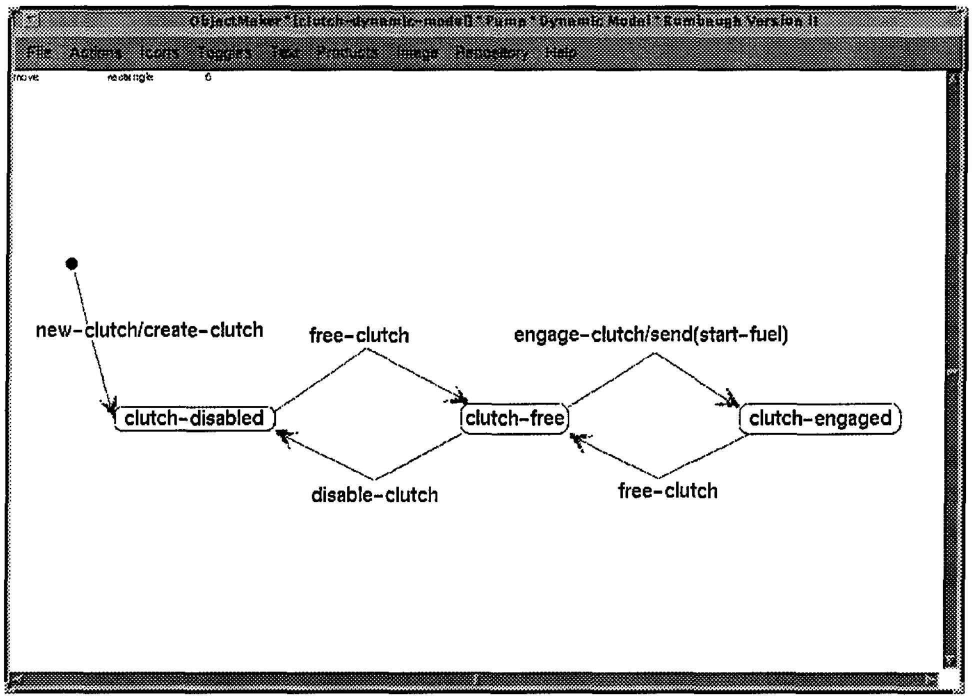

Figure 8.3 Clutch Class Dynamic Model

Equation 8.9 uses the *start-fuel-obj* object-valued attribute and *start-fuel* event defined by Rules OMT-81 and OMT-77 through OMT-80 to "send" the event *engage-clutch.*

The *Motor* class dynamic model is shown in Figure 8.4. Its translation is similar to that of the *Clutch* class above and its O-SLANG representation is shown in Appendix G.

The *Gun* dynamic model is shown in Figure 8.5. It is similar to the dynamic model presented above except for the *replace-gun* transition from state *gun-enabled* to *gun-disabled* that sends three different events. The axioms generated by the *replace-gun* transition are shown in below.

$$
\begin{aligned}
&(GUN\text{-}STATE(G) = GUN\text{-}ENABLED) => \\
&\quad (GUN\text{-}STATE(REPLACE\text{-}GUN(G)) = GUN\text{-}DISABLED \\
&\quad \& \; START\text{-}TIMER\text{-}OBJ(REPLACE\text{-}GUN(G)) \\
&\qquad = START\text{-}TIMER(START\text{-}TIMER\text{-}OBJ(G)) \\
&\quad \& \; DISABLE\text{-}PUMP\text{-}OBJ(REPLACE\text{-}GUN(G)) \\
&\qquad = DISABLE\text{-}PUMP(DISABLE\text{-}PUMP\text{-}OBJ(G)) \\
&\quad \& \; CLOSE\text{-}HOLSTER\text{-}SWITCH\text{-}OBJ(REPLACE\text{-}GUN(G)) \\
&\qquad = CLOSE\text{-}HOLSTER\text{-}SWITCH(CLOSE\text{-}HOLSTER\text{-}SWITCH\text{-}OBJ(G))); \qquad (8.10)
\end{aligned}
$$

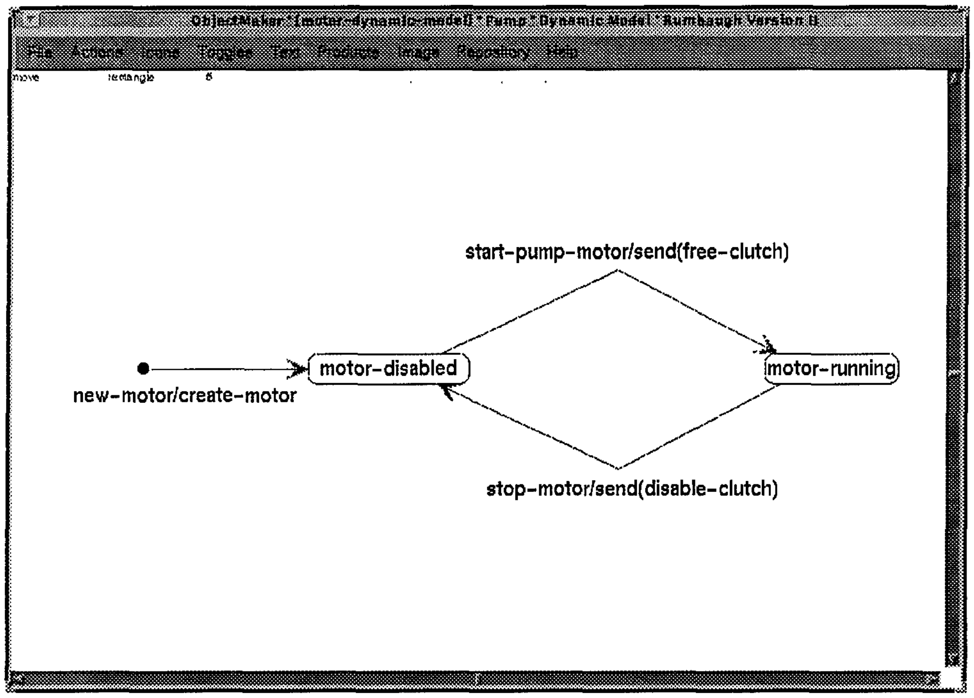
File Actions Icons Toggles Text Products Image Repository Help
start-pump-motor/send(free-clutch)
motor-disabled
motor-running
new-motor/create-motor
stop-motor/send(disable-clutch)

Figure 8.4 Motor Class Dynamic Model

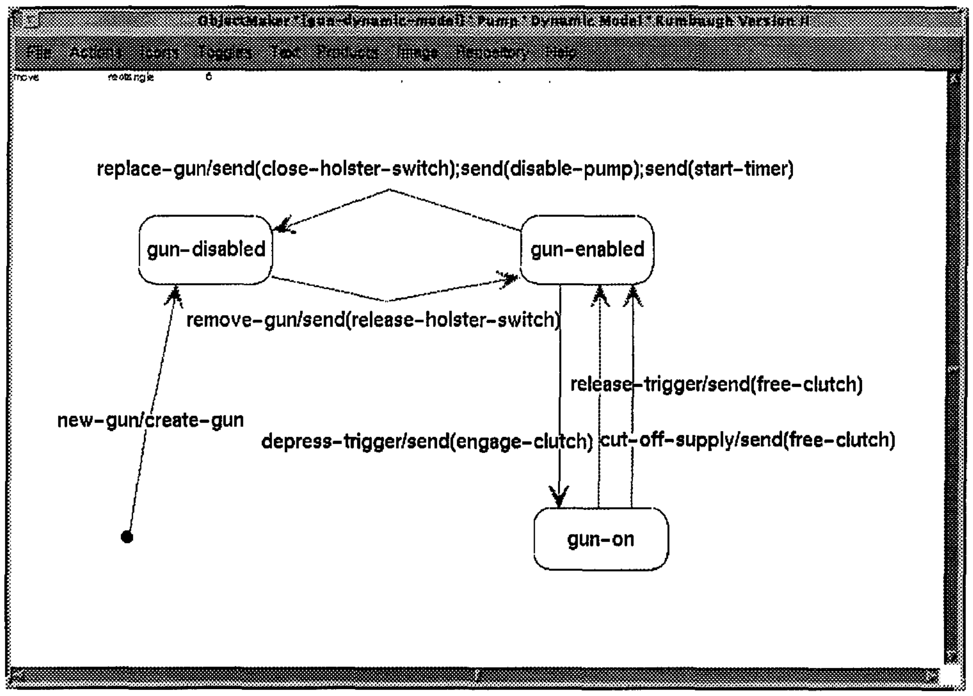
File Actions Icons Toggles Text Products Image Repository Help
replace-gun/send(close-holster-switch);send(disable-pump);send(start-timer)
gun-disabled
gun-enabled
remove-gun/send(release-holster-switch)
release-trigger/send(free-clutch)
new-gun/create-gun
depress-trigger/send(engage-clutch)
cut-off-supply/send(free-clutch)
gun-on

Figure 8.5 Gun Class Dynamic Model

This example shows clearly the use of a separate object-valued attribute for each event sent, even though those events might actually be sent to the same object. This ambiguity is acceptable since O-SLANG is only representing a domain model at this point.

The *Holster* class dynamic model shown in Figure 8.6 is relatively simple and similar to the dynamic models discussed above. No further clarification of its translation (as shown in Appendix G) is required.

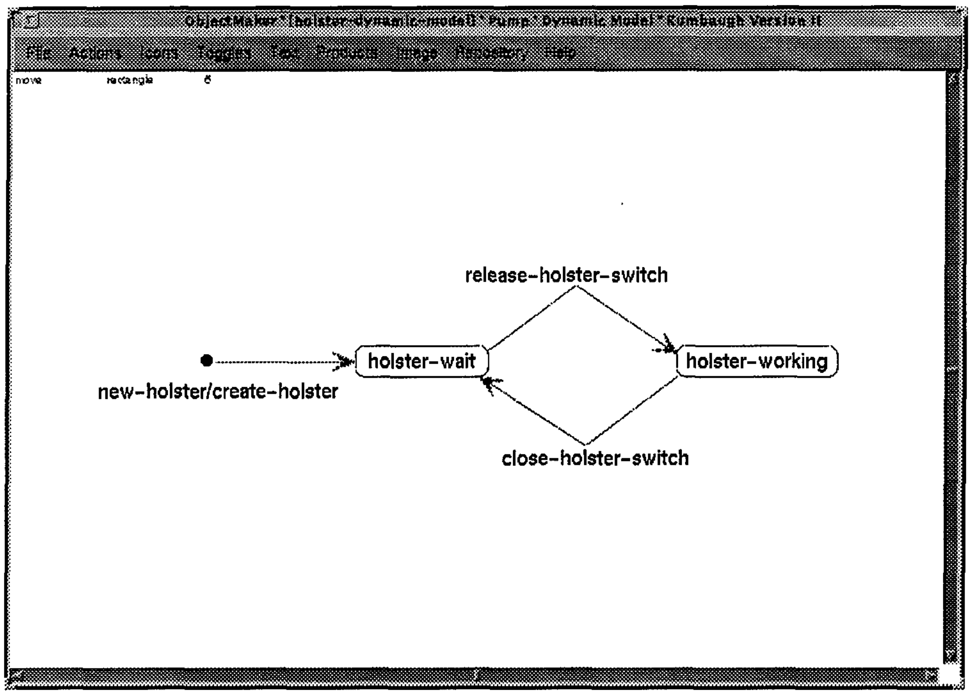

Figure 8.6 Holster Class Dynamic Model

The *Pump* dynamic model (Figure 8.7) introduces two new transition features: *parameters* and *guards*. Actually, both of these features are incorporated into the *enable-pump* transition from state *pump-disabled* to *pump-enabled*. The parameter x of type *pump-id* is received by the pump object with the *enable-pump* event. If the pump is in the *disabled-pump* state and the guard condition $x =$ *pump-id* is true, then the transition takes place. Again, this transition is converted to O-SLANG by Rule OMT-82 and is shown below.

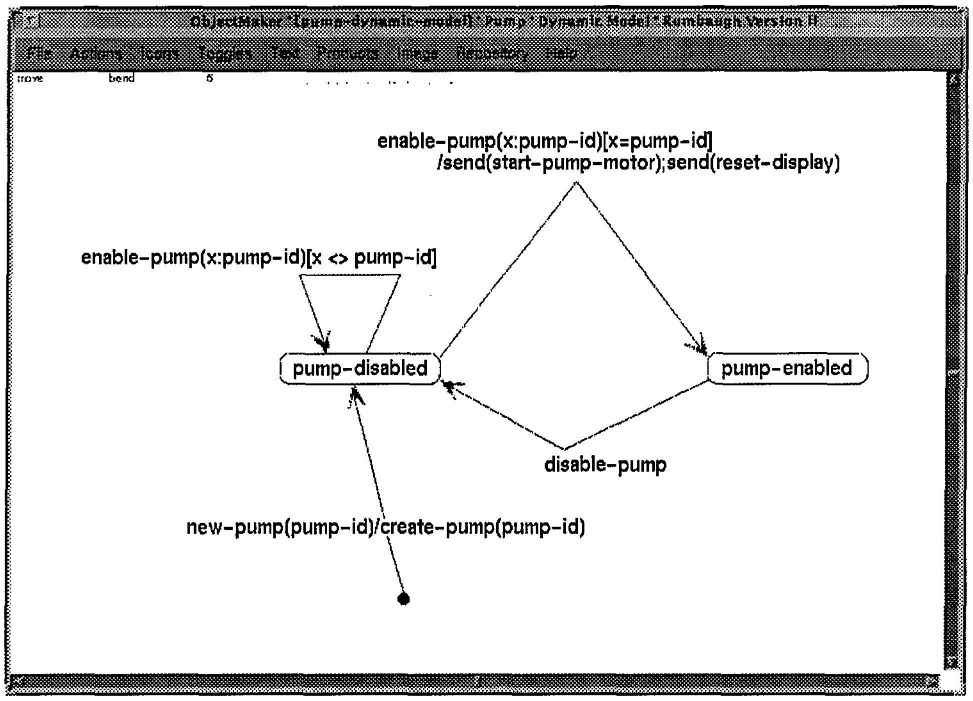
enable-pump(x:pump-id)[x=pump-id] /send(start-pump-motor);send(reset-display)
enable-pump(x:pump-id)[x <> pump-id]
pump-disabled
pump-enabled
disable-pump
new-pump(pump-id)/create-pump(pump-id)

Figure 8.7 Pump Class Dynamic Model

$$\begin{aligned}
&(PUMP\text{-}STATE(P) = PUMP\text{-}DISABLED \ \& \ (X = PUMP\text{-}ID(P))) \\
&\quad => (PUMP\text{-}STATE(ENABLE\text{-}PUMP(P, X)) = PUMP\text{-}ENABLED \\
&\quad \& \ RESET\text{-}DISPLAY\text{-}OBJ(ENABLE\text{-}PUMP(P, X)) \\
&\qquad = RESET\text{-}DISPLAY(RESET\text{-}DISPLAY\text{-}OBJ(P)) \\
&\quad \& \ START\text{-}PUMP\text{-}MOTOR\text{-}OBJ(ENABLE\text{-}PUMP(P, X)) \\
&\qquad = START\text{-}PUMP\text{-}MOTOR(START\text{-}PUMP\text{-}MOTOR\text{-}OBJ(P))); \qquad (8.11)
\end{aligned}$$

Equation 8.11 inserts the guard condition $(X = PUMP\text{-}ID(P))$ before the implication to ensure the condition holds before forcing the events *start-pump-motor* and *reset-display* to be sent. As defined by Assumption V.4, the user is responsible for ensuring the completeness and consistency of guard conditions used in a dynamic model; therefore, the *enable-pump* transition from *pump-disabled* to *pump-disabled* with a guard condition of $(X <> PUMP\text{-}ID(P))$ is added. The transition is shown below in O-SLANG syntax.

$$\begin{aligned}
&(PUMP\text{-}STATE(P) = PUMP\text{-}DISABLED \ \& \ (X <> PUMP\text{-}ID(P))) \\
&\quad => (PUMP\text{-}STATE(ENABLE\text{-}PUMP(P, X)) = PUMP\text{-}DISABLED \qquad (8.12)
\end{aligned}$$

```
class-constraints: Display
        display-state(d) = zero-display => cost(d) = 0 & volume(d) = 0;
        display-state(d) = increment-display => cost(d) >= 0 & volume(d) >= 0;
        cost(d) >= 0;
        volume(d) >= 0
end class-constraints.

definition: update-display class = display;
        grade(update-display(d)) = grade(d);
        cost(update-display(d)) = cost(d) + 1;
        volume(update-display(d)) = volume(d) + 1
end definition.

definition: zero-out-display class = display;
        grade(zero-out-display(d)) = grade(d);
        cost(zero-out-display(d)) = 0;
        volume(zero-out-display(d)) = 0
end definition.
```

Figure 8.8 Pump MANUAL.TEXT File

8.2.3 Pump MANUAL.TEXT. The *MANUAL.TEXT* file as shown in Figure 8.8 is used in the *Pump* domain model to define constraints on the *Display* class and to add the semantics for two display operations: *update-display* and *zero-out-display.* Basically, the body of these three declarations are O-SLANG axioms which are incorporated directly into the axiom block of the *Display* class as shown in Appendix G.

8.3 Faculty Student Database Domain

The *Faculty Student Database* domain defines a specification for a simple school database. The database consists of a set of records for students, faculty, courses, classes, sections and quarters and are related by a set of associations. The domain object model is presented in Section 8.3.1, the functional model is discussed in Section 8.3.2, and the additional textual input is defined in Section 8.3.3. There is no dynamic model in the faculty student database domain. The O-SLANG specification of the domain model is shown in Appendix G.

8.3.1 Faculty Student Database Domain Object Model. Figure 8.9 shows the object model for the faculty student database domain model. Since the classes in the object model are relatively straightforward, I do not discuss them in detail as they are translated exactly like the classes of the *Pump* domain in Section 8.2.1. Their O-SLANG specifications are shown in Appendix G. Instead I concentrate on the associations defined in the domain object model: *member-of*, *advises*, *teaching*, *taught-as*, *offering*, *scheduled-in*, and *teaches*. The first six associations listed are basically identical in that they are simple binary associations between two classes. The only difference in their definitions is the multiplicity axioms used. The last association, *teaches*, is unique in that it has two link attributes defined: *times-taught* and *average-size*. I first discuss the *member-of* association as an example of the six simple binary relations followed by a detailed description of the *teaches* relation.

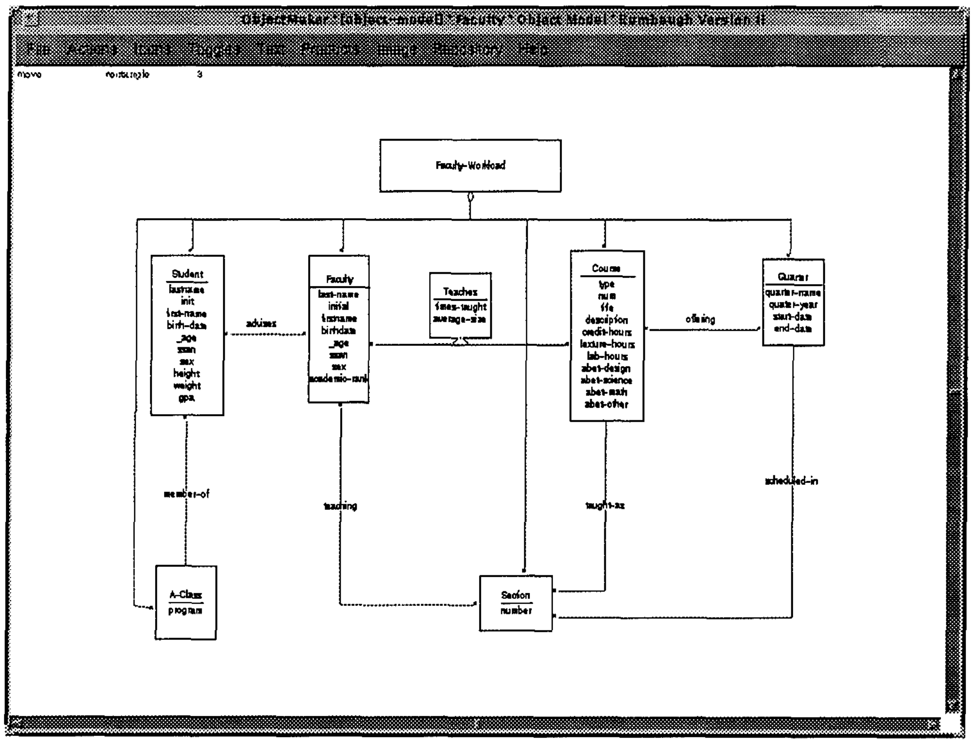

Figure 8.9 Faculty-Student Database Object Model

As described in Section 6.4, the theory-based object model defines two specifications for each association: a *link* specification and an *association* specification. The link specification defines a class of objects with attribute-valued objects that reference particular objects involved in a relationship as well as any link attributes or operations. The association specification, on the other hand, is analogous to a class set specification and is used to the define the set of links and any multiplicity constraints. The link specification for the *member-of* association is shown below.

```
link MEMBER-OF-LINK is
    class-sort MEMBER-OF-LINK
    sort STUDENT, A-CLASS
    operations ATTR-EQUAL : MEMBER-OF-LINK, MEMBER-OF-LINK → BOOLEAN
    attributes
        A-CLASS-OBJ : MEMBER-OF-LINK → A-CLASS
        STUDENT-OBJ : MEMBER-OF-LINK → STUDENT
    methods CREATE-MEMBER-OF-LINK : A-CLASS, STUDENT → MEMBER-OF-LINK
    events NEW-MEMBER-OF-LINK : A-CLASS, STUDENT → MEMBER-OF-LINK
    axioms
        ATTR-EQUAL(M1, M2) <=> (STUDENT-OBJ(M1) = STUDENT-OBJ(M2)
                & A-CLASS-OBJ(M1) = A-CLASS-OBJ(M2));
        STUDENT-OBJ(CREATE-MEMBER-OF-LINK(M, S, A)) = S;
        A-CLASS-OBJ(CREATE-MEMBER-OF-LINK(M, S, A)) = A;
        ATTR-EQUAL(NEW-MEMBER-OF-LINK(M, S, A), (CREATE-MEMBER-OF-LINK(M, S, A)))
end-link
```

Rule OMT-43 actually creates the specification and the class sort. The sorts *student* and *a-class* are unified with the class sort of the associated classes in the appropriate aggregate layer and are defined by Rule OMT-47. The attributes *a-class-obj* and *student-obj* are object-valued attributes that reference objects from their respective classes and are defined by Rule OMT-61. The *attr-equal*, *create*, and *new* operations are defined the same as for simple classes in the object model.

The association specification for *member-of* is shown below. The basic specification, class sort, and link class are defined in Rule OMT-44. The sorts *student-class* and *a-class-class* are to be unified with the class set sorts of the associated classes and are defined by Rule OMT-45.

```
association MEMBER-OF is
    class-sort MEMBER-OF
    link-class MEMBER-OF-LINK
    sort STUDENT-CLASS, A-CLASS-CLASS
    operations
        IMAGE : MEMBER-OF, STUDENT → A-CLASS-CLASS
        IMAGE : MEMBER-OF, A-CLASS → STUDENT-CLASS
    eventsNEW-MEMBER-OF :→ MEMBER-OF
    axioms
        NEW-MEMBER-OF() = EMPTY-SET;
        fa (M : MEMBER-OF, S : STUDENT)SIZE(IMAGE(M, S)) = 1;
        fa (M : MEMBER–OF, A : A–CLASS)SIZE(IMAGE(A, X)) >= 0
    fa (S : MEMBER-OF, M : STUDENT, B : A-CLASS)
        (ex(A : MEMBER-OF-LINK)in(A, S)
            & MEMBER-OF-OBJ(A) = M & MEMBER-OF-OBJ(A) = B)
        <=> in(B, image(S, M));
    fa (S : MEMBER-OF, M : STUDENT, B : A-CLASS)
        (ex(A : MEMBER-OF-LINK)in(A, S)
            & MEMBER-OF-OBJ(A) = B & MEMBER-OF-OBJ(A) = M)
        <=> in(M, image(S, B))
end-association
```

In order to constrain the multiplicities of objects in the association, an *image* operation is created for each class in the association. Therefore, in the *member-of* association, Rule OMT-46 requires the definition of two *image* operations, each returning the set of objects associated with a given object as shown below.

```
fa (M : MEMBER-OF, S : STUDENT)SIZE(IMAGE(M, S)) = 1
fa (M : MEMBER-OF, A : A-CLASS)SIZE(IMAGE(A, X)) >= 0
```

The second multiplicity axiom shown is not actually included in the automatically generated O-SLANG in Appendix G since by definition, the size of any set is always greater than or equal to zero.

8.3.2 Faculty Student Database Domain Functional Model. There are actually two functional models for the Faculty Student Database domain. The first is the *Faculty Workload* functional model found in Section G.2. The second, the *Update-Teaches* model, is shown in Figure 8.10 and is further refined in Figure 8.11. Both models are translated into O-SLANG in the same manner; however, since the *Update-Teaches* function has fewer subprocesses while incorporating more aspects of the OMT functional model, I discuss its functional model translation in detail here. The O-SLANG for the *Faculty-Workload* function is defined in Appendix G.

The top-level diagram for the *Update-Teaches* function is shown in Figure 8.10. There are three explicit inputs to the function: *name*, *type*, and *num*; however, by Rule OMT-85, the object upon which the function works is also an input. Because, as shown in Figure 8.11, a subprocess of

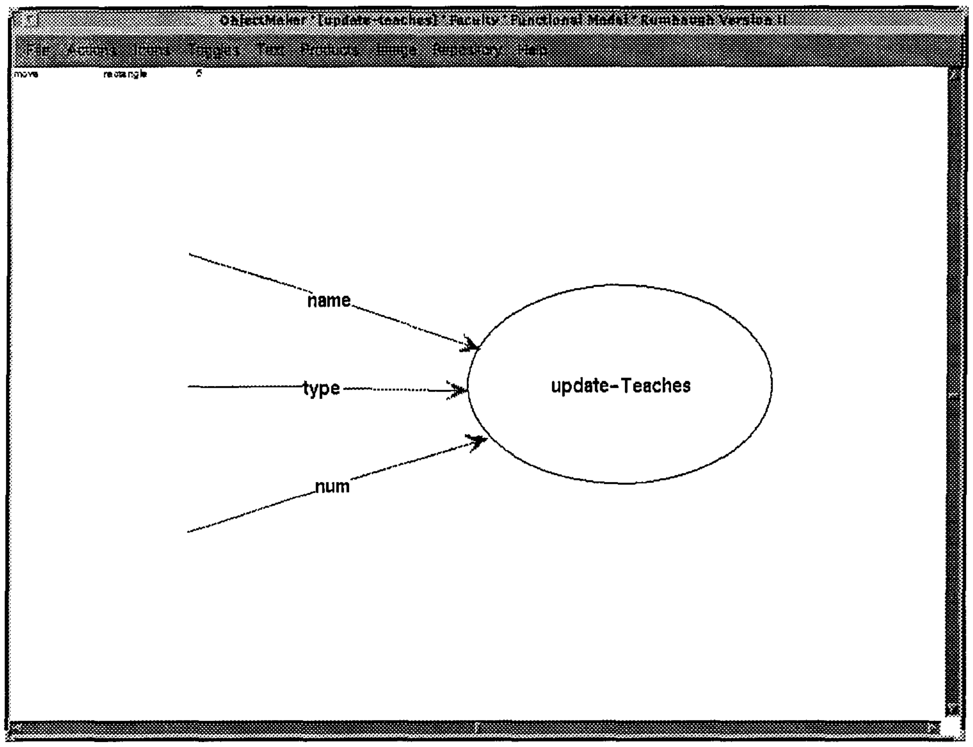
ObjectMaker * [update-teaches] * Faculty * Functional Model * Rumbaugh Version II
File Actions Icons Toggles Text Products Image Repository Help
move rectangle 6
name
type
num
update-Teaches

Figure 8.10 Update-Teaches Functional Model

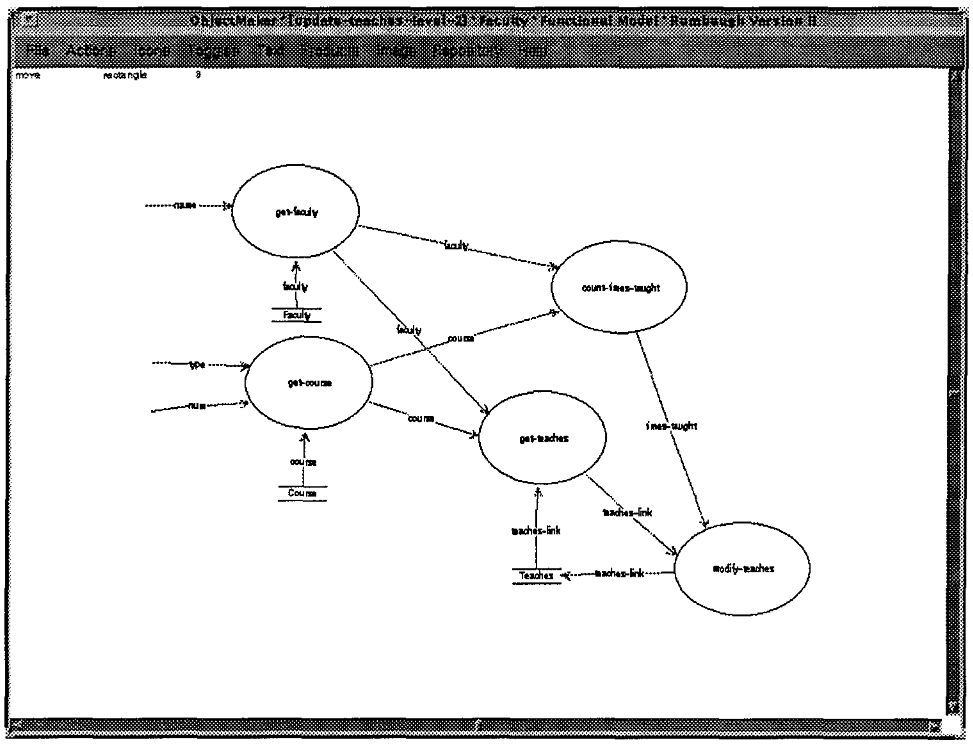

Figure 8.11 Update-Teaches Functional Model Level 2

Update-Teaches (*modify-teaches*) modifies the *Teaches* association, *Update-Teaches* is a method as shown below by its signature.

$$UPDATE\text{-}TEACHES : FACULTY\text{-}WORKLOAD, NUM, NAME, TYPE \rightarrow FACULTY\text{-}WORKLOAD$$

Update-Teaches is a method in the *Faculty Workload* class since the process *modify-teaches*, a subprocess of *Faculty Workload*, modifies the *Teaches* association as shown in Figure 8.11. The remaining operation signatures as defined by Rule OMT-85 or Rule OMT-86 are shown below.

```
GET-FACULTY : FACULTY-CLASS, NAME → FACULTY
GET-COURSE : COURSE-CLASS, NUM, TYPE → COURSE
GET-SECTIONS-TAUGHT : SECTION-CLASS, FACULTY → SECTION-CLASS
GET-SECTIONS-OFFERED : SECTION-CLASS, COURSE → SECTION-CLASS
COMPUTE-SECTION-UNION : SECTION-CLASS, SECTION-CLASS → TIMES-TAUGHT
COUNT-TIMES-TAUGHT : SECTION-CLASS, COURSE, FACULTY → TIMES-TAUGHT
GET-TEACHES : TEACHES, FACULTY, COURSE → TEACHES-LINK
MODIFY-TEACHES : TEACHES, TIMES-TAUGHT, TEACHES-LINK → TEACHES
```

The implementing axiom for the method *Update-Teaches* as defined by Rule OMT-87 is shown below. The parameters come directly from the functional model diagram with the exception of the datastores (classes and associations). Datastore names are translated to their object-valued attribute names. The operation *Count-Times-Taught* has a similar definition based on its functional model as shown in Figure 8.12. The O-SLANG translation of the implementing axioms for *Count-Times-Taught* is shown in Appendix G.

```
UPDATE-TEACHES(F, NUM, NAME, TYPE) = F1
    & TEACHES-OBJ(F1)
        = MODIFY-TEACHES(TEACHES-OBJ(F), TIMES-TAUGHT, TEACHES-LINK)
    & TEACHES-LINK = GET-TEACHES(TEACHES-OBJ(F), FACULTY, COURSE)
    & TIMES-TAUGHT = COUNT-TIMES-TAUGHT(SECTION-OBJ(F), COURSE, FACULTY)
    & COURSE = GET-COURSE(COURSE-OBJ(F), NUM, TYPE)
    & FACULTY = GET-FACULTY(FACULTY-OBJ(F), NAME);
```

8.3.3 Faculty Student Database MANUAL.TEXT. The Faculty Student Database *MANUAL.TEXT* file (Figure 8.13) is used to define super/subprocess relationships. In this example, the "leaf" processes semantics are not included for simplicity; however, the process definitions would be similar to *update-display* and *zero-out-display* shown in Figure 8.8.

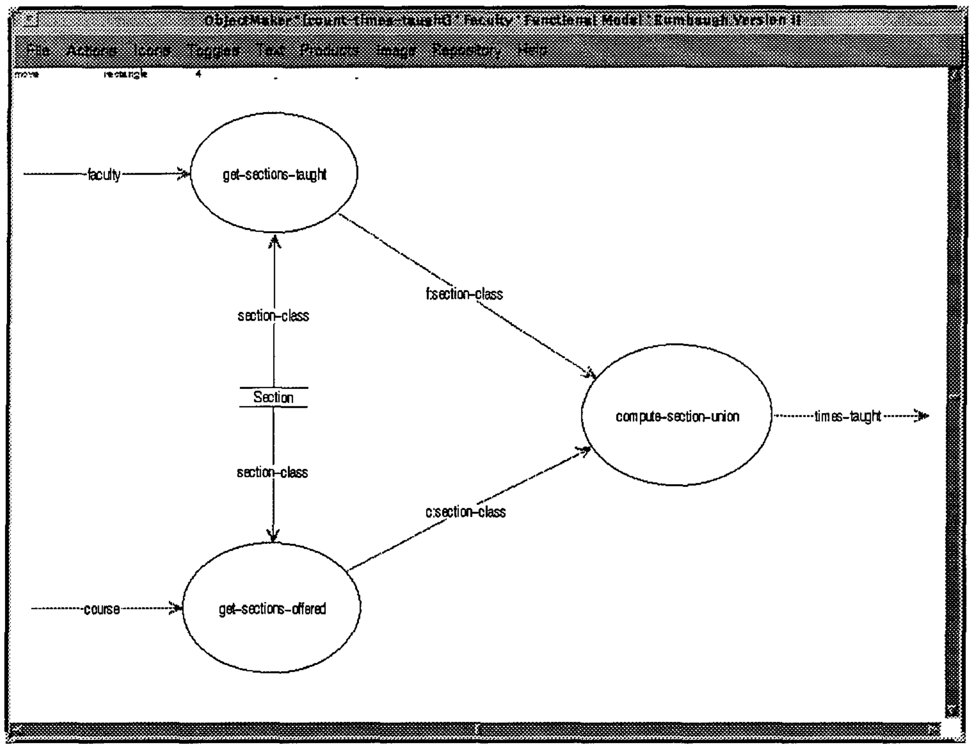

Figure 8.12 Count-Times-Taught Functional Model

```
subprocess get-faculty < process calculate-faculty-workload.
subprocess calculate-course-load < process calculate-faculty-workload.
subprocess calculate-student-load < process calculate-faculty-workload.
subprocess calculate-workload < process calculate-faculty-workload.
subprocess get-sections < process calculate-course-load.
subprocess compute-credits < process calculate-course-load.
subprocess get-students-advised < process calculate-student-load.
subprocess count-students < process calculate-student-load.

subprocess get-faculty < process update-teaches.
subprocess get-course < process update-teaches.
subprocess count-times-taught < process update-teaches.
subprocess get-teaches < process update-teaches.
subprocess modify-teaches < process update-teaches.

subprocess get-sections-taught < process count-times-taught.
subprocess get-sections-offered < process count-times-taught.
subprocess compute-section-union < process count-times-taught.
```

Figure 8.13 Faculty Student Database MANUAL.TEXT File

8.4 Summary

This chapter presented two examples of the automated translation of graphically-based domain model specifications into O-SLANG using the translations defined in Chapter VII. The first example, the *Pump* domain, was almost exclusively dynamic in nature and showed the feasibility of automatically translating dynamic models into valid theory-based specifications. The second example, the *Student Faculty Database* domain, was almost purely functional in nature and showed the automatic translation of the restricted functional model for both methods and operations. This section concentrated on instructive features from both domains; however, the complete O-SLANG domain model is contained in Appendix G.

IX. Conclusions and Recommendations

The purpose of this research was to investigate the feasibility of a parallel refinement approach to the acquisition of formal specifications based on graphically-based, object-oriented concepts and theory-based algebraic specifications. This investigation focused on two main areas: a formal mathematical framework of object-oriented concepts using theories within the category **Spec**, and the automatic translation of graphically-based, object-oriented diagrams into this theory-based framework.

The first phase in this investigation focused on establishing the formal mathematical framework for the object-oriented paradigm within a categorical setting. First, classes were defined as theory presentations or specifications within the category **Spec** while their models were equated with an *implementation* of the class. The theoretical concept of an object instance was defined and used to show the desired effect of inheritance. Both single and multiple inheritance were formally defined using category theory operations on classes. This formal definition of inheritance was then shown to preserve the "Substitution Property", a commonly proposed notion of what valid inheritance should be. Next, a theory-based object model defining concepts from Rumbaugh's OMT notation in the formal mathematical framework was developed. Because OMT is a semi-formal technique, a formal semantics for each OMT model was first defined.

Provably correct translations, with respect to the previously defined formal semantics, from the restricted OMT models to a theory-based specification were then defined. These translations map each concept in OMT into a specific representation within O-SLANG. To show the feasibility of automating these translations, a proof of concept system was developed which took OMT models created with a commercially available, graphically-based OMT drawing tool and automatically translated them into a generic abstract syntax tree representation and then into O-SLANG.

9.1 Summary of Contributions

This section summarizes the contributions of this research as enumerated in Chapter I. The first contribution is the formalization of basic object-oriented concepts using algebraic and category theory constructs. While there has been prior work on the formalization of the individual aspects of object-orientation (12, 14, 23, 38, 69, 94), this research is the first effort to formally define all the important aspects of object-orientation (i.e., classes, inheritance, aggregation, association, and communication) in a cohesive, computationally tractable framework that is applicable to semi-automatic software synthesis (55, 86, 87).

The second contribution of this work is the formalization of a generally accepted notion of class inheritance, the *Substitution Property.* While other attempts have been made to formalize inheritance (12, 39), the use of category theory constructs to define valid inheritance leads naturally to a computationally tractable sufficiency criteria for proving adherence to that formalization. In fact, adherence to this formalization of inheritance, along with aggregation and association, provides techniques for ensuring the consistency of object-oriented specifications based on the composition process itself.

The next contribution of this effort is the formalization of the semantics of the object, dynamic, and functional OMT models. While Harel (45) defined the semantics of statecharts based on traditional automata theory, formalization of communications paths within a domain specification along with the object model and the functional models have seen little work. While the concept of global, event-based communications is relatively simple, its formalization is not. However, use of category theory concepts allows the specification of the *capability* of a class to communicate with other classes. Building on the work of Bourdeau and Cheng (14), formalization of the object model semantics was accomplished using specifications to define classes and boolean-valued predicates to define the relationships between classes. The informal semantics of the functional model required major restrictions before it could be automatically translated into a formal representation. The work of Tao and Kung (92) on standard data flow diagrams was tailored to my *restricted* functional model in order to complete the formalization of the OMT semantics.

The advances described above made possible the major contribution of this work, which is defining, formalizing, and automating the translation of graphically-based, object-oriented specifications into algebraic specifications. This formalization and automation of specification translation increases the level of abstraction at which formal specifications may be developed and thus holds the potential to dramatically increase the acceptance of formal specifications and methods. Without raising the level of abstraction at which formal specifications can be developed, specification of large, complex systems will remain too difficult for the average software developer, and the potential of formal methods (e.g., automated software synthesis, etc.) will never be realized.

9.2 Conclusions and Results

Several specific conclusions can be drawn from this investigation.

1. The category **Spec** provides a formal foundation for the rigorous definition of object-oriented concepts. Classes and associations are defined as theories. Single and multiple inheritance can be formally modeled and correctly constructed using specification morphisms. Aggregate objects may be effectively modeled and correctly constructed using colimits of component classes and specification morphisms.

2. The semantics of the OMT models were formalized by restricting the informal and semi-formal notation allowed in OMT. Formalization of these semantics allows the models to be automatically translated into theories in the category **Spec**.

3. Translations from graphically-based OMT object, dynamic, and functional models were developed using a theory-based model of object-orientation. These translations were shown to correctly translate the models into O-SLANG based on their restricted formal semantics.

4. A category theory-based algebraic specification language, O-SLANG, was developed based on the functional algebraic specification language SLANG. O-SLANG allows object-oriented models to be captured naturally using algebraic specifications. O-SLANG incorporates basic object-oriented concepts such as classes, associations, and aggregates as well as basic category-theory operations of specification morphisms, diagrams, and colimits.

5. The feasibility of creating an automated translation system was established through the development of a proof-of-concept system using a commercial front-end graphics tool. This proof-of-concept tool was used to automatically translate non-trivial dynamically and functionally based OMT domain models into algebraic theories.

9.3 Future Work

This investigation has laid the foundation for the *Specification Acquisition Mechanism* defined in Chapter II and shown again in Figure 9.1. However, to complete this vision, the results of this investigation must be extended. Several areas requiring additional research are identified and summarized below.

1. The generic OMT AST developed to capture object-oriented concepts from the three basic OMT models should be analyzed to determine its capability to capture object-oriented concepts using other object-oriented modeling methodologies and techniques. A single generic AST would allow any number of methodologies to be incorporated into an automated tool such as the proof-of-concept tool developed in this investigation. While the generic OMT AST was not developed with this more general purpose in mind, some object-oriented techniques appear to be similar enough to OMT to allow their incorporation into the generic AST (13, 19, 21, 84).

2. Investigation of the transformations required to develop problem-specific system specifications from O-SLANG domain models should be undertaken. This is the *Specification Generation*

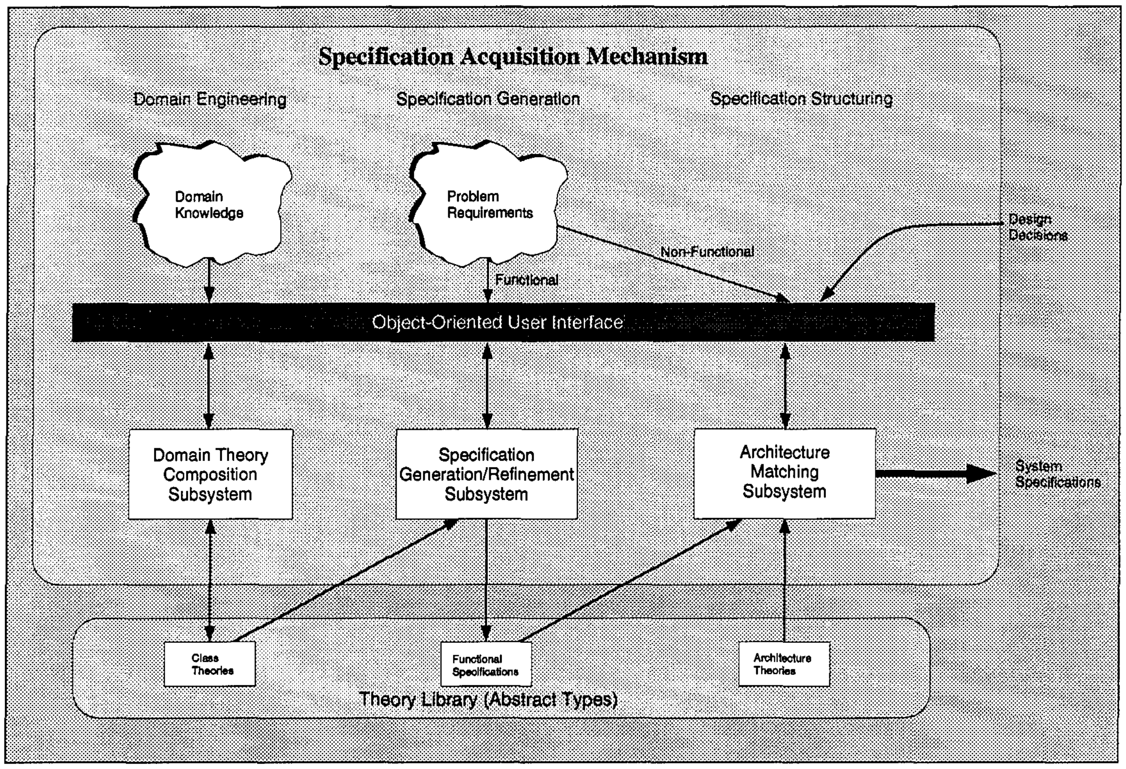
Specification Acquisition Mechanism
Domain Engineering
Specification Generation
Specification Structuring
Domain Knowledge
Problem Requirements
Non-Functional
Functional
Design Decisions
Object-Oriented User Interface
Domain Theory Composition Subsystem
Specification Generation/Refinement Subsystem
Architecture Matching Subsystem
System Specifications
Class Theories
Functional Specifications
Architecture Theories
Theory Library (Abstract Types)

Figure 9.1 Parallel Refinement Specification Acquisition Mechanism

phase shown in Figure 9.1. There has been some work in this area (49). Some possible transformations might include the following.

(a) parameter instantiation
(b) specialization selection
(c) multiplicity restriction
(d) initialization definition
(e) communication path definition
(f) constraint restriction

3. Domain modeling literature often refers to providing a domain-specific language for developing problem-specific system specifications. In this research, I assumed the transformations would be accomplished using a generic object-oriented representation. However, domain-specific graphically-based languages should be a topic for further research. Given a theory-based domain model, the domain engineer should be able to define graphically-based icons for classes of objects and associations within the domain that a user could then use to build a problem-

specific specification. Such a domain-specific language might even implicitly incorporate some of the domain to problem-specific transformations discussed above. Some work has been done in this area (56).

4. Matching theory-based, problem-specific functional specifications to architecture theories in the *Specification Structuring* phase of Figure 9.1 requires additional research. While theory-based architecture theories have been addressed (32), many problems associated with automatically matching them to system specifications need further research. One such problem closely associated with the object-oriented paradigm is the dynamic creation of objects, which requires the ability to specify dynamically modifiable architectures.

5. The formal, automated translation of O-SLANG to SLANG should be developed. Assuming the ability of SPECWARE or some other suitable transformation system to transform SLANG to executable code, this would be the final link between graphically-based object-oriented specifications and executable code.

6. The definition of a graphically-based object-oriented methodology or technique designed specifically for formal transformation should be investigated. While many graphically-based object-oriented methodologies have been proposed (13, 15, 19, 20, 21, 72, 73, 84), they all have some degree of informality or ambiguity associated with them. Research starting from the theory-based object model and attempting to develop graphically-based mechanisms for defining those critical theory-based concepts might yield some unique, possibly more efficient graphically-based specification techniques.

7. A graphically-based tool to work directly with the generic OMT AST should be developed. Such a tool could enforce the assumptions made about the OMT model usage and would simplify tool development. Also, such a tool would aid in the investigation of the generic OMT AST as well as an object-oriented methodology/technique designed specifically for translation to algebraic theories.

8. O-SLANG should be extended to add the necessary structuring devices to ensure a two-way transformation from the GOMT AST to O-SLANG and back again. While it should not be difficult to implement, it is critical to the flexibility of the Parallel Successive Refinement Approach.

9.4 Reflections on the Parallel Successive Refinement Approach

A Parallel Successive Refinement Approach to specification acquisition was presented in Chapter I and a Specification Acquisition Mechanism based on such an approach was introduced in Chapter II. In a true parallel refinement approach, two versions of the specification are kept and refined in parallel. In my Specification Acquisition Mechanism I proposed that two versions of the same specification need not be kept *assuming* the specification representation could be automatically generated. Upon completion of my research, I find I was correct. (Even though the two-way translation is not yet complete, it appears clear that it is possible). Maintaining one specification, with multiple representations, is much simpler than maintaining two specifications and ensures consistency. Therefore, a better name for the methodology represented by the system in Figure 9.1 is a *Multiple Representation Approach.*

The Multiple Representation system proposed in Figure 9.1 shows specifications being stored in "theory" form; however, the best long-term storage format has not been ascertained, if, in fact, it makes a significant difference. While it appears evident that modifications to domain theories should be made via the graphical user interface (which argues for storing domain theories in a graphical format), the development of elicitor-harvester technology to help in selecting appropriate domain theories from Theory Library might be more tractable if domain theories are stored in a theory-based format. It is also possible that some intermediate format such as the GOMT AST might also be the best choice.

In conclusion, it is my conviction that formal methods will never reach their true potential unless a straightforward, cost-effective capability for developing large-scale system specifications can be found. From the experience gained in this investigation, I believe a Multiple Representation Approach is a viable, if not the best, approach to providing such a capability in the near future.

9.5 Summary

Given the results, contributions and conclusions presented in this and previous chapters, it is obvious that the objectives of this investigation were successfully accomplished. Specifically, a formal mathematical framework of object-oriented concepts within the category **Spec** was defined and the automatic translation of graphically-based, object-oriented diagrams into this theory-based framework was demonstrated. Additionally, several areas of future research were identified.

Appendix A. Generic OMT Abstract Syntax Tree

A.1 Introduction

This Appendix defines the generic model of object-orientation based on Rumbaugh's OMT (83) that is used in my research. Section A.2 defines the features of OMT that are translated to O-SLANG using a generic OMT (GOMT) AST representation. Section A.3 discusses the GOMT AST objects derived from class defintions in the OMT object model, including attributes and operations, while Section A.4 defines the AST objects associated with OMT object model associations. Section A.5 presents the objects of the GOMT AST derived from the OMT dynamic model while Section A.6 discusses the GOMT AST objects defined from the OMT functional model. Completeness of the GOMT in terms of Rumbaugh's OMT representation is discussed in Section A.7.

A GOMT AST is used to store domain models specified using OMT notation. In this research, this specification, translation, and storage is performed using the demonstration tool described in Appendix D. Using this tool, once a domain model is stored as a GOMT specification it can be automatically translated into O-SLANG as defined in Chapter VII.

The notation used to present the GOMT AST structure follows the conventions of the OMT object model with two extensions. First, rectangles represent classes of AST objects (which may or may not have subobjects or attributes) while rectangles with rounded corners represent *symbols* or *numbers* that define specific attributes associated with their parent AST objects. Second, AST subobjects with multiplicities greater than one may either represent a *set* of AST subobjects or a *sequence* of AST subobjects. Sets are denoted by braces ({ }) surrounding the AST object name while sequences are denoted by brackets ([ ]).

A.2 Object Modeling Technique Representation Assumptions

An OMT specification has three basic components: an object model, a dynamic model, and a functional model. However, these three models do not operate independently, although Rumbaugh does not rigorously define their precise interaction. The OMT object model defines the structure of a domain in terms of the object classes in the domain and the relationships between them. Each object class is defined in terms of its attributes and operations while the relationships between them are captured via associations, inheritance, and aggregation. The dynamic model captures the state-based behavior of a class and is generally defined at the class level. The functional model describes how a domain transforms data and is not concerned with the objects involved or transformation timing.

Rumbaugh does not tie the three models together tightly; therefore, I made some assumptions about the how the OMT models are used.

1. **Assumption A.1** *The object model is developed first and defines all object classes and associations present in a domain. Attributes are defined for each object class.*
2. **Assumption A.2** *There is an implied domain-level aggregate class for each object model. This domain-level object class is an aggregate composed of each object class and association defined in the object model.*
3. **Assumption A.3** *There is a dynamic model for each object class. This dynamic model defines the states, events, actions, and communications for each object class. A purely functional object class has one state and a unique event for each operation defined.*
4. **Assumption A.4** *The dynamic model is represented as a Mealy state machine where all actions occur on transitions between states. Non-state attributes values are not modified simply by receiving an event; a specific* action *must be specified in response to a state transition in order to modify an object's non-state attribute values.*
5. **Assumption A.5** *Events not included in a dynamic model are assumed to have no affect on the object.*

6. **Assumption A.6** *Events with common names represent the same event between all dynamic models in a given domain.*

7. **Assumption A.7** *There is a functional model for each aggregate object class in the domain. Aggregate functional models are used to define the effect of actions specified in the aggregate dynamic model across all components of the aggregate. They may not define the effect of aggregate actions on components outside the aggregate.*

8. **Assumption A.8** *All primitive actions (those not defined by a functional model) are defined axiomatically using first order logic in* O-SLANG *syntax. These definitions completely define the effect of the operation on each object attribute.*

Although OMT has three distinct models, it basically models object classes and their relationships. The dynamic and functional OMT models (making the above assumptions) serve only to define the behavior of those object classes. Therefore, the GOMT AST defined in this Appendix is based on a set of object classes and a set of associations between them. Figure A.1 shows the top-level of the GOMT AST. Other relationships such as aggregation and inheritance are captured naturally in the object class definitions.

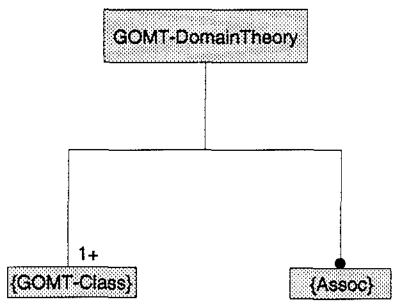

Figure A.1 Top Level GOMT Abstract Syntax Tree

A.3 Class Objects

The OMT object model captures the structure of a domain model by defining the classes of objects in the domain model in terms of their attributes and operations, as well as relationships between object classes. Each class describes a group of objects with similar attributes and behavior. Class relationship constructs include inheritance and aggregation as well as associations that may exist between classes. As stated above, due to my assumptions about how OMT is used to define domain models, an OMT class consists of object model class constructs as well as dynamic and functional components. Figure A.2 shows the AST for the GOMT "Class". Basic attributes such as the class name, attributes, and operation definitions are captured directly in the AST. Each of the subobjects in the class AST is discussed below.

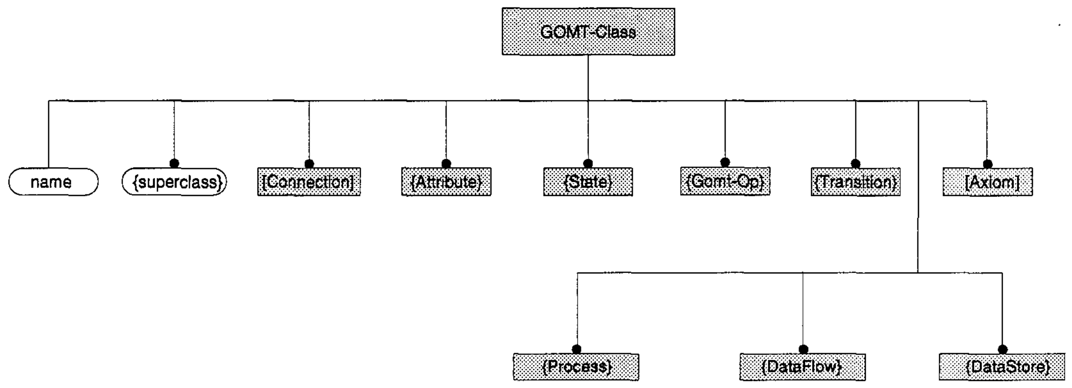

Figure A.2 GOMT Class Abstract Syntax Tree

The *name* attribute is simply a symbol that stores the name of the class. Inheritance is captured in the superclass attribute. The *superclass* attribute is just a set of symbols representing the names of all superclasses.

Aggregation is captured by a set of *connection* objects as shown in Figure A.3. Connection objects are used to represent aggregates as well as associations (as discussed below). Each connection links the current class to a component class. The *name* attribute is the name of the component class. The *qualifier* subobject defines aggregate (or association) qualifiers and has an associated *name* and a *datatype.* Each connection may also have a role name associated with it as defined by the *role* attribute. The role is simply a name placed on one end of an aggregate or association. Finally, the multiplicity of the component (or association) connection is defined by a *mult* subobject. Each connection has a multiplicity and this multiplicity can be either one, many,

many ordered, one-or-more, optional (zero or one) or specified by a subset of non-negative integers. These options are captured in a *mult* object. Subtypes of the multiplicity object – one, many, plus, optional, and specified – capture these possibilities. The subtypes *one*, *optional*, and *many* have no attributes. Their object type alone uniquely identifies the multiplicity. The *plus* subtype actually captures more than the basic one-or-more multiplicity as defined by Rumbaugh and allows a more general "*n* or more" multiplicity by allowing the user to insert any non-negative integer as the *int* attribute. The *specified* multiplicity object is also very flexible. Each specified object has a set of ranges defined by the value attributes. Each spec-range object may have either one or two values. If only one value is specified, then exactly that value is taken as a valid multiplicity. If two values are specified then the entire range between *value1* and *value2* (inclusive) is a valid range.

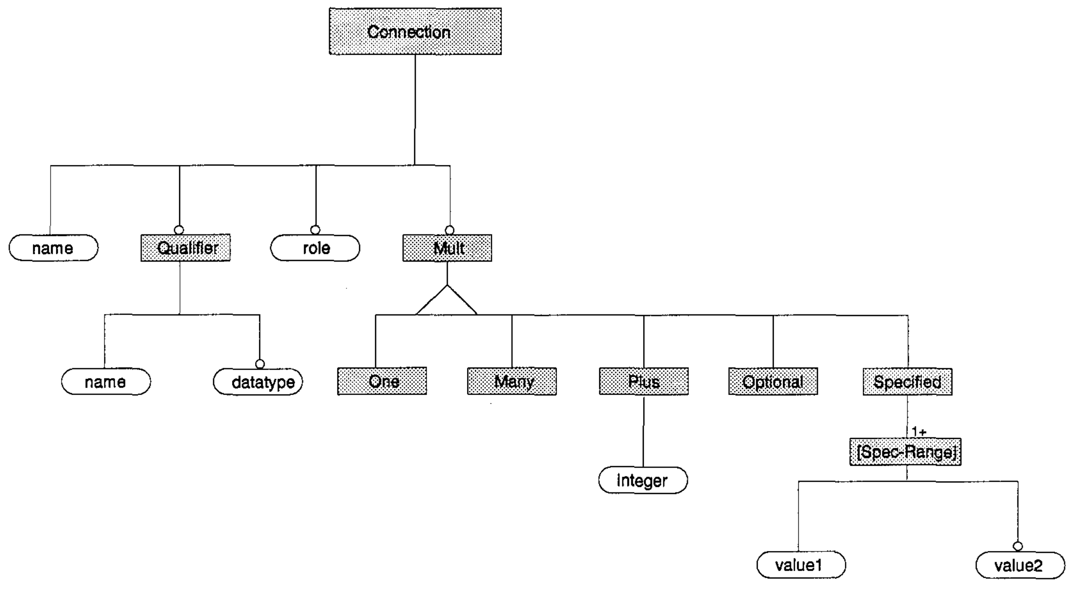

Figure A.3 Connection Abstract Syntax Tree

Class-level constraints are captured via the class object's *constraints* subobject. This constraint object is actually a set of *axiom* objects defined below. In my research, the axiom objects are first-order algebraic axioms; however, in general, the axiom object definition may be replaced by any axiom notational style. Therefore, other notations such as Z's set-based syntax, could be inserted as well.

Each class dynamic model is captured by the *states* and *transitions* objects. Each of these objects are sets of subobjects that define the state or transition. Both state objects and transition objects are discussed in Section A.5. The class functional model is captured by the *process*, *datastore*, and *dataflow* Each these objects are discussed in detail in A.6.

A.3.1 Attribute Objects. The AST for the GOMT Attribute object is shown in Figure A.4. There are two types of attributes as defined by Rumbaugh: *normal* and *derived.* Normal attributes are those attributes that have a name and a datatype and may take on any values in the datatype. Unless constrained via class-level constraints, these attribute values are generally independent of other attribute values. Derived attributes, on the other hand, derive their value from values of the other class attributes. Therefore, there are two subtypes of an attribute object: *derived-attr* and *normal-attr.* Although both have a similar structure–*name*, a set of *axioms*, and a *datatype*–the interpretation of that structure is different. Actually, the name and the datatype attributes have the same interpretation. It is in the set of axioms where the difference lies. The axioms of a derived attribute define the value of the derived attribute based on other class attributes. In a normal attribute, the axioms define only the initial value associated with the attribute. This set of axioms may be a single value such as "12.3" or it may be a complex set of axioms based on other attribute values set during object initialization.

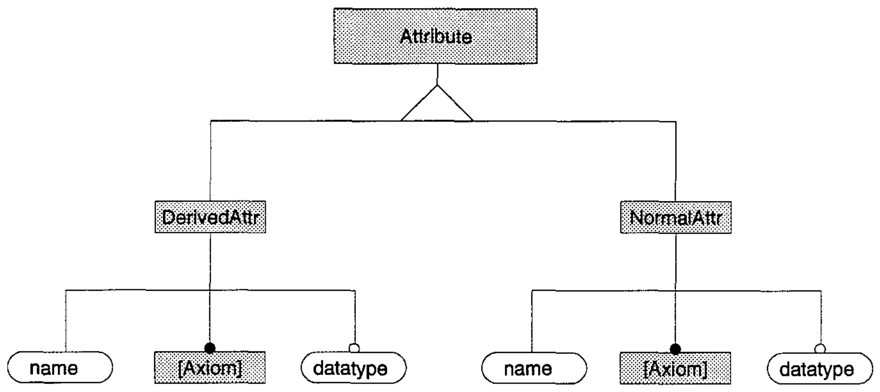

Figure A.4 GOMT Attribute Abstract Syntax Tree

A.3.2 Operation Objects. Figure A.5 shows the AST for the *GOMT-Op* object. An operation object can fully define a class operation. The operation signature is defined by the *name* attribute, parameter subobjects, and the result subobject. The *parameter* subobject is a set of parameter objects with *name* and *datatype* attributes. The *result* subobject defines the *datatype* returned by the operation (if required). The *definition* subobject is a set of axiom objects that define the semantics of the operation. Finally, *is-abstract* is a boolean-valued attribute that states whether the operation is fully defined or not.

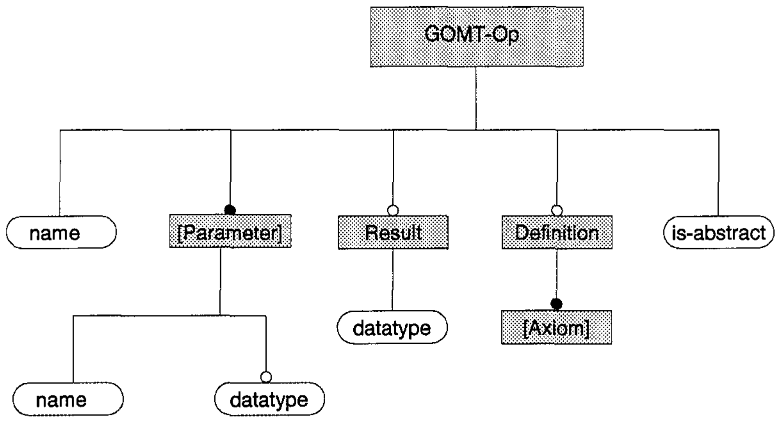

Figure A.5 GOMT Operation Abstract Syntax Tree

A.4 Association Objects

The AST for an *Assoc* (association) object is shown in Figure A.6. Every association has a *name* and at least two connections (object classes). The *connection* object is the same as defined above in Section A.3 and defines the classes, roles, qualifiers, and multiplicities of each class in the association. Associations may also have link *attributes* and link *operations.* These are the same attributes and operations as defined in Sections A.3.1 and A.3.2.

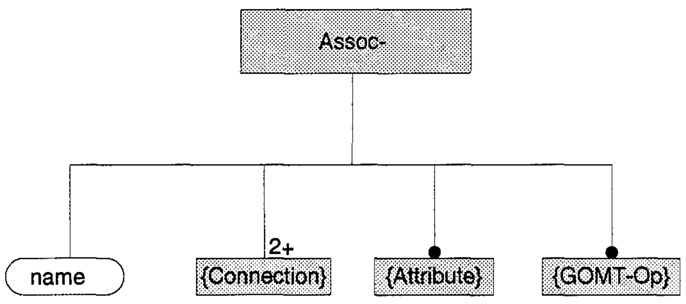

Figure A.6 GOMT Association Abstract Syntax Tree

A.5 Dynamic Model Objects

The OMT dynamic model captures the temporal behavior of object classes defined in the object model. The basic concepts of the dynamic model include events, actions, and states. An *event* is a one-way transmission from one object to another and may pass additional information. An *action* is an operation performed by an object. This operation may modify the object state or generate an event. A *state* is defined as an abstraction of attribute values of an object. The dynamic model is represented using state charts and event flow diagrams. A statechart models the transitions between states of a given object class. In the dynamic model, these transitions are caused by the reception of a given event and result in an action being taken. In the dynamic model, OMT allows actions to occur *in* a state or *on* transitions (i.e., a combined Mealy-Moore machine); however, to simplify the translation process in this research, I have restricted the dynamic model to a Mealy machine representation where all actions occur on transitions. This does not represent a semantic restriction since the equivalence of Mealy and Moore machines is well known (47:44). Guards (boolean expressions) may be added to transitions to prevent them from occurring unless certain conditions are met. If the guard is true and the event for that transition is received, the transition takes place. If the guard is false, then the transition does not take place even though the required event is received.

For a given class, the dynamic model is captured as a set of states and transitions between states. The ASTs for the state and transition objects are shown in Figures A.7 and A.8. Each *state* object has a *name* attribute, possibly a set of *axiom* objects defining state invariants, and a set of *substate* objects. Actually, Rumbaugh allows two special types of states: concurrent and substates. Concurrent states are defined implicitly in a set of states. Concurrent states are partitions of a set of states where there are no transitions from one partition to the other. The second special type of state is a substate. Substates are a set of states that further define transitions while within some other, higher-level state. Substates are defined by a set of *state* objects internal another state. Thus states in a class AST are arranged hierarchically. That is, the *state* objects of a class may not contain all the states and substates for the class. It only directly contains the top-level states. Each substate is actually stored under its superstate object.

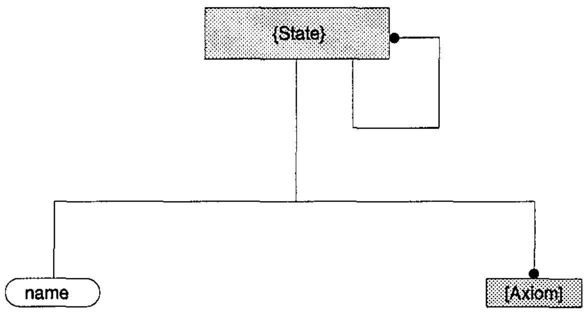

Figure A.7 GOMT State Abstract Syntax Tree

While states are stored hierarchically, transition objects are stored as a single set under the class object. This forces all states in a class to have unique names. A transition defines an arc from one state to another based on the receipt of an event. The transition *name* attribute is the name of the incoming event while the *from-state* and *to-state* attributes are the names of the two states involved. Each transition may have parameters, a guard condition, and a set of actions that are performed upon receipt of the named event. The *parameter* objects are a sequence of parameters as defined in Section A.3.2, the guard condition is defined by a single *axiom*, and the actions are a sequence of *action* objects defining the actions performed by the object upon receipt of the event. Each *action* object has a *name* attribute, a sequence of *parameter* objects, and possibly another *action* object. These subaction objects are only used when its parent action is a "send" action. A "send" action is the OMT syntax for broadcasting an event, defined in the subaction object, to the system. Subaction objects are only used with a "send" action.

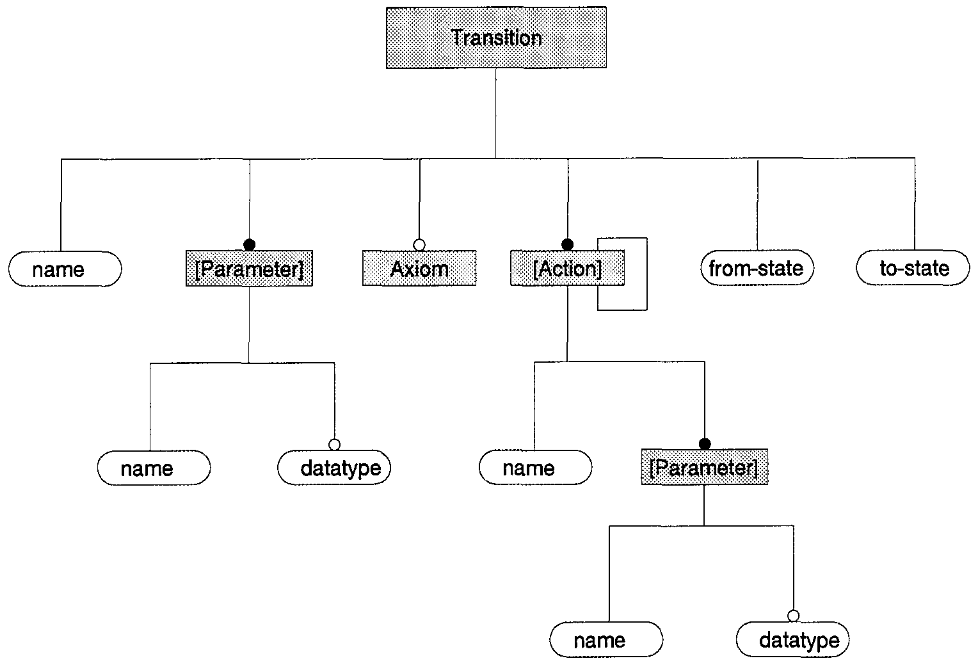

Figure A.8 GOMT Transition Abstract Syntax Tree

A.6 Functional Model Objects

In OMT, the functional model is represented by a data flow diagram that shows how an aggregate-level class implements functions using component events and methods within the aggregate. Basically, there are three components in the restricted OMT functional model: processes, data flows, and data stores. *Processes* translate data and may use subprocesses to accomplish their function. Such higher-level processes are generally decomposed into subprocesses using nested data flow diagrams. *Data flows* show how data is passed between processes and datastores. Besides passing data from the output of one process to the input of the next process, data flows may be (1) duplicated and passed to many processes, (2) decomposed into multiple components, or (3) composed from a number of components into a single aggregate value. *Data stores* are passive objects that store data for later use. Obvious data stores are the object class sets and associations. A data store allows a process to create, update, and store data for later use.

The ASTs for the three functional components are shown in Figure A.9. The *process* and *datastore* objects are very similar. Each has a *name* attribute and two sets of *flow* objects: *data-flows-in* and *data-flows-out.* A *flow* object simply stores the *name* and *type* attributes of *dataflow* object defined below. The process object is slightly different from the datastore object in that the process object may also have subprocess *process* objects. This set of subprocesses define the implementation of the process as defined in Section 5.5.

The third functional object is a *dataflow* object. A dataflow object has four attributes: name, type, source, and target. The *name* of the dataflow object is the name of a dataflow arc between two entities (processes or datastores) in the functional model while the *type* attribute represents the datatype of the dataflow. The *source* and *target* attributes store the name of the process or datastore where the dataflow originates and terminates. A dataflow with a blank source or target attribute denotes an off-page connector. In the case of a top-level dataflow, an off-page connector represents an input or output external to the domain. The actual source/target of an off-page connector in a nested diagram is (or at least should be) the source/target of the "parent" dataflow from the parent diagram.

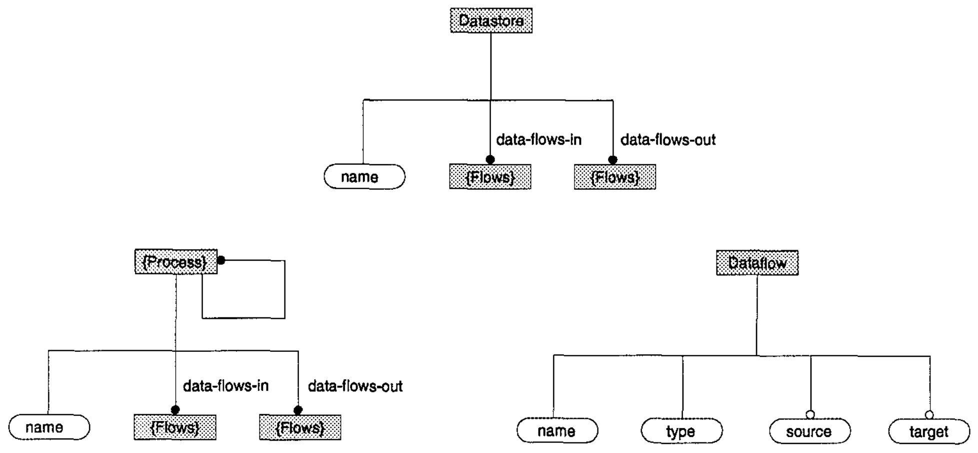

Figure A.9 GOMT Functional Model Abstract Syntax Tree

A.7 Generic OMT AST Completeness

This section discusses the completeness of the GOMT AST in terms of what it captures and what it does not. Section A.7.1 discusses the completeness of the GOMT AST in terms of the OMT object model, Section A.7.2 discusses the OMT dynamic model, and Section A.7.3 discusses the OMT functional model.

A.7.1 Object Model. The GOMT AST has the capability to capture all the basic features of the OMT object model. The GOMT AST directly models the classes (concrete and abstract), attributes (normal and derived), and operations (concrete and abstract). The GOMT AST also captures the relationships defined in an OMT object model including inheritance, aggregation, and association. More subtle features of OMT relationships such as discriminators, multiplicities, and link attributes and operations are also captured.

There are a few, non-critical, OMT object model items not directly captured by the GOMT AST. The first of these is operator propagation; however, the inability to directly model operator propagation does not restrict the domain designer since the effect of operator propagation may be explicitly expressed through the axiomatic definition of the operation. Derived classes and associations are also not captured in the GOMT AST. According to Rumbaugh (83:75) derived classes and associations are redundant and are completely determined by other objects; the only reason for derived classes and associations is to aid understandability. Therefore, since the lack of derived classes and associations does not restrict the domain designer's modeling ability, they are not included in the GOMT AST. General association constraints are also not modeled directly in the GOMT AST; however, the domain designer may capture these constraints as class-level constraints (via first-order axioms) in the aggregate containing the association.

A.7.2 Dynamic Model. The OMT dynamic model consists of set of states and transitions based on a Mealy-Moore state machine. The GOMT AST directly captures the set of states and transitions between them; however, while the OMT dynamic model allows states to have internal activities and actions, in the GOMT AST, I assume that all actions occur only on transitions (i.e., a Mealy state machine). However a Mealy machine (where activities and actions occur only on transitions) has been shown to be equivalent in expressive power to a Moore machine (where activities and actions occur in a state). Thus any dynamic model captured in a Mealy-Moore machine is translatable to a Mealy machine and may be captured by the GOMT AST. The OMT dynamic model also allows for entry and exit actions that are performed on all transitions into or out of a state. Although entry and exit actions simplify the diagram, they are easily placed on appropriate incoming and outgoing transition with the same affect.

OMT also allows for substates and concurrent states. Substates are modeled directly in the GOMT AST. Concurrent states are not explicitly denoted in the GOMT AST; however, the set of concurrent states can be computed by partitioning all class states such that no states in distinct partitions share a transition. Once again, although not captured directly in the GOMT AST, the set of concurrent states are computable.

Two state transition features allowed in the OMT dynamic model that are not captured directly in the GOMT AST are control splitting and synchronization. In essence, *control splitting* corresponds to transitioning to a state with concurrent substates, as shown in Figure A.10. *Synchronization* of control corresponds to leaving a set of states after two or more events occur in any order. In Figure A.10 *event1* causes the state to change from state A to C, or more correctly to the two concurrent states $C1$ and $C2$. When events *event2* and *event3* have both happened, in any order, the state changes to state B. Without special split and synchronization operators, this same control mechanism is implemented as shown in Figure A.11. The attributes *synch2* and *synch3* are used to determine when event 2 or 3 has been received. Then two separate transitions leave state C guarded by the condition that the appropriate synchronization attribute has been set to true. Simple enumeration of all possible combinations of state and synchronization variable values

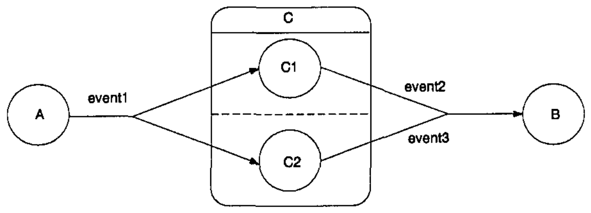

Figure A.10 OMT Split/Synchronization

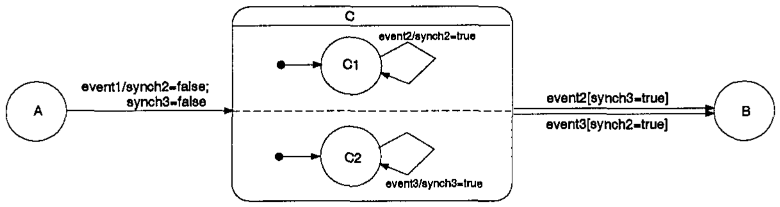

Figure A.11 Generic OMT Split/Synchronization

shows that, although Figure A.10 is more aesthetically pleasing, both Figures A.10 and A.11 have the same semantic result. Therefore, although the GOMT AST does not include transition splits or synchronizations, their effect is realizable by the user.

A.7.3 Functional Model. The OMT functional model uses data flow diagrams consisting of processes, dataflows, datastores, actors, and control flows. Actors and control flows are not directly included in the GOMT AST. A functional model process transforms data and may be further refined by subprocesses. Processes and subprocesses are stored directly in the GOMT AST.

Simple dataflows are also directly stored in the GOMT AST; however, the GOMT AST does not capture some special dataflow notation included in OMT for convenience: duplication, composition, and decomposition. Duplication allows the output of one process to flow into the inputs of two or more processes. Composition takes multiple dataflows and composes their data into an aggregate datatype. Decomposition accomplishes the inverse of composition–it allows aggregate datatypes to broken into it component datatypes. Each of these operations can be modeled with appropriately defined processes. Therefore, lack of direct inclusion in the GOMT AST does not present modeling difficulties.

According to Rumbaugh, datastores are passive objects that store data and are defined in the object model. Thus datastores are classes or associations and are directly modeled in the GOMT AST.

Actors are active objects that drive OMT functional models by producing and consuming the data used in the corresponding dataflow diagram. However, since actors represent entities interacting with the domain being modeled, they are external to the domain and are not required in order to specify the domain model. Therefore, actors are not modeled in the GOMT AST.

The last items of interest in the OMT dynamic model are control flows. However, according to Rumbaugh, control flows are duplicative. Therefore, omitting control flows from the GOMT AST eliminates a source of inconsistency and does not restrict the domain designer.

A.8 Conclusions

This Appendix presents the generic OMT abstract syntax tree definition that is used in my research as the starting point for the translation from OMT to my theory-based object model. The GOMT captures the essential details of OMT based on assumptions of how OMT is used. Because OMT has a rich set of duplicative features, some of the non-essential features are not directly modeled in the GOMT AST; however, OMT's modeling power has not been decreased. Any domain modeled in Rumbaugh's OMT representation may be modeled in the generic OMT AST without semantic compromise.

Appendix B. O-SLANG

B.1 Introduction

This Appendix defines the syntax and semantics of the algebraic specification language O-SLANG. O-SLANG is an object-oriented extension of SLANG (54) and like SLANG incorporates category theory operations directly and implicitly. The syntax of O-SLANG is defined by the grammar and the corresponding O-SLANG AST as described in Section B.3. The semantics of O-SLANG is defined by its translation to SLANG.

B.2 Background

O-SLANG is an extension of the SLANG specification language used in Specware (54, 55). SLANG is based on first-order logic and category theory. A SLANG specification is a *theory presentation* of a formal theory. *Theories* consist of a finite set of sorts, operations, and a set of axioms closed under logical entailment. A specification is a set of sorts, operations, and a set of finite axioms. Under logical entailment, a specification generates a theory that includes, as axioms, all theorems that can be generated from the axioms in the specification. Specifications and specification morphisms define the cocomplete category **Spec**. Diagrams in **Spec** consist of specifications and specification morphisms and are used to define system structure. SLANG uses the category theory *colimit* operation to combine smaller specifications into larger, more complex specifications.

O-SLANG uses the concepts from SLANG to capture object-oriented system specifications. Sorts and operations are used to describe various internal object class features while category theory concepts and operations are used to define the relationships between object classes.

B.3 O-SLANG *Syntax*

O-SLANG syntax is very similar to the core SLANG syntax with some additional language constructs. The Refine O-SLANG AST definition and grammar are shown in Section B.3.1. Whereas core SLANG has only specification and diagram constructs to define a system, O-SLANG has numerous object-oriented constructs that map into specifications and diagrams. Also, O-SLANG hides much of the diagram construction except in the case of aggregation. A table of basic O-SLANG constructs and their SLANG counterparts are shown in Table B.1.

O-SLANG	SLANG
Class	Diagram & Specification
Abstract-Class	Diagram & Specification
Event	Specification
Link	Specification
Association	Diagram & Specification
Aggregate	Diagram

Table B.1 O-SLANG Constructs

The top level O-SLANG abstract syntax tree is shown in Figure B.1. Figure B.1 defines an O-SLANG Domain Theory as consisting of one or more O-SLANG specifications. O-SLANG specifications represent either classes, abstract classes, events, links, associations, or aggregates. Classes, events, links, and associations are all similar in that they share a similar specification body.

Figure B.2 shows a further breakdown of classes, events, links, and associations. Actually Figure B.2 is over generalized. While classes have all of the features shown, events, links, and associations only have a subset of those as shown in Figure B.3.

Figure B.4 shows the abstract syntax tree for O-SLANG axioms. Although both O-SLANG and SLANG are based on first order logic, their axiom syntax is different. SLANG uses a hard to read "Lisp-like" prefix notation. To simplify use, O-SLANG uses standard infix notation. Although they differ in appearance, the two axiom formats are semantically equivalent.

B.3.1 O-SLANG *Grammar.* This section defines the O-SLANG grammar. Words in bold typeface indicate language key words; brackets indicate optional items; a bar, |, represents the

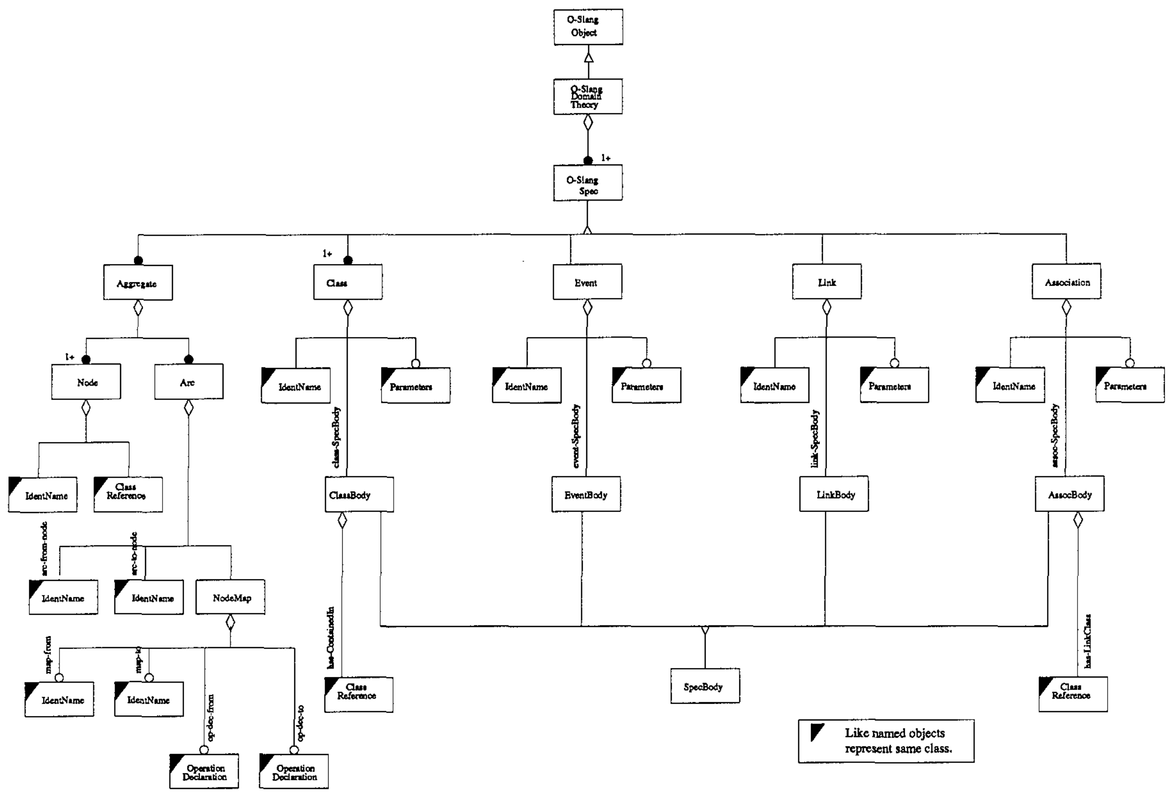

Figure B.1 O-SLANG Abstract Syntax Tree (Part I)

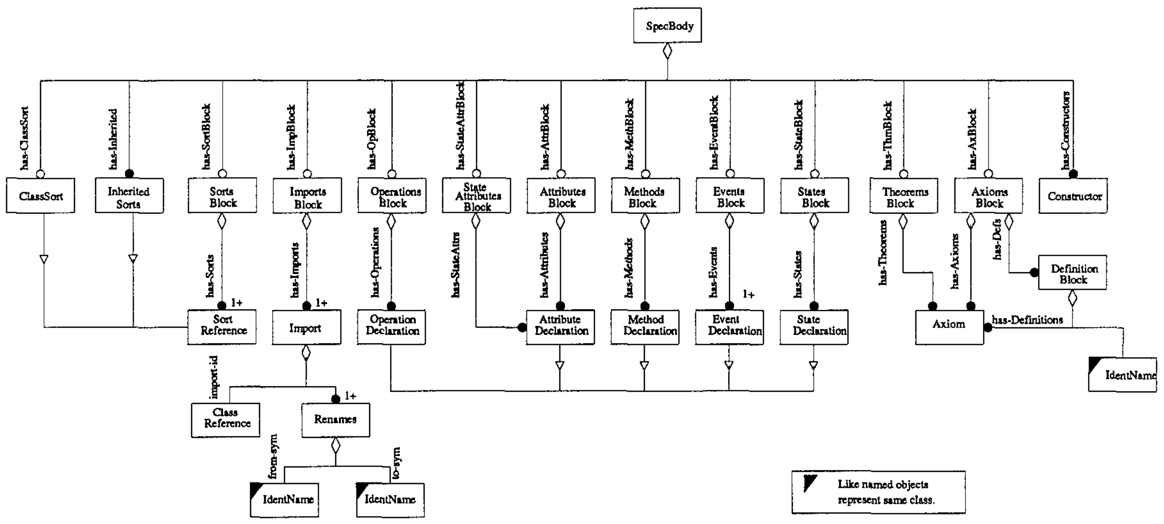

Figure B.2 O-SLANG Abstract Syntax Tree (Part II)

Event	Link	Association
Class Sort	Class Sort	Class Sort
	Import	Import
Sort-Ref	Sort-Ref	Sort-Ref
	Operation	Operation
	Attribute	Attribute
	Method	Method
Event-Op		
	Constructor	Constructor
Axiom	Axiom	Axiom
Theorem	Theorem	Theorem

Figure B.3 O-SLANG Features Sublist

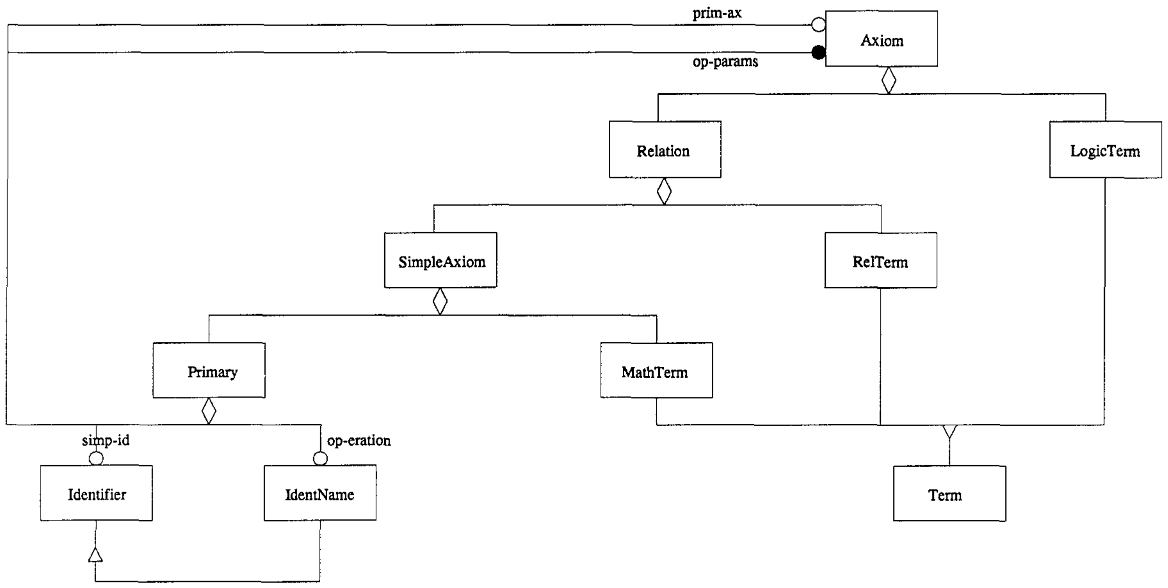

Figure B.4 O-SLANG Abstract Syntax Tree (Part III)

choice operator; and the * and + operators indicate zero or one or more of the preceding items are allowable.

O-Slang-DomainTheory → *Spec*$^+$

Spec → **class** *Class* | **abstract-class** *AbClass* | **event** *Event* | **link** *Link*
| **association** *Association* | **aggregate** *Aggregate*

Class → *Id* [*Parameter* [, *Parameter*]*] **is** *ClassBody*

AbClass → *Id* [*Parameter* [, *Parameter*]*]] **is** *AbclassBody*

Event → *Id* [*Parameter* [, *Parameter*]*]] **is** *EventBody*

Link → *Id* [*Parameter* [, *Parameter*]*]] **is** *LinkBody*

Association → *Id* [*Parameter* [, *Parameter*]*]] **is** *AssocBody*

Aggregate → *Id* **is nodes** *Node*[,*Node*]* **arcs** *Arc*[,*Arc*]* **end-aggregate**

Node → [*Id* :] *Id*

Arc → *Node* − > *Node* [: { *NodeMap*[,*NodeMap*]* }]

NodeMap → *Node* − > *Node* | (*OperationDecl*) − > (*OperationDecl*))

Parameters → *Id* : *Id*

ClassBody → **class-sort** *ClassSort*
  contained-class *Id*[,*Id*]*
  imports *Id*[,*Id*]*
  sorts *Id*[,*Id*]*
  sort-axioms *SortAxiom*[;*SortAxiom*]*
  operations *OperationBlock*
  attributes *AttributeBlock*
  state-attributes *StateAttributeBlock*
  methods *MethodBlock*
  states *StateBlock*
  events *EventBlock*
  axioms *AxiomBlock*
  theorems *TheoremBlock*
  end-class

AbClassBody → **class-sort** *ClassSort*
  contained-class *Id*[,*Id*]*
  imports *Id*[,*Id*]*
  sorts *Id*[,*Id*]*
  sort-axioms *SortAxiom*[;*SortAxiom*]*
  operations *OperationBlock*
  attributes *AttributeBlock*
  state-attributes *StateAttributeBlock*
  methods *MethodBlock*
  states *StateBlock*
  events *EventBlock*
  axioms *AxiomBlock*
  theorems *TheoremBlock*
  end-class

EventBody $\rightarrow$ **class-sort** *ClassSort*
imports *Id*[,*Id*]*
sorts *Id*[,*Id*]*
operations *OperationBlock*
attributes *AttributeBlock*
state-attributes *StateAttributeBlock*
methods *MethodBlock*
states *StateBlock*
events *EventBlock*
axioms *AxiomBlock*
theorems *TheoremBlock*
end-event

LinkBody $\rightarrow$ **class-sort** *ClassSort*
imports *Id*[,*Id*]*
sorts *Id*[,*Id*]*
operations *OperationBlock*
attributes *AttributeBlock*
state-attributes *StateAttributeBlock*
methods *MethodBlock*
states *StateBlock*
events *EventBlock*
axioms *AxiomBlock*
theorems *TheoremBlock*
end-link

AssocBody $\rightarrow$ **class-sort** *ClassSort*
link-class *LinkClass*
imports *Id*[,*Id*]*
sorts *Id*[,*Id*]*
sort-axioms *SortAxiom*[;*SortAxiom*]*
operations *OperationBlock*
attributes *AttributeBlock*
state-attributes *StateAttributeBlock*
methods *MethodBlock*
states *StateBlock*
events *EventBlock*
axioms *AxiomBlock*
theorems *TheoremBlock*
end-association

ClassSort $\rightarrow$ *Id*[,*Id*]* < *Id*[,*Id*]*

SortAxiom $\rightarrow$ *Id*[=*Id*]*

OperationBlock → *OperationDecl**

AttributeBlock → *OperationDecl**

StateAttributeBlock → *OperationDecl**

MethodBlock → *OperationDecl**

StateBlock → *OperationDecl**

EventBlock → *OperationDecl**

Constructor → **constructors** { *Id* [, *id*]* } **construct** *Id*

AxiomBlock → *AxiomDef*[;*AxiomDef*]*

Axiom-Def → *Axiom* | *DefinitionBlock*

TheoremsBlock → *Axiom*[;*Axiom*]*

DefinitionBlock → **definition of** *Id* **is** *Definition*[;*Definition*]* **end-definition**

OperationDecl → *Id* : *OpSig*

OpSig → *Id*[,*Id*]* − > *Id*[,*Id*]*

Axiom → *Relation* | *LogicTerm*

Relation → *SimpleAxiom* | *RelTerm*

SimpleAxiom → *Primary* | *MathTerm*

Primary → *SimpId* | *Tuple* | *Id* (*Axiom*[,*Axiom*]*) | (*Axiom*))

SimpleId → *symbol*

Tuple → < $SimpId^{++}$, >

LogicTerm → *And-Term* | *Or-Term* | *Not-Term* | *Iff-Term*
| *Implies-Term* | *Uquant-Term* | *Equant-Term*

And-Term → *Axiom* & *Relation*

Or-Term → *Axiom* | *Relation*

Not-Term → *Relation*

Iff-Term → *Axiom* <=> *Relation*

Implies-Term → *Axiom* => *Relation*

Uquant-Term → ∀ (*Relation*[,*Relation*]*) *Relation*

Equant-Term → ∃ (*Relation*[,*Relation*]*) *Relation*

RelTerm → *Equals-Term* | *NotEquals-Term* | *User-Term*
| *LT-Term* | *LTE-Term* | *GT-Term* | *GTE-Term*

Equals-Term → *Relation* = *SimpleAxiom*

NotEquals-Term → *Relation* <> *SimpleAxiom*

User-Term → *Relation Id SimpleAxiom*

LT-Term → *Relation* < *SimpleAxiom*

LTE-Term → *Relation* <= *SimpleAxiom*

GT-Term → *Relation* > *SimpleAxiom*

GTE-Term → *Relation* >= *SimpleAxiom*

MathTerm → *Add-Term* | *Sub-Term* | *Mult-Term* | *Div-Term*

Add-Term → *SimpleAxiom* + *Primary*

Sub-Term → *SimpleAxiom* − *Primary*

Mult-Term → *SimpleAxiom* ∗ *Primary*

Div-Term → *SimpleAxiom* / *Primary*

SimpId → *integer* | *real* | *Id*

Id → [*symbol* .] *symbol*

B.4 O-SLANG *Semantics*

The semantics of O-SLANG specifications are defined by the underlying SLANG translations. Most of this translation is straightforward. Sorts in O-SLANG map to sorts in SLANG; operations, attributes, methods, events, and states map to operations in SLANG; and imports, theorems, axioms, and definitions in O-SLANG map to identical constructs in SLANG. More complex translations are required for container classes, associations, and inherited classes. Actually, the container classes and associations use the SLANG colimit operation and SET specification to build sets of objects as defined in Chapter VII.

B.4.1 Classes. An example of an O-SLANG class is shown Figure B.5. The class sort, sorts, operations, attributes, state-attributes, methods, states, and events define the signature of the underlying SLANG specification (Figure B.6). All O-SLANG sorts, including the class sort, become SLANG sorts while O-SLANG operations, attributes, state-attributes, methods, and events become SLANG operations. O-SLANG states map to SLANG constants, or nullary operations. Translation of the axioms from O-SLANG to SLANG is a straight-forward rewriting exercise. The axioms are translated from infix notation to prefix notation. The import mechanism works identically in both O-SLANG and SLANG and thus no translation is required. The sort axiom defined in Figure B.5 has no real purpose in this specification except to illustrate its use. The O-SLANG sort axiom only allows the equivalencing of sorts and thus is a subset of the SLANG sort axiom construct.

B.4.2 Class Sets. In O-SLANG, whenever a class is defined, a "class set" class is automatically created as shown in Figure B.7. Because the class set, denoted by a *contained-class* construct in the class specification, is a class whose class sort is a set of objects, the underlying SLANG specification becomes more complicated. Figure B.8 shows the underlying SLANG specifications that are generated by an O-SLANG class set specification.

```
class Acct is
import Amnt, Date
class sort Acct
sorts Acct-State
sort-axioms Amnt = Integer
operations
  attr-equal : Acct, Acct → Boolean
attributes
  date : Acct → Date
  bal : Acct → Amnt
state-attributes
  acct-state : Acct → Acct-State
methods
  create-acct : Date → Acct
  credit, debit : Acct, Amnt → Acct
states
  ok, overdrawn : → Acct-State
events
  new-acct : Date → Acct
  deposit, withdrawal : Acct, Amnt → Acct
axioms
 % state uniqueness and invariant axioms
  ok ≠ overdrawn;
  ∀ (a: Acct) acct-state(a) = ok ⇒ bal(a) ≥ 0;
  ∀ (a: Acct) acct-state(a) = overdrawn ⇒ bal(a) < 0;
 % operation definitions
  ∀ (a,a1: Acct) attr-equal(a, a1) ⇒ date(a) = date(a1) ∧ bal(a) = bal(a1);
 % method definitions
  ∀ (d: Date) date(create-acct(d)) = d ∧ bal(create-acct(d)) = 0;
  ∀ (a: Acct, x: Amnt) bal(credit(a,x)) = bal(a) + x
     ∧ date(credit(a,x)) = date(a) ∧ rate(credit(a,x)) = rate(a)
     ∧ int-date(credit(a,x)) = int-date(a) ∧ check-cost(credit(a,x)) = check-cost(a);
 % event definitions
  ∀ (d: Date) acct-state(new-acct(d))=ok ∧ attr-equal(new-acct(d), create-acct(d))
  ∀ (a: Acct, x: Amnt) acct-state(a)=ok
     ⇒ acct-state(deposit(a,x))=ok ∧ attr-equal(deposit(a,x), credit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=overdrawn ∧ bal(a) + x ≥ 0
     ⇒ acct-state(deposit(a,x))=ok ∧ attr-equal(deposit(a,x), credit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=overdrawn ∧ bal(a) + x < 0
     ⇒ acct-state(deposit(a,x))=overdrawn ∧ attr-equal(deposit(a,x), credit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=ok ∧ bal(a) ≥ x
     ⇒ acct-state(withdrawal(a,x))=ok ∧ attr-equal(withdrawal(a,x), debit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=ok ∧ bal(a) < x
     ⇒ acct-state(withdrawal(a,x))=overdrawn ∧ attr-equal(withdrawal(a,x), debit(a,x));
  ∀ (a: Acct, x: Amnt) acct-state(a)=overdrawn
     ⇒ acct-state(withdrawal(a,x))=overdrawn ∧ attr-equal(withdrawal(a,x), a)
end-class
```

Figure B.5 Object Class

The *Acct-Class-Colimit* specification creates a specification with a set of *Acct* objects. The *Acct-Class-Set* specification simply renames the set to *Acct-Class.* This renamed specification is then included into the *Acct-Class* specification where the class operation and axioms defined in the O-SLANG class specification are translated into SLANG operations and axioms. The colimit and renaming specification are automatically generated based on the class name specified in the *contained-class* construct on the O-SLANG specification.

B.4.3 Communication. An example of an O-SLANG event theory is shown in Figure B.9. This is translated as defined above for a class into a SLANG specification that defines a theory

```
spec Acct is
  import Amnt, Date
  sorts Acct, Acct-State
  sort-axiom Amnt = Integer
  op attr-equal : Acct, Acct -> Boolean
  op date : Acct -> Date
  op bal : Acct -> Amnt
  op acct-state : Acct -> Acct-State
  op create-acct : Date -> Acct
  op credit, debit : Acct, Amnt -> Acct
  const ok, overdrawn : Acct-State
  op new-acct : Date -> Acct
  op deposit, withdrawal : Acct, Amnt -> Acct
 % state uniqueness and invariant axioms
  axiom (not (equal ok overdrawn))
  axiom (fa (a: Acct) (implies (equal (acct-state a) ok) (greater-than-or-equal (bal a) zero)))
  axiom (fa (a: Acct) (implies (equal (acct-state a) overdrawn) (less-than (bal a) zero)))
 % operation definitions
  axiom (fa (a a1: Acct) (implies (attr-equal a a1) (and (equal (date a) (date a1)) (equal (bal a) = (bal a1)))))
 % method definitions
  axiom (fa (d: Date) (and (equal (date (create-acct d)) d) (equal (bal (create-acct d)) zero)))
  axiom (fa (a: Acct x: Amnt) (and (and (equal (bal (credit a x)) (plus (bal a) x)) (date (credit a x)) = (date a))
     (and (equal (rate (credit a x)) (rate a))
        (and (equal (int-date (credit a x)) (int-date a)) (equal (check-cost (credit a x)) (check-cost a))))))
 % event definitions
  axiom (fa (d: Date) (and (equal (acct-state (new-acct d)) ok) (attr-equal (new-acct d) (create-acct d))))
  axiom (fa (a: Acct x: Amnt) (implies (equal (acct-state a) ok)
     (and (equal (acct-state (deposit a x)) ok) (attr-equal (deposit a x) (credit a x)))))
  axiom (fa (a: Acct x: Amnt) (implies (and (equal (acct-state a) overdrawn)
        (greater-than-or-equal (plus (bal a) x) zero))
     (and (equal (acct-state (deposit a x)) ok) (attr-equal (deposit a x) (credit a x)))))
  axiom (fa (a: Acct x: Amnt) (implies (and (equal (acct-state a) overdrawn) (less-than (plus (bal a) x) zero))
     (and (equal (acct-state (deposit a x)) overdrawn) (attr-equal (deposit a x) (credit a x)))))
  axiom (fa (a: Acct x: Amnt) (implies (and (equal (acct-state a) ok) (greater-than-or-equal (bal a) x))
     (and (equal (acct-state (withdrawal a x)) ok) (attr-equal (withdrawal a x) (debit a x)))))
  axiom (fa (a: Acct x: Amnt) (implies (and (equal (acct-state a) ok) (less-than (bal a) x))
     (and (equal (acct-state (withdrawal a x)) overdrawn) (attr-equal (withdrawal a x) (debit a x)))))
  axiom (fa (a: Acct x: Amnt) (implies (equal (acct-state a) overdrawn)
     (and (equal (acct-state (withdrawal a x)) overdrawn) (attr-equal (withdrawal a x) a))))
end-spec
```

Figure B.6 Underlying SLANG Specification

signature. This signature is used in a colimit operation (via an aggregate definition) to unify an event in one class with an event in a second class.

B.4.4 Links. A link is used to define a general relationship between two classes. An example of an O-SLANG link is shown in Figure B.10 with the SLANG equivalent specification shown in Figure B.11. Because a link is created without knowing the actual classes it is associating, the sorts X and Y do not have meaning until they are unified with a class sort in the aggregate colimit. All sorts and operations are translated exactly like those of a class as described in Section B.4.1.

B.4.5 Associations. An association is a set of links and thus has a very similar translation to that of a class set. However, instead of using just the class specifications related through the

```
class Acct-Class is
contained-class ACCT
class sort Acct-Class
events
  new-acct-class : → Acct-Class
  withdrawal : Acct-Class, Amnt → Acct-Class
  deposit : Acct-Class, Amnt → Acct-Class
axioms
  new-acct-class() = empty-set;
  ∀ (a: Acct, ac: Acct-Class, x: Amnt) a ∈ ac ⇔ deposit(a,x) ∈ deposit(ac,x);
  ∀ (a: Acct, ac: Acct-Class, x: Amnt) a ∈ ac ⇔ withdrawal(a,x) ∈ withdrawal(ac,x)
end-class
```

Figure B.7 O-SLANG Class Set Specification

```
spec Acct-Class-Colimit is
  colimit of diagram
    nodes TRIV, ACCT, SET
    arcs  TRIV -> ACCT : {E -> Acct}
          TRIV -> SET : {}
  end-diagram

spec Acct-Class-Set is
  translate ACCT-CLASS-COLIMIT
  by {Set -> Acct-Class, E -> Acct}

spec Acct-Class is
  import ACCT-CLASS-SET
  op new-acct-class : -> Acct-Class
  op withdrawal : Acct-Class, Amnt -> Acct-Class
  op deposit : Acct-Class, Amnt -> Acct-Class
  axiom (equal (new-acct-class) empty-set)
  axiom (fa (a: Acct ac: Acct-Class x: Amnt) (iff (in a ac) (in (deposit a x) (deposit ac,x))))
  axiom (fa (a: Acct ac: Acct-Class x: Amnt) (iff (in a ac) (in (withdrawal a x) (withdrawal ac x))))
end-spec
```

Figure B.8 SLANG Class Set Specification

link specification, the sets of the related classes must be used to allow for various OMT association multiplicities (optional, many, ordered, etc.). This does not present a problem since every class has an associated class set specification already defined. An example of an O-SLANG association is shown in Figure B.12 with its SLANG counterparts shown in Figure B.13.

B.4.6 Aggregates. Aggregates are a unique type of O-SLANG specification. Aggregates define a colimit operation over previously defined classes, associations, and events that make up the aggregate specification. An aggregate specification does not have the ability to add attributes,

```
event Event is
class sort Event-Sort
sorts X, Y
events
  event : Event-Sort, X, Y → Event-Sort
end-class
```

Figure B.9 Event Theory

```
link XY-Link is
class sort XY-Link
sorts X, Y
operations
  attr-equal : XY-Link, XY-Link → Boolean
attributes
  x-obj : XY-Link → X
  y-obj : XY-Link → Y
methods
  create-xy-link : X, Y → XY-Link
events
  new-xy-link : X, Y → XY-Link
axioms
  % operation definition
  ∀ (x1,x2: X) attr-equal(x1,x2) ⇔ x-obj(x1) = x-obj(x2) ∧ y-obj(x1) = y-obj(x2);
  % create method definition
  ∀ (x: X, y: Y) x-obj(create-xy-link(x,y)) = x ∧ y-obj(create-xy-link(x,y)) = y;
  % new event definition
  ∀ (x: X, y: Y) attr-equal(new-xy-link(x,y), create-xy-link(x,y))
end-link
```

Figure B.10 O-SLANG Link Specification

```
spec XY-Link is
  sorts X, Y, XY-Link
  op attr-equal : XY-Link, XY-Link -> Boolean
  op x-obj : XY-Link -> X
  op y-obj : XY-Link -> Y
  op create-xy-link : X, Y -> XY-Link
  op new-xy-link : X, Y -> XY-Link
  axiom (fa (x1:X x2:X) (iff (attr-equal x1 x2) (and (equal (x-obj x1) (x-obj x2)) (equal (y-obj x1) (y-obj x2)))))
  axiom (fa (x:X y:Y) (and (equal (x-obj (create-xy-link x y)) x) (equal (y-obj (create-xy-link x y)))))
  axiom (fa (x:X y:Y) (attr-equal (new-xy-link x y) (create-xy-link x y)))
end-spec
```

Figure B.11 SLANG Link Specification

methods, events, constraints, etc. These additions are made through an extension of the aggregate specification using a class specification that imports the aggregate. The nodes of an aggregate are the classes, associations, and events included in the aggregate while the arcs are the specification morphisms between the nodes that define the relationships between the nodes. Besides simply combining a number of class and association specifications into a single aggregate specification, the colimit operation unifies sorts and operations defined in separate classes and associations.

An example of an O-SLANG aggregate is shown in Figure B.14. The diagram of the aggregate is shown in Figure B.15. The SLANG version of the aggregate is shown in Figure B.16.

Once the colimit specification is specified, new operations and axioms are added to an extension of colimit specification. This extension is created by importing the colimit specification into a class specification and adding new operations and axioms as defined in Section B.4.1.

```
association XY-Assoc is
link-class XY-Link
class sort XY-Assoc
sorts X-Set, Y-Set
methods
  image : XY-Assoc, Y → X-Set
  image : XY-Assoc, X → Y-Set
events
  new-xy-assoc : → XY-Assoc
axioms
 % multiplicity axioms
  ∀ (l: XY-Assoc, y: Y) size(image(l, y)) ≥ 1;
  ∀ (l: XY-Assoc, x: X) size(image(l, x)) = 1;
 % new event definition
  new-xy-assoc() = empty-set;
 % image definitions
  ∀ (xy:XY-Assoc, l:XY-Link, y:Y) (l ∈ xy & y-obj(l) = y) ⇒ x-obj(l) ∈ image(xy,y);
  ∀ (xy:XY-Assoc, l:XY-Link, x:X) (l ∈ xy & x-obj(l) = x) ⇒ y-obj(l) ∈ image(xy,x)
end-association
```

Figure B.12 O-SLANG Association Specification

```
spec XY-Assoc-Colimit is
  colimit of diagram
    nodes TRIV, XY-LINK, SET
    arcs  TRIV -> XY-LINK : {E -> XY-Link}
          TRIV -> SET : {}
  end-diagram

spec XY-Assoc-Set is
  translate XY-ASSOC-COLIMIT
  by {Set -> XY-Assoc, E -> XY-Link}

spec XY-Assoc is
  import XY-ASSOC-SET, X-CLASS, Y-CLASS
  op image : XY-Assoc, Y → X-Class
  op image : XY-Assoc, X → Y-Class
  op new-xy-assoc : → XY-Assoc
  axiom (fa (l:XY-Assoc y:Y) (greater-than-or-equal (size (image l y)) one))
  axiom (fa (l:XY-Assoc x:X) (equal (size (image l x)) one))
  axiom (equal (new-xy-assoc) empty-set)
  axiom (fa (xy:XY-Assoc l:XY-Link y:Y) (iff (and (in l xy) (equal (y-obj l) y)) (in (x-obj l) (image xy y))))
  axiom (fa (xy:XY-Assoc l:XY-Link x:X) (iff (and (in l xy) (equal (x-obj l) x)) (in (y-obj l) (image xy x))))
end-spec
```

Figure B.13 SLANG Association Specification

B.4.7 Inheritance. To this point, translation of O-SLANG specifications to SLANG specifications is relatively simple; however, inheritance is more complex. Because SLANG has a limited subsorting feature and makes no allowance for multiple subsorting, subsorting, as defined in O-SLANG, is *simulated* in SLANG. This simulation requires defining a subsort predicate based on the defined attributes and copying each superclass attribute and operation into each subclass.

B.4.7.1 Single Inheritance. A typical example of O-SLANG single inheritance is shown in Figures B.17 and B.18. *Person* is the supersort and *Student* is a subsort of *Person.* The Student class defines only one new attribute, *GPA.* (For simplicity, no events have been defined for

```
aggregate XY-Aggregate is
    nodes INTEGER, SET-1: SET, SET-2: SET, SET-3: SET,
          X-CLASS, Y-CLASS, XY-Assoc
    arcs  SET-1 → X-CLASS : {E → X, SET → X-Class},
          SET-1 → XY-Assoc : {E → X SET → X-Class},
          SET-2 → Y-CLASS : {E → Y, SET → Y-Class},
          SET-2 → XY-Assoc : {E → Y, SET → Y-Class},
          SET-3 → XY-Assoc : {E → XY-Link, SET → XY-Assoc},
          INTEGER → SET-1 : {},
          INTEGER → SET-2 : {}
          INTEGER → SET-3 : {}
end-aggregate
```

Figure B.14 O-SLANG Aggregation Specification

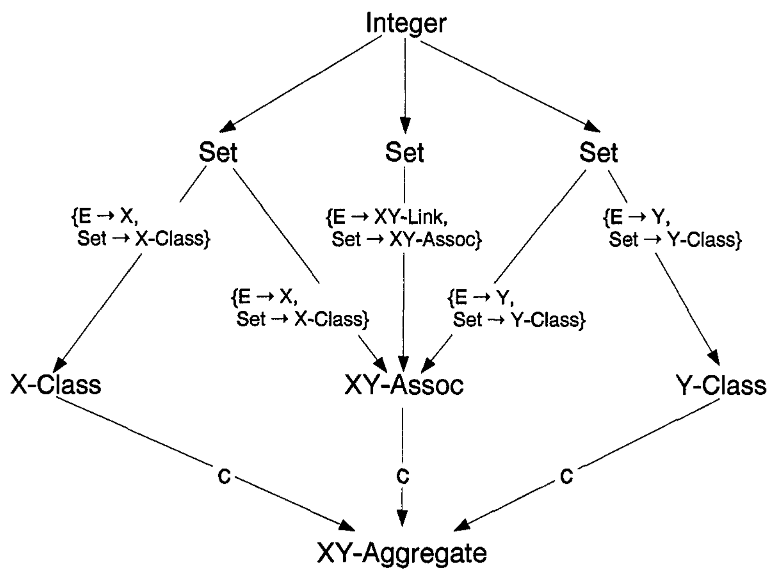

Figure B.15 Aggregation Composition

```
spec XY-Aggregate is
  colimit of diagram
    nodes INTEGER, SET-1: SET, SET-2: SET, SET-3: SET,
          X-CLASS, Y-CLASS, XY-Assoc
    arcs  SET-1 → X-CLASS : {E → X},
          SET-1 → XY-Assoc : {E → X},
          SET-1 → X-CLASS : {SET → X-Class},
          SET-1 → XY-Assoc : {SET → X-Class},
          SET-2 → Y-CLASS : {E → Y},
          SET-2 → XY-Assoc : {E → Y},
          SET-2 → Y-CLASS : {SET → Y-Class},
          SET-2 → XY-Assoc : {SET → Y-Class},
          SET-3 → XY-Assoc : {E → XY-Link, SET → XY-Assoc},
          INTEGER → SET-1 : {},
          INTEGER → SET-2 : {},
          INTEGER → SET-3 : {}
  end-diagram
```

Figure B.16 SLANG Aggregation Specification

the *Person*, *Student*, or *Faculty* classes defined below or in their SLANG counterparts. These would be defined as usual and translated the same as the methods shown below.)

```
class Person is
import Date, Sex
class sort Person
attributes
  name : Person → String
  birthday : Person → Date
  ssan : Person → Integer
  sex : Person → Sexuality
methods
  create-person : String → Person
  change-name : Person, String → Person
events
axioms
  ssan(p) > 0;
 % create-person
  name(create-person(n)) = n;
  birthday(create-person(n)) = default-date;
  ssan(create-person(n)) = 0;
  sex(create-person(n)) = male;
 % change-name
  name(change-name(p,n)) = n;
  birthday(change-name(p,n)) = birthday(p);
  ssan(change-name(p,n)) = ssan(p);
  sex(change-name(p,n)) = sex(p)
end-class
```

Figure B.17 O-SLANG Person Superclass

```
class Student is
import Person, Gpa
class sort Student < Person
attributes
  gpa : Student → Gpa
methods
  create-student : String → Student
axioms
 % create-student
  name(create-student(n)) = name(create-person(n));
  birthday(create-student(n)) = birthday(create-person(n));
  ssan(create-student(n)) = ssan(create-person(n));
  sex(create-student(n)) = sex(create-person(n));
  gpa(create-student(n)) = 0
end-class
```

Figure B.18 O-SLANG Student Subclass

Inheritance, as defined in Section 4.4, requires that (1) all objects of a subclass be objects of each of its superclasses, (2) that all operations defined on a superclass are defined on the subclass, and (3) that the semantics of those operations are identical on attributes defined in the superclass. In SLANG, subsorting is accomplished via sort axioms as shown below.

sort-axiom *subsort* = *supersort* | *predicate*

The sort axiom states that the subsort consists of all values from the supersort where the *predicate* is true. Thus, given the right predicate definition, SLANG sort axioms can directly define simple inheritance. Figures B.19 and B.20 show the SLANG translations of the *Person* and *Student* classes defined above.

```
spec PERSON is
  import DATE, SEX
  sort Person, String
  op name : Person - > String
  op birthday : Person - > date
  op ssan : Person - > Integer
  op sex : Person - > Sexuality
  op create-person : String - > Person
  op change-name : Person, String - > Person
  axiom (greater-than (ssan p) zero)
  axiom (equal (name (create-person n)) n)
  axiom (equal (birthday (create-person n)) default-date)
  axiom (equal (ssan (create-person n)) zero)
  axiom (equal (sex (create-person n)) male)
  axiom (equal (name (change-name p n)) n)
  axiom (equal (birthday (change-name p n)) (birthday p))
  axiom (equal (ssan (change-name p n)) (ssan p))
  axiom (equal (sex (change-name p n)) (sex p))
end-spec
```

Figure B.19 SLANG Person Superclass

The *Person* class translates to SLANG as described in Section B.4.1. It is in the *Student* class that the inheritance simulation takes place. Because SLANG is very strongly typed, an operation defined on a supersort is not automatically defined on its subsorts. SLANG does provide a built-in inclusion function, *relax*, that maps elements of the subsort to their corresponding elements of the supersort. Thus the axiom

(equal (name s) (name ((relax student?) s)))

or,

$$name(s) = name(relax(student?)(s))$$

states the name of a student, s, is equal to the name of an equivalent person in the supersort denoted by *((relax student?) s)*. Thus the *relax* operator maps s from the student subsort to the *Person* supersort by relaxing the *student?* predicate. Therefore, for each operation defined in the *Person* class, an equivalent operation must be defined on the *Student* class. Axioms are used (as shown above) to ensure that the semantics of the operations on superclass attributes are equivalent.

```
spec STUDENT is
  import PERSON
  sort Student, Gpa
  sort-axiom Student = Person | student?
  op name : Student -> String
  op birthday : Student -> Date
  op ssan : Student -> Integer
  op sex : Student -> Sexuality
  op gpa : Student -> Gpa
  op create-student : String -> Student
  op change-name : Student, String -> Student
  op student? : Person -> Boolean
  axiom (fa (p:person s:student)
            (implies (equal p ((relax student?) s))
                     (equal (student? p)
                            (ex (r:gpa) (equal (gpa s) r)))))
  axiom (equal (name s) (name ((relax student?) s)))
  axiom (equal (birthday s) (birthday ((relax student?) s)))
  axiom (equal (ssan s) (ssan ((relax student?) s)))
  axiom (equal (sex s) (sex ((relax student?) s)))
  axiom (equal (name (create-student n)) (name (create-person n)))
  axiom (equal (birthday (create-student n))
               (birthday (create-person n)))
  axiom (equal (ssan (create-student n)) (ssan (create-person n)))
  axiom (equal (sex (create-student n)) (sex (create-person n)))
  axiom (equal (gpa (create-student n)) zero)
  axiom (equal (name (change-name s n))
               (name (change-name ((relax student?) s) n)))
  axiom (equal (birthday (change-name s n))
               (birthday (change-name ((relax student?) s) n)))
  axiom (equal (ssan (change-name s n))
               (ssan (change-name ((relax student?) s) n)))
  axiom (equal (sex (change-name s n))
               (sex (change-name ((relax student?) s) n)))
  axiom (equal (gpa (change-name s n)) (gpa s))
end-spec
```

Figure B.20 SLANG Student Subclass

The predicate *student?* is defined by the signature and axiom shown below.

```
op student? : Person -> Boolean
axiom (fa (p:person s:student)
          (implies (equal p ((relax student?) s))
                   (equal (student? p)
                          (ex (r:gpa) (equal (gpa s) r)))))
```

In conventional notation this would be as follows.

$$\forall\ (\text{p:person s:student})\ p = \text{relax}(\text{student?})(s)$$
$$\Rightarrow \text{student?}(p) = (\exists\ (\text{r:gpa})\ \text{gpa}(s) = r)$$

Basically, the *student?* predicate states that for an object to be in the *Student* class, the attribute, *gpa*, must be defined on that object. In general, the subclass predicate is defined by requiring all attributes of the class to be defined on its objects. Thus SLANG allows us to determine if an element of the supersort is a member of a subsort and to create an element of the supersort from elements in the subsort; unfortunately, it does not provide a simple or elegant way to accomplish these operations. Therefore, when translating a subclass definition, not only must the superclass specification be imported, but an operation for each operation in the superclass must be created and explicitly defined as equivalent to the superclass operation acting on a relaxed subsort element.

The translation of O-SLANG single inheritance satisfies the three requirements for inheritance described above. First, because SLANG supports simple subsorting, all objects of a subclass are objects of each of its superclasses, even though use of the *relax* operation is required. Second, all operations of the superclass sort are defined on the subsort since they are explicitly redefined (using the same names and parameters) on the subclass objects. Finally, the semantics of superclass operations are identical on attributes defined in the superclass since the subclass operations are equivalenced to the superclass operations over the superclass attributes. Therefore, the translation described above fully specifies the semantics of O-SLANG single inheritance.

B.4.7.2 Multiple Inheritance. Now that I have defined the O-SLANG to SLANG translation for single inheritance, I extend this translation to multiple inheritance. Unfortunately, this extension is not straightforward. SLANG does not allow a sort to be a subsort of more than one supersort; therefore, this requirement must be simulated. The requirements that all superclass operations be defined on the subclass and that the semantics of the operations are equivalent over superclass attributes are translated the same way as for single inheritance. To set up an example, assume a *Faculty* class is defined as a subclass of *Person* as shown in Figure B.21 and B.22.

```
class Faculty is < Person
import Person, Academic-Rank
class sort Faculty
attributes
  academic-rank : Faculty → Academic-Rank
methods
  create-faculty : String → Faculty
axioms
 % create-faculty
  name(create-faculty(n)) = name(create-person(n));
  birthday(create-faculty(n)) = birthday(create-person(n));
  ssan(create-faculty(n)) = ssan(create-person(n));
  sex(create-faculty(n)) = sex(create-person(n));
  academic-rank(create-faculty(n)) = instructor
end-class
```

Figure B.21 O-SLANG Faculty Subclass

```
spec FACULTY is
  import PERSON, ACADEMIC-RANK
  sort Faculty
  sort-axiom Faculty = Person | faculty?
  op name : Faculty -> String
  op birthday : Faculty -> Date
  op ssan : Faculty -> Integer
  op sex : Faculty -> Sexuality
  op academic-rank : Faculty -> Academic-Rank
  op create-faculty : String -> Faculty
  op change-name : Faculty, String -> Faculty
  op faculty? : Person -> Boolean
  axiom (fa (p:person f:faculty)
              (implies (equal p ((relax faculty?) f))
                       (equal (faculty? p)
                              (ex (r:Academic-Rank) (equal (academic-rank f) r)))))
  axiom (equal (name f) (name ((relax faculty?) f)))
  axiom (equal (birthday f) (birthday ((relax faculty?) f)))
  axiom (equal (ssan f) (ssan ((relax faculty?) f)))
  axiom (equal (sex f) (sex ((relax faculty?) f)))
  axiom (equal (name (create-faculty n)) (name (create-person n)))
  axiom (equal (birthday (create-faculty n))
               (birthday (create-person n)))
  axiom (equal (ssan (create-faculty n)) (ssan (create-person n)))
  axiom (equal (sex (create-faculty n)) (sex (create-person n)))
  axiom (equal (academic-rank (create-faculty n)) instructor)
  axiom (equal (name (change-name f n))
               (name (change-name ((relax faculty?) f) n)))
  axiom (equal (birthday (change-name f n))
               (birthday (change-name ((relax faculty?) f) n)))
  axiom (equal (ssan (change-name f n))
               (ssan (change-name ((relax faculty?) f) n)))
  axiom (equal (sex (change-name f n))
               (sex (change-name ((relax faculty?) f) n)))
  axiom (equal (academic-rank (change-name f n)) (academic-rank f))
end-spec
```

Figure B.22 SLANG Faculty Subclass

What is desired is to create a teaching assistant class, *TA*, that is a subclass of both the *Student* class and the *Faculty* class. The O-SLANG for such a class is shown in Figure B.23. By importing both *Student* and *Faculty* classes and requiring that the *TA* class sort be a subsort of both the *Student* and *Faculty* class sorts, multiple inheritance is achieved. The operations have been imported and defined over both *Student* and *Faculty*, and thus are automatically defined over *TA* objects since the *TA* objects are a subset of both the *Student* and *Faculty* objects. A new operation, *create-ta*, is defined in accordance with the appropriate *Student* and *Faculty* create operations.

The translation of *TA* to SLANG is not quite as simple as for single inheritance. As defined in the theory-based object model in Chapter VII, multiple inheritance is defined by the colimit of the superclass specifications (with the shared part defined by common superclasses and any shared data type specifications). Since both *Student* and *Faculty* are subclasses of the *Person* class, the

```
class TA is
   Student, Faculty
class sort TA < Student, Faculty
methods
  create-ta : String → TA
axioms
  % create-ta
  name(create-ta(n)) = name(create-person(n));
  birthday(create-ta(n)) = birthday(create-person(n));
  ssan(create-ta(n)) = ssan(create-person(n));
  sex(create-ta(n)) = sex(create-person(n));
  gpa(create-ta(n)) = gpa(create-student(n));
  academic-rank(create-ta(n)) = academic-rank(create-faculty(n))
end-class
```

Figure B.23 O-SLANG TA Subclass

diagram consisting of *Person*, *Student*, and *Faculty* (and the appropriate morphisms between them) defines the colimit specification used to create the *TA* specification (Figure B.24).

Once the colimit specification has been created, it is imported into the *TA* SLANG specification where it is extended by adding the class sort, *TA*, and constructing the subsort equivalences. Because SLANG only supports a single subsort definition, the *TA* sort must be defined to be a subsort of the lowest common supersort of all of the superclasses used to define the subclass. In this example, since *Student* and *Faculty* are both subclasses of *Person*, *TA* is defined to be a subclass of *Person* as well. If some of the superclasses are not descendants of a common superclass, the *object-class sort* (the sort which includes all object names) is used. This situation is not discussed further except to state that all object class sorts are, by definition, subsorts of the object-class sort.

The fact that *TA* is a subsort of both *Person* and *Faculty* classes is captured by the definition of a subsort predicate *ta?* as defined below.

```
axiom (fa (t:ta) (ex (f:faculty) (equal ((relax ta?) t)
                                        ((relax faculty?) f))))
axiom (fa (t:ta) (ex (s:student) (equal ((relax ta?) t)
                                        ((relax student?) s))))
```

The first axiom states that the predicate for every object in the *TA* class there is an object in the *Faculty* class such that if the *ta?* predicate and *faculty?* predicates are relaxed, they are the same underlying *Person*. The second axiom does the same for the *Student* class.

The next set of axioms define *ta-faculty* and *ta-student* operations that, in essence, relax a *TA* object into a *Faculty* or *Student* object. The definitional axiom states that if a relaxed *TA* object is equal to a relaxed *Faculty/Student* object then the result of the *ta-faculty/ta-student* operation is the *Faculty/Student* object. These relaxation operations are used to define the equivalence of attributes defined in each of the superclasses respectively.

```
op ta-faculty : Ta - > Faculty
axiom (implies (equal ((relax ta?) t) ((relax faculty?) f))
                (equal (ta-faculty t) f))
op ta-student : Ta - > Student
axiom (implies (equal ((relax ta?) t) ((relax student?) s))
                (equal (ta-student t) s))
```

The operations are copied from the superclasses as defined for single inheritance. The only difference is that the definition of the subclass operations must be based on where the operation is originally defined. Thus if the operation is defined in *Person*, the *relax* operation is used in the definitions whereas if the operation is defined in *Faculty*, the *ta-faculty* relaxation operation is used in the definition.

The translation of O-SLANG multiple inheritance satisfies the three requirements for inheritance defined in Section B.4.7.1. First, by simulating multiple subsorting through the definition of superclass relaxation operations, all objects of the subclass are objects of each of its superclasses. Second, all operations of each superclass are defined on the subsort since they are explicitly redefined (using the same names and parameters) on the subclass objects. Finally, the semantics of superclass operations are identical on operations defined in the superclasses since the results of the subclass operations are equivalenced to the superclass operations over the superclass attributes. Therefore, the translation described above fully specifies the semantics of O-SLANG multiple inheritance.

B.5 Summary

This Appendix defined the interpretation of O-SLANG in SPECWARE's SLANG, thus defining the syntax and semantics of O-SLANG. Although most O-SLANG features translate simply, almost trivially, into SLANG, inheritance requires a slightly more untidy approach due to *Slang*'s restricted notion of subsorting. While the inheritance translation increases the level of translation sophistication, the end results satisfy the requirements of the substitution property as defined in Chapter IV.

```
spec FACULTY-STUDENT-COLIMIT is
  colimit of
    diagram nodes FACULTY, STUDENT, PERSON
    arcs          PERSON - > FACULTY:
                          {(FEMALE: SEXUALITY) - > (FEMALE: SEXUALITY),
                          (MALE: SEXUALITY) - > (MALE: SEXUALITY),
                          SEXUALITY - > SEXUALITY,
                          (YEAR: DATE - > INTEGER) - > (YEAR: DATE - > INTEGER),
                          (MONTH: DATE - > INTEGER) - > (MONTH: DATE - > INTEGER),
                          (DAY: DATE - > INTEGER) - > (DAY: DATE - > INTEGER),
                          (DEFAULT-DATE: DATE) - > (DEFAULT-DATE: DATE), DATE - > DATE,
                          (MAX: INTEGER, INTEGER - > INTEGER)
                               - > (MAX: INTEGER, INTEGER - > INTEGER),
                          (MIN: INTEGER, INTEGER - > INTEGER)
                               - > (MIN: INTEGER, INTEGER - > INTEGER),
                          (TIMES: INTEGER, INTEGER - > INTEGER)
                               - > (TIMES: INTEGER, INTEGER - > INTEGER),
                          (MINUS: INTEGER, INTEGER - > INTEGER)
                               - > (MINUS: INTEGER, INTEGER - > INTEGER),
                          (IPLUS: INTEGER, INTEGER - > INTEGER)
                               - > (IPLUS: INTEGER, INTEGER - > INTEGER),
                          (GREATER-THAN-OR-EQUAL: INTEGER, INTEGER - > BOOLEAN)
                               - > (GREATER-THAN-OR-EQUAL: INTEGER, INTEGER - > BOOLEAN),
                          (LESS-THAN-OR-EQUAL: INTEGER, INTEGER - > BOOLEAN)
                               - > (LESS-THAN-OR-EQUAL: INTEGER, INTEGER - > BOOLEAN),
                          (GREATER-THAN: INTEGER, INTEGER - > BOOLEAN)
                               - > (GREATER-THAN: INTEGER, INTEGER - > BOOLEAN),
                          (LESS-THAN: INTEGER, INTEGER - > BOOLEAN)
                               - > (LESS-THAN: INTEGER, INTEGER - > BOOLEAN),
                          (TEN: INTEGER) - > (TEN: INTEGER),
                          (NINE: INTEGER) - > (NINE: INTEGER),
                          (EIGHT: INTEGER) - > (EIGHT: INTEGER),
                          (SEVEN: INTEGER) - > (SEVEN: INTEGER),
                          (SIX: INTEGER) - > (SIX: INTEGER),
                          (FIVE: INTEGER) - > (FIVE: INTEGER),
                          (FOUR: INTEGER) - > (FOUR: INTEGER),
                          (THREE: INTEGER) - > (THREE: INTEGER),
                          (TWO: INTEGER) - > (TWO: INTEGER),
                          (ONE: INTEGER) - > (ONE: INTEGER),
                          (ZERO: INTEGER) - > (ZERO: INTEGER), INTEGER - > INTEGER,
                          PERSON - > PERSON, STRING - > STRING,
                          (CREATE-PERSON: STRING - > PERSON)
                               - > (CREATE-PERSON: STRING - > PERSON),
                          (CHANGE-NAME: PERSON, STRING - > PERSON)
                               - > (CHANGE-NAME: PERSON, STRING - > PERSON),
                          (SEX: PERSON - > SEXUALITY) - > (SEX: PERSON - > SEXUALITY),
                          (SSAN: PERSON - > INTEGER) - > (SSAN: PERSON - > INTEGER),
                          (BIRTHDAY: PERSON - > DATE) - > (BIRTHDAY: PERSON - > DATE),
                          (NAME: PERSON - > STRING) - > (NAME: PERSON - > STRING)},
                  PERSON - > STUDENT: import-morphism
    end-diagram
```

Figure B.24 SLANG Faculty Student Colimit

```
spec TA is
  import FACULTY-STUDENT-COLIMIT
  sort Ta
  sort-axiom Ta = Person | ta?
  axiom (fa (t:ta) (ex (f:faculty) (equal ((relax ta?) t)
                                   ((relax faculty?) f))))
  axiom (fa (t:ta) (ex (s:student) (equal ((relax ta?) t)
                                   ((relax student?) s))))
  op ta-faculty : Ta -> Faculty
  axiom (implies (equal ((relax ta?) t) ((relax faculty?) f))
                 (equal (ta-faculty t) f))
  op ta-student : Ta -> Student
  axiom (implies (equal ((relax ta?) t) ((relax student?) f))
                 (equal (ta-student t) f))
  op name : Ta -> String
  op birthday : Ta -> Date
  op ssan : Ta -> Integer
  op sex : Ta -> Sexuality
  op academic-rank : Ta -> Academic-Rank
  op gpa : Ta -> Gpa
  op create-ta : String -> Ta
  op change-name : TA, String -> TA
  op ta? : Person -> Boolean
  axiom (fa (p:person t:ta)
            (implies (equal p ((relax ta?) t))
                     (equal (ta? p)
                            (and (faculty? p) (student? p)))))
  axiom (fa (t:ta) (equal (name t) (name ((relax ta?) t))))
  axiom (fa (t:ta) (equal (birthday t) (birthday ((relax ta?) t))))
  axiom (fa (t:ta) (equal (ssan t) (ssan ((relax ta?) t))))
  axiom (fa (t:ta) (equal (sex t) (sex ((relax ta?) t))))
  axiom (fa (t:ta) (equal (academic-rank t)
                          (academic-rank (ta-faculty t))))
  axiom (fa (t:ta) (equal (gpa t) (gpa (ta-student t))))
  axiom (equal (name (create-ta n)) (name (create-person n)))
  axiom (equal (birthday (create-ta n))
               (birthday (create-person n)))
  axiom (equal (ssan (create-ta n)) (ssan (create-person n)))
  axiom (equal (sex (create-ta n)) (sex (create-person n)))
  axiom (equal (gpa (create-ta n))
               (gpa (create-student n)))
  axiom (equal (academic-rank (create-ta n))
               (academic-rank (create-faculty n)))
  axiom (fa (t:ta) (equal (name (change-name t n))
                          (name (change-name ((relax ta?) t) n))))
  axiom (fa (t:ta) (equal (birthday (change-name t n))
                          (birthday (change-name ((relax ta?) t) n))))
  axiom (fa (t:ta) (equal (ssan (change-name t n))
                          (ssan (change-name ((relax ta?) t) n))))
  axiom (fa (t:ta) (equal (sex (change-name t n))
                          (sex (change-name ((relax ta?) t) n))))
  axiom (fa (t:ta) (equal (gpa (change-name t n))
                          (gpa (change-name (ta-student t) n))))
  axiom (fa (t:ta) (equal academic-rank (change-name t n))
                          (academic-rank (change-name (ta-faculty t) n))))
end-spec
```

Figure B.25 SLANG TA Subclass

Appendix C. Generic OMT and O-SLANG *ASTs*

C.1 Introduction

This appendix contains the definition of the O-SLANG and GOMT abstract syntax trees. The notation used is described in Table C.1.

Table C.1 Abstract Syntax Tree Notation

Notation	**Meaning**
$\langle ... \rangle$	Tuple
$\{...\}$	Set
[...]	Sequence
$\mid$	Logical OR
Mixed Case	Object
Lower Case	Low-level symbol/number

Generally, dot notation is used to traverse the tree. For example, if $\mathcal{C}$ is a class in a GOMT domain theory, $\mathcal{DT}$, then

$$\mathcal{C} \in \mathcal{DT}.GOMT\text{-}Class$$

where $\mathcal{DT}.GOMT\text{-}Class$ is the set of all classes in the domain theory $\mathcal{DT}$. Likewise, for some connection c, by the GOMT AST definition below, c has four components. Assume c is defined as shown below.

$$\begin{aligned} c.name &= Pump \\ c.Qualifier &= \langle Pump\text{-}Number, integer \rangle \\ c.role &= \text{undefined} \\ c.Mult &= Many \end{aligned}$$

Then the connection c defines a connection to the *Pump* class that has the qualifier *Pump-Number* which is an integer. There is no role name assigned for this particular connection and the multiplicity is defined as *Many* (zero or more).

C.2 Generic OMT Abstract Syntax Tree

```
GOMT-DomainTheory = <{GOMT-Class}, {Assoc}>

GOMT-Class = <name, {Superclass}, [Connection], {Attribute}, {State},
              {Transition}, {Axiom}, {GOMT-Op}, {Functional-Obj}>

Assoc = <name, [Connection], {Attribute}, {GOMT-Op}>

Connection = <name, Qualifier, role, Mult>

Qualifier = <name, datatype>

Mult = One | Many | Plus | Optional | Specified

Plus = integer

Specified = {Spec-Range}

Spec-Range = <value1, value2>

Attribute = DerivedAttr | NormalAttr

DerivedAttr = <name, {Axiom}, datatype>

NormalAttr = <name, {Axiom}, datatype>

State = <name, {Axiom}, {State}>

Transition = <name, [Parameter], Axiom, {Action}, FromState, ToState>

FromState = name

ToState = name

Action = <name, [Parameter], {Action}>

GOMT-Op = <name, [Parameter], Result, Definition>

Result = datatype

Parameter = <name, datatype>

Definition = {Axiom}

Functional-Obj = Process | Datastore | Dataflow

Process = <name, [InFlows], [OutFlows], {Process}>

Datastore = <name, [InFlows], [OutFlows]>
```

```
InFlow = <name, type>

OutFlow = <name, type>

Dataflow = <name, type, source, target>

SuperClass = superclass

SubClass = subclass
```

C.3 O-SLANG *Abstract Syntax Tree*

```
O-Slang-DomainTheory = {Spec}

Spec = Class | AbClass| Event | Aggregate | Link | Association

Class = <name, ClassSort, {SortAxiom}, {Operation}, {Import}, {Sort},
         {Attribute}, {Method}, {StateAttr}, {Event}, {State}, {Axiom},
         contained-in>

AbClass = <name, ClassSort, {SortAxiom}, {Operation}, {Import}, {Sort},
           {Attribute}, {Method}, {StateAttr}, {Event}, {State},
           {Axiom}, contained-in>

Event = <name, ClassSort, {SortAxiom}, {Operation}, {Import}, {Sort},
         {Attribute}, {Method}, {StateAttr}, {Event}, {State}, {Axiom}>

Association = <name, ClassSort, {SortAxiom}, {Operation}, {Import},
               {Sort}, {Attribute}, {Method}, {StateAttr}, {Event},
               {State}, {Axiom}, link-class>

Link = <name, ClassSort, {SortAxiom}, {Operation}, {Import}, {Sort},
        {Attribute}, {Method}, {StateAttr}, {Event}, {State}, {Axiom}>

Aggregate = <name, {Node}, {Arc}>

ClassSort = <class-sort-id, {Inherited-Sort-Id}>

SortAxiom = sort-id

Import = class-ref

Sort = sort-id

Inherited-Sort-Id = sort-id

Operation = <name, [Domain-Ident], [Range-Ident]>

Attribute = <name, [Domain-Ident], [Range-Ident]>
```

```
StateAttr = <name, [Domain-Ident], [Range-Ident]>

Method = <name, [Domain-Ident], [Range-Ident]>

Event = <name, [Domain-Ident], [Range-Ident]>

State = <name, [Domain-Ident], [Range-Ident]>

Node = <name, class-ref>

Arc = <arc-from-node, arc-to-node, {NodeMap}>

NodeMap = <map-from, map-to, FromOp, ToOp>

FromOp = <name, [Domain-Ident], [Range-Ident]>

ToOp = <name, [Domain-Ident], [Range-Ident]>

Domain-Ident = sort-id

Range-Ident = sort-id
```

Appendix D. Demonstration System

D.1 Introduction

This Appendix documents the implementation of a graphical, object-oriented user interface used in the proof-of-concept demonstration of a parallel refinement specification acquisition system. The goal of this demonstration is to show that Rumbaugh's Object Modeling Technique (OMT) diagrams (83) can be automatically transformed into theory-based specifications consistent with the original diagrams, not to demonstrate a complete parallel refinement environment. This Appendix informally presents the method used to obtain a generic OMT abstract syntax tree (AST) representation.

To simplify implementation of the demonstration software, a commercially available object-oriented drawing package, ObjectMaker[1], was used to implement the user interface. A diagram of the demonstration system is shown in Figure D.1. Rumbaugh OMT diagrams are developed in ObjectMaker and exported to external *.TXT* files. A *.TEXT* file allows the user to overcome some shortcomings of ObjectMaker by manually entering data not handled properly by ObjectMaker. These files are converted to a different format, via the program *read.c*, and merged into a single OMT specification. This specification is parsed into a Refine[2] AST using a parser developed in Dialect[3]. Once in Refine, a rule-based conversion program transforms the ObjectMaker OMT AST into the Generic OMT (GOMT) AST as defined in Appendix A.

A brief overview of ObjectMaker is presented in Section D.2 followed by a description of the ObjectMaker specific OMT parser in Section D.3. Finally, the ObjectMaker OMT AST to GOMT AST transformation program is discussed in section D.4.

[1]ObjectMaker is a registered trademark of Mark V Systems Limited Encino California
[2]Refine is a trademark of Reasoning Systems Inc. Palo Alto California
[3]Dialect is a trademark of Reasoning Systems Inc. Palo Alto California

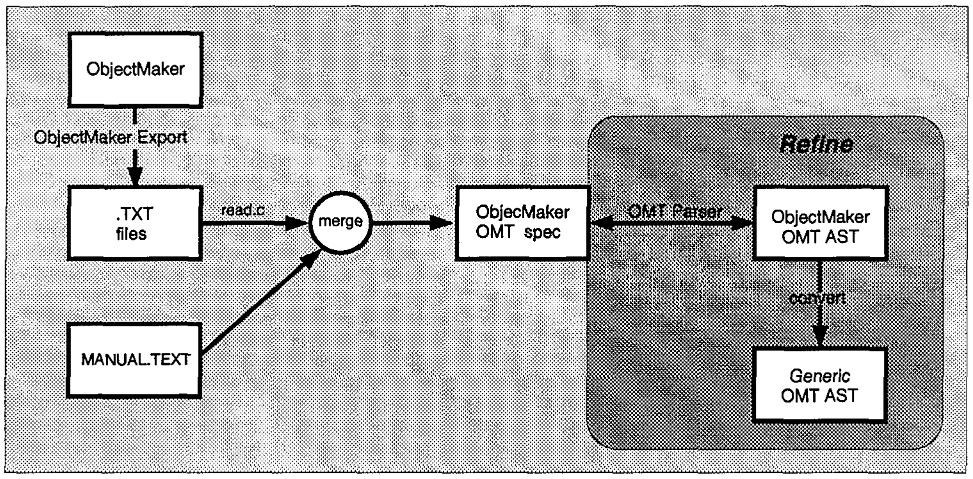

Figure D.1 ObjectMaker to GOMT Transformation System

D.2 ObjectMaker

ObjectMaker is a commercially available object-oriented drawing, code generation, and re-engineering tool. It supports many object-oriented design techniques including Rumbaugh's OMT. An example of an ObjectMaker window is shown in Figure D.2.

In ObjectMaker, OMT diagrams are created and stored in project *repositories.* Actually, diagrams are stored separately; however, the information contained on the diagrams is stored in repositories. Rumbaugh diagrams supported by ObjectMaker include the Dynamic Model, Event Flow diagram, Event Trace diagram, Functional Model, and Object Model. With a few exceptions, ObjectMaker allows the user to draw diagrams as described by Rumbaugh. Specific problems with ObjectMaker are enumerated below.

1. Does not allow aggregate qualifiers.

2. Lacks an adequate device for inserting constraints at the class level.

3. Lacks the ability to define operation semantics.

4. Does not capture substate–superstate relationships.

5. Does not capture subprocess–superprocess relationships.

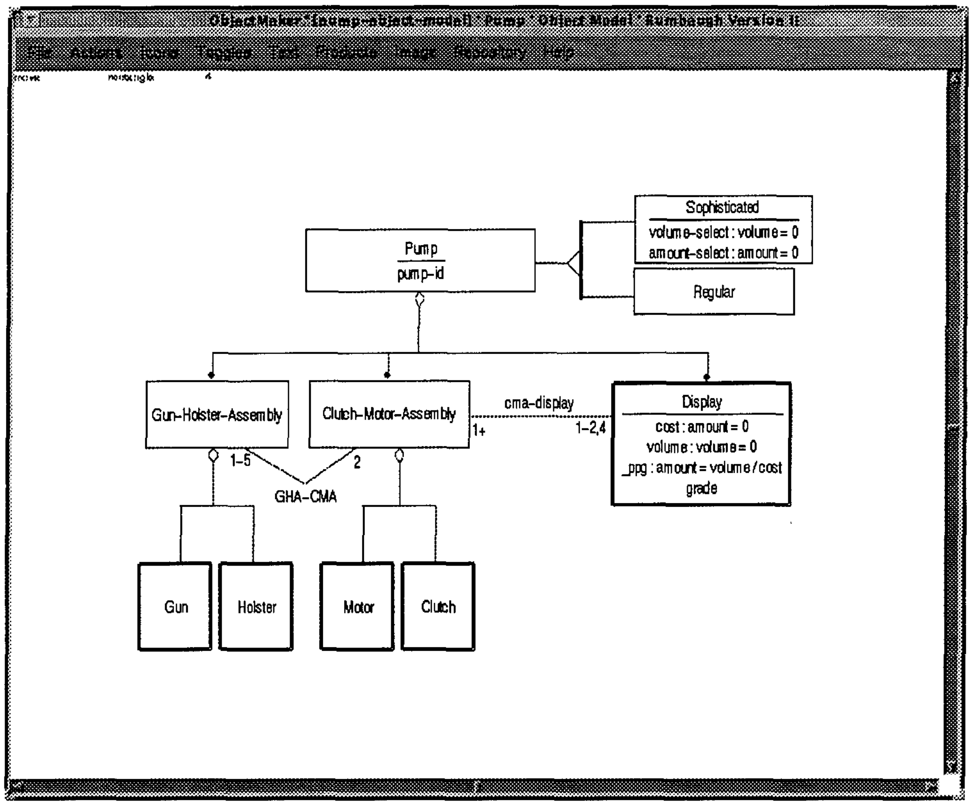

Figure D.2 ObjectMaker Window

6. Does not provide links between classes and their dynamic and functional diagrams.

7. Intermittent problems in mapping diagrams to their repositories and exporting repositories to text files.

The first five problems with ObjectMaker listed above involved the inability of ObjectMaker to capture the entire Rumbaugh model. These problems were alleviated by using manual data entry as described in Section D.2.1. Item six, lack of a link between object classes and their dynamic and functional models, was overcome through a naming convention. Because ObjectMaker does store the diagram on which particular dynamic and functional concepts reside, I required the first word in the diagram name to be the name of the associated object class. This work around solved the problem. The last item listed above, intermittent problems mapping and exporting data, is a nuisance but not a fatal flaw. Inconsistencies between the diagrams and the repository occur quite often, with no messages or explanations from ObjectMaker, requiring the user to completely rebuild the repositories from scratch. Then, once the repository and the diagrams do match, exporting the repositories is often incomplete due to errors; however, ObjectMaker does report these errors to the user. Exiting ObjectMaker and re-exporting generally solves the problem.

D.2.1 Manual Text File. As stated above, to workaround the inability of ObjectMaker to capture certain vital data, a manually created text file, MANUAL.TEXT, is used to augment the .TXT files exported by ObjectMaker. An example MANUAL.TEXT file is shown in Figure D.3. Five types of data may be entered in MANUAL.TEXT: aggregate qualifiers, class constraints, method definitions, substate definitions, and subprocess definitions. Each of these is shown in Figure D.3. The aggregate qualifier, substate definition, and subprocess definition simply state that a relationship exists between a qualifier/substate/subprocess and its associated aggregate/state/process. The class-constraints allows the user to enter general class constraints via first-order axioms. These constraints may be constraints on attribute values or state invariants. Method definitions define the effect of a method on each attribute defined in its class. Once again, these definitions are in the form of a set of first-order axioms. The first line in the definition defines the method's class.

The MANUAL.TEXT file is merged directly into the OMT specification which is parsed into the ObjectMaker OMT AST. ObjectMaker does have the capability to manually enter some the data found in MANUAL.TEXT; however, this manually entered data is lost whenever the repository becomes inconsistent and has to be rebuilt.

D.3 OMT Parser

The ObjectMaker OMT parser is defined in Refine Dialect and is used to parse ObjectMaker OMT specifications into a Refine AST. The ObjectMaker OMT AST mirrors the ObjectMaker OMT specification language and was only intended as a convenient way to get the ObjectMaker OMT specification into a Refine AST where it is more easily manipulated and transformed into the

```
aggregate: Pump has qualifier display-id to component Display.

class-constraints: Display
  display-state(d) = zero-display => cost(d) = 0 & volume(d) = 0;
  display-state(d) = increment-display => cost(d) >= 0 & volume(d) >= 0;
  cost(d) >= 0;
  volume(d) >= 0
end class-constraints.

definition: update-display
  class = display;
  update-display(d) = update-cost(update-volume(d))
end definition.

definition: update-volume
  class = display;
  grade(update-cost(d)) = grade(d);
  volume(update-volume(d)) = volume(d) + 1;
  cost(update-cost(d)) = cost(d)
end definition.

definition: update-cost
  class = display;
  grade(update-cost(d)) = grade(d);
  volume(update-cost(d)) = volume(d);
  cost(update-cost(d)) = cost(d) + 1
end definition.

definition: zero-out-display
  class = display;
  grade(update-cost(d)) = grade(d);
  cost(zero-out-display(d)) = 0;
  volume(zero-out-display(d)) = 0
end definition.

substate Locked < state Display-On .
substate Running < state Display-On .

subprocess update-volume < process update-display.
subprocess update-cost < process update-display.
```

Figure D.3 MANUAL.TEXT Example

GOMT AST. I decided not to attempt to parse the ObjectMaker OMT specification directly into the GOMT AST due to the required complexity of the required parser. No semantic processing is done on the ObjectMaker OMT specification after being parsed into the AST. All semantic processing is done after the conversion to the GOMT AST described in the next section.

D.4 ObjectMaker OMT AST to GOMT AST Transformation

Transformation from the ObjectMaker OMT AST and the GOMT AST is accomplished through a rule-based Refine program. This transformation program takes individual objects from the ObjectMaker OMT AST and transforms them into the appropriate GOMT AST object. There are four phases to the transformation process: (I) creation of class and association objects, (II) creation of dynamic and functional objects for each class, (III) filling in attributes of objects defined in phases two and three, and (IV) checking redundant information for consistency.

Because ObjectMaker exports its repositories in a series of flat files with redundant information, the ObjectMaker OMT AST reflects that architecture; therefore, the transformation process must be completed in steps. Since the basic objects in the GOMT AST are classes and associations, these objects must be created before ObjectMaker OMT AST objects that are logically a part of a class or association are transformed. Therefore, in the first transformation phase, only the root, class, and association objects are allowed to be transformed. During the preorder traversal of the ObjectMaker OMT AST, each object and association object encountered causes the appropriate class or association object to be created in the GOMT AST. Any attributes, partitions, operations, constraints, or superclasses defined in the class object get transformed into the appropriate attribute in the new class object. Likewise, any association classes, attributes, or operations are also transformed into the new AST.

Once all the basic classes and associations have been created in the GOMT AST, the dynamic model objects (state and transitions) and functional model objects (actors, processes, dataflows, and datastores) are created and attached to their appropriate classes defined in phase I. It is in this phase that the transformation system uses the diagram names to determine which class owns the dynamic and functional model components. If the class specified by the diagram name has not been created, error messages are generated.

After all the functional and dynamic model components have been transformed and attached to their owning classes, the attributes specified in various ObjectMaker OMT AST objects are used to fill in the existing class, association, functional, and dynamic objects. Information transformed in this step includes association and aggregate role names, and qualifiers; operation parameters, results, and axioms; substate and subprocess relationships; attribute datatype and initial value or derivation axioms; and inheritance information. During phase III all duplicate data in the objects being transformed is checked against the existing objects to ensure consistency.

Once phase III is completed, the entire GOMT AST is complete. The only phase left is consistency checking. There is only one ObjectMaker OMT AST object that is completely redundant: the generalization object. This object is used to ensure that all superclass – superclass relationships have been captured.

Once phase IV is completed, the GOMT AST is complete. There is no semantic processing for this tree. All semantic processing is done during or after transformation to the O-SLANG AST as discussed in Chapter VII.

D.5 GOMT AST to O-SLANG *Transformation*

Once in the GOMT AST, a rule-based transformation program implementing the transformation rules defined in Chapter VII transforms the GOMT AST into an O-SLANG AST within the Refine environment. This transformation process is much like the process for transforming the ObjectMaker AST into the GOMT AST except it carries out the transformation in one phase. Once in a valid O-SLANG AST, the Dialect pretty printer is used to produce a textual representation of the O-SLANG AST.

The actual transformation is performed by creating the root node of the O-SLANG AST and then automatically transforming each class and association, one at a time, within the GOMT AST. The only sequencing done in the transformation is done to ensure that all component classes of an aggregate are transformed before the aggregate itself is transformed.

In its current state, the GOMT to O-SLANG transformation system converts almost all of the GOMT AST objects correctly into O-SLANG with the following exceptions.

1. Multiple Event Theories

2. Super/Substate Axiom Generation
3. Multiple Parameter Events
4. Association and Aggregate Qualifiers

These items were not implemented because I did not consider the time required to implement and debug them useful. None of these items were omitted due to the inability to implement them.

D.6 Summary

This appendix documents the use of the object-oriented drawing tool ObjectMaker as a front end for the user interface of a parallel refinement specification acquisition system. Data is exported from ObjectMaker, merged with additional manually entered data, and parsed into an AST based on the ObjectMaker output files. This AST is then transformed via a rule-based Refine program into the GOMT AST which is used as the starting point for a formal transformation from OMT to theory-based specifications. A conversion system for the GOMT to O-SLANG transformation, based on the rules defined in Chapter VII, was developed and used to produce the demonstration examples shown in Chapter VIII.

Specification of TRIV

spec TRIV **is**
sorts E
end-spec

Specification of SET

```
spec SET is
import INTEGER
sorts E, Set
constants
  empty-set: Set
operations
  in : E, Set → boolean
  empty? : Set → boolean
  insert : E, Set → Set
  singleton : E → Set
  union : Set, Set → Set
  delete : E, Set → Set
  size : Set → integer
  subset : Set, Set → boolean
constructors {insert, empty-set} construct Set
constructors {union, singleton, empty-set} construct Set
axioms
  union(x,y) = union(y,x);
  union(x,union(y,z)) = union(union(x,y) z);
  union(x,empty-set) = x;
  union(empty-set x) = x;
  union(x,x) = x;
definition
  in(x,insert(y,c)) ⇔ x = y ∨ in(x,c);
  in(x,empty-set) = false;
  end-definition
definition
  theorem
    empty?(insert(x,c)) = false;
    empty?(empty-set) = true;
  end-definition
definition
  in(x,union(u,v)) ⇔ in(x,u) ∨ in(x,v));
  in(x,singleton(x)) = true;
  in(x,empty-set) = false;
  end-definition
definition
  empty?(union(u,v)) ⇔ empty?(u) ∧ empty?(v);
  empty?(singleton(x)) = false;
  empty?(empty-set) = true;
  end-definition
```

```
definition of delete is
  delete(x,empty-set) = empty-set;
  delete(x,insert(x,s)) = s;
  x1 ≠ x2 ⇒ delete(x1,insert(x2,s)) = insert(x2,delete(x1,s));
  end-definition
definition set-equal-def of equal is
  s = t ⇔ (∀ (x:E) in(x,s) ⇔ in(x,t));
  end-definition
definition set-size-def of size is
  size(empty-set) = 0;
  in(e,s) ⇒ size(insert(e,s)) = size(s);
  ¬ in(e,s) ⇒ size(insert(e,s)) = size(s) + 1;
  in(e,s) ⇒ size(delete(e,s)) = size(s) − 1;
  ¬ in(e,s) ⇒ size(delete(e,s)) = size(s);
  end-definition
definition subset-def of subset is
  subset(s1,s2) = in(e,s1) ⇒ in(e,s2);
  end-definition
end-spec
```

Specification of INTEGER

```
spec INTEGER is
sorts Integer
operations
   zero : → Integer
   one : → Integer
   < : Integer, Integer → Boolean
   > : Integer, Integer → Boolean
   ≤ : Integer, Integer → Boolean
   ≥ : Integer, Integer → Boolean
   + : Integer, Integer → Integer
   − : Integer, Integer → Integer
   × : Integer, Integer → Integer
   min : Integer, Integer → Integer
   max : Integer, Integer → Integer
axioms
   (zero < one);
   (x ≤ x);
   (x ≤ y) ∨ (y ≤ x);
   (x ≤ y) ∧ (y ≤ z)) ⇒ (x ≤ z);
   (x ≤ y) ∧ (y ≤ x)) ⇒ (x = y);
   (x > y) ⇔ ((y + one) ≤ x);
   (x < y) ⇔ ((x + one) y);
   (¬ ((y + one) ≤ x)) ⇔ (x ≤ y);
   (x ≥ y) ⇔ (y ≤ x);
   (x + y) = (y + x);
   (x + (y + z)) = ((x + y) + z);
   ((x + y) + z) = (x + (y + z));
   (x + zero) x);
   (x − x) zero);
   ((z − x) − y) = (z − (x + y));
   ((x + y) = (x + z)) ⇔ (y = z);
   (x ≤ (y − z)) ⇔ ((x + z) ≤ y);
   ((x − y) ≤ z) ⇔ (x ≤ (y + z));
   ((y + x) = (z + x)) ⇒ (y = z);
   ((zero ≤ x) ∧ (zero ≤ y)) ⇒ (zero ≤ (x + y));
   (y zero) ⇒ ((x × y) = zero);
   (y = one) ⇒ ((x × y) = x);
   ((¬ (y = zero)) ∧ (¬ (y = one))) ⇒ ((x × y) = (((x + (y − one)) × x));
   (x ≤ y) ⇒ (min(x,y) = x);
   (x > y) ⇒ (min(x,y) = y);
   (x ≥ y) ⇒ (max(x,y) = y);
   (x > y) ⇒ (max(x,y) = x)
end-spec
```

Appendix F. Translation Correctness

In this Appendix, I prove Theorems VII.1, VII.2, and VII.3. These theorems show that the transformation rules as defined in Chapter VII preserve the semantics of the the object model, the dynamic model, and the functional model as defined Chapter V.

F.1 Object Model Correctness Proof

In this section, Theorem VII.1 is proved.

Proof. Preservation of the object model semantics by τ is established by showing the equivalence of two sets of object model semantics, OM and OM', created from a generic OMT domain theory, $\mathcal{G}$. OM is the object model semantics defined by transforming $\mathcal{G}$ by φ while OM' is the object model semantics defined by transforming $\mathcal{G}$ by τ, into an O-SLANG domain theory $\mathbb{O}$, and then by ω. In this proof, I assume that $\mathcal{G}$ has a well defined object model in which $\mathcal{C}$ is a class and $\mathcal{A}$ is an association.

I prove the theorem by showing that, given a valid generic OMT domain theory $\mathcal{G}$, each component (Name, Imports, Sorts, Operations, and Axioms) of each specification in OM exists in OM' and that each component of each specification in OM' exists in OM. I start by proving that the set of specifications in OM and OM' are equivalent and then complete the proof by showing that each component within those specifications are equivalent.

1. In this section I show that the set of specifications in OM and OM' are equivalent by showing for any specification in OM, there is a corresponding specification on OM' and vice versa.

   $\underline{S \in OM \Rightarrow S' \in OM'}$. If S_C is some specification in OM, it must have been generated by Equation 7.40 (S is a class specification S_C) or 7.49 (S is an association specification S_A). If S is a class then by Equation 7.40, there must exist a class $\mathcal{C}$ in $\mathcal{G}$ such that $\mathcal{C}.Name =$ $S_C.Name$. Then by OMT-1 or OMT-2 there exists a class $\mathbb{C}$ in $\mathbb{O}$ such that $\mathbb{C}.Name =$

$\mathcal{C}.Name = S_C.Name$ that in turn, by Equation 7.58, generates a class S'_C in OM' such that $S'_C.Name = \mathbb{C}.Name = S_C.Name.$

If S is an association then by Equation 7.49 there must exist an association $\mathcal{A}$ in $\mathcal{G}$ such that $\mathcal{A}.Name = S_A.Name.$ Then by OMT-44 there exists an association $\mathbb{A}$ in $\mathbb{O}$ such that $\mathbb{A}.Name = \mathcal{A}.Name = S_A.Name$ that in turn, by Equation 7.68, generates an association S'_A in OM' such that $S'_A.Name = \mathbb{C}.Name = S_A.Name.$

$\underline{S' \in OM' \Rightarrow S \in OM}.$ If S' is some specification in OM', it must have been generated by Equation 7.58 (S' is a class specification S'_C) or 7.68 (S' is an association specification S'_A). If S' is a class then by Equation 7.58 there must exist a class $\mathbb{C}$ in $\mathbb{O}$ such that $\mathbb{C}.Name = S'_C.Name.$ Then, since OMT-1 or OMT-2 are the only rules in τ that create simple classes in $\mathbb{O}$, there exists a class $\mathcal{C}$ in $\mathcal{G}$ such that $\mathcal{C}.Name = \mathbb{C}.Name = S'_C.Name$ that in turn, by Equation 7.40, generates a class S_C in OM such that $S_C.Name = \mathcal{C}.Name = S'_C.Name.$

If S' is an association then by Equation 7.68 there must exist an association $\mathbb{A}$ in $\mathbb{O}$ such that $\mathbb{A}.Name = S'_A.Name.$ Then, since OMT-44 is the only rule that creates associations in $\mathbb{O}$, there exists an association $\mathcal{A}$ in $\mathcal{G}$ such that $\mathcal{A}.Name = \mathbb{A}.Name = S'_A.Name$ that in turn, by Equation 7.49, generates an association S_A in OM such that $S_A.Name = \mathcal{C}.Name = S'_A.Name.$

Therefore, since $S \in OM \Rightarrow S' \in OM'$, $OM \subseteq OM'$ and since $S' \in OM' \Rightarrow S \in OM.$, $OM' \subseteq OM$, therefore, the set of specifications in OM and OM' are equivalent. Now I show that each item in associated specifications in OM and OM' are equivalent.

2. Imports

$\underline{i \in S.Imports \Rightarrow i' \in S'.Imports.}$ Within each specification $S \in OM$, there is a set of imported specifications whose names are in $S.Imports$. In the definition of φ, there are three equations that map names into *S.Imports*: Equations 7.41, 7.48, and 7.50. Each of these possibilities is analyzed below.

(a) If a specification name $s \in$ *S.Imports* is generated via Equation 7.41, then i represents the datatype of an attribute and thus there exists an attribute $a = \langle$*a.name*,, *axiom*, $i\rangle \in C$ such that i is the datatype of a. If a is a normal attribute, OMT-13 generates an attribute $\langle$*a.Name*, [$\mathbb{C}$.*Name*], $[i]\rangle \in \mathbb{C}$.*Attribute* and by Equation 7.59 in ω, *Range-Ident*(1) $= i \in S_C$.*Imports*. If a is a derived attribute, OMT-14 generates an operation $\langle$*a.Name*, [$\mathbb{C}$.*Name*], $[i]\rangle \in \mathbb{C}$.*Operation* and by Equation 7.60 in ω, *Range-Ident*(1) $=$ $i \in S'_C$.*Imports*.

(b) If $i \in S_C$.*Imports* is generated by Equation 7.48, then by Equation 7.48 there exists a $c \in C$.*Superclass* such that $c = i$. Then, by OMT-4, $i \in C$.*Superclass* generates an i in $\mathbb{C}$.*class-sort.inherited-class-id* that in turn, by Equation 7.67, inserts i into S'_C.*Imports*.

(c) If $i \in S_A$.*Imports* is generated by Equation 7.50, then i must be the name of a connection in $\mathcal{A}$. Then by OMT-46, if i is the name of some c in $\mathcal{A}$.*Connection* there exists an operation $o = \langle$*IMAGE*, [$\mathbb{A}$.*Name*, *c.Name*], $[x]\rangle \in \mathbb{A}$.*Operation* that in turn, by Equation 7.69, places *c.Range-Ident*(2) $= i$ in S'_A.*Imports*. Therefore, if $i \in$ *S.Imports*, $i \in S'$.*Imports*.

Therefore, since for each possible source of i in S (Equations 7.41, 7.48, and 7.50), if i was generated by that equation then $i \in$ *S.Imports* $\Rightarrow i \in S'$.*Imports* and *S.Imports* $\subseteq$ S'.*Imports*.

$\underline{i' \in S'.Imports \Rightarrow i \in S.Imports.}$ If $i' \in S'$.*Imports*, there are four possible sources – Equations 7.59, 7.60, 7.67, or 7.69. Each is analyzed below.

(a) If the source of $i \in S'.Imports$ is Equation 7.59, then there must be some attribute $a \in \mathbb{C}.Attribute$ such that $a.Range\text{-}Ident = [i]$ and i is not a class set or association sort. The only transformation capable of creating such an attribute in $\mathbb{C}$ from $\mathcal{G}$ is OMT-13. Thus by OMT-13 there must be some a in $\mathcal{C}.Attribute$ such that $i = type(a)$ and by the assumption that all attributes have defined datatypes $i = a.datatype$. Therefore, by Equation 7.41 in transformation φ, $a.datatype \in S.Imports$ and thus $i \in S.Imports$.

(b) If the source of $i \in S'.Imports$ is Equation 7.60, then there must be some operation $o \in \mathbb{C}.Operation$ such that $i \in o.Range\text{-}Ident$. The only transformation that creates such an operation in $\mathbb{C}$ from $\mathcal{G}$ is OMT-14. (Actually, OMT-15 and OMT-90 could also create operations in $\mathbb{C}$; however, OMT-15 creates the *attr-equal* operation and OMT-90 creates operations manually defined by the user that, by assumption, do not exist.) Thus by OMT-14 there must be some a in $\mathcal{C}.Attribute$ such that $i = = a.datatype$ and by Equation 7.41 in φ, $a.datatype \in S.Imports$ and thus $i \in S.Imports$.

(c) If the source of $i \in S'.Imports$ is Equation 7.67 then i must be a sort in $\mathbb{C}.Class\text{-}Sort.Inherited\text{-}Sort\text{-}Id$. Since OMT-4 is the only transformation that generates sorts in $\mathbb{C}.Class\text{-}Sort.Inherited\text{-}Sort\text{-}Id$, it must be the case that i is in $\mathcal{C}.Superclass$ and thus, by φ Equation 7.48, $i \in S.Imports$.

(d) Finally, if the source of $i \in S'.Imports$ is Equation 7.69 then i must be $o.Domain\text{-}Ident(2)$ in some operation named *IMAGE* in $\mathbb{A}.Operation$. Since OMT-46 is the only transformation in τ capable of placing an *IMAGE* operation in $\mathbb{A}.Operation$ and (OMT-49 defines *qualified* operations that, by assumption, are not included in $\mathcal{G}$), i must be the name of some connection, c, in $\mathcal{A}$ and thus $i = c.Name$. Then by Equation 7.50 $s.Name \in S.Imports$ and thus $i \in S.Imports$.

Therefore, since for each possible source of i in S' (Equations 7.59, 7.60, 7.67, and 7.69), if i was generated by that equation then $i \in S'.Imports \Rightarrow i \in S.Imports$ and $S'.Imports \subseteq$

$S.Imports$. Also, since I already established that $S.Imports \subseteq S'.Imports$, it must be the case that $S.Imports = S'.Imports$.

3. Sorts

   $\underline{s \in S.Sorts \Rightarrow s' \in S'.Sorts}$. Within each specification $S \in OM$, there is a set of sorts defined by placing their names in $S.Sorts$. In the definition of φ, there are two equations that map names from $\mathcal{G}$ into $S.Sorts$: Equations 7.40 and 7.49. Since, as shown in Item 1 above using Equations 7.40 and 7.49, any $S \in OM$ with name $n \Rightarrow \exists\ S' \in OM'$ with name n and by Equations 7.58 and 7.68 if $n = S'.Name$ then $n \in S'.Sort$. Therefore $S.Sorts \subseteq S'.Sorts$.

   $\underline{s \in S'.Sorts \Rightarrow s \in S.Sorts}$. In the definition of ω, there are two equations that map from $\mathbb{C}$ into $S'.Sorts$: Equations 7.58 and 7.68. Since, as shown in Item 1 above using Equations 7.58 and 7.68, if $S' \in OM'$ and $S'.name = n$ then there exists an $S \in OM$ with $S.name = n$ as well. And since S was created by either Equation 7.58 or 7.68, this implies $n \in S.Sort$. Therefore $S'.Sorts \subseteq S.Sorts$ and since $S.Sorts \subseteq S'.Sorts$, it must be true that $S.Sorts = S'.Sorts$.

4. Operations

   $\underline{o \in S.Operations \Rightarrow o' \in S'.Operations}$. If $o \in S.Operations$, there are four possible sources of o as transformed from $\mathcal{G}$: Equations 7.41, 7.42, 7.48, and 7.50. Each of these possibilities is analyzed below.

   (a) If the source of $o \in S.Operations$ is Equation 7.41 then $o = \langle a.Name, [\mathcal{C}.Name], [a.Datatype]\rangle$ for some a in $\mathcal{C}.Attribute$. If this a is a normal attribute, then by OMT-13 $\langle a.Name, [\mathcal{C}.Name], [a.Datatype]\rangle \in \mathbb{C}.Attribute$ and by Equation 7.59, $\langle a.Name, [\mathbb{C}.Name], [a.Datatype]\rangle = \langle a.Name, [\mathcal{C}.Name], [a.Datatype]\rangle \in S'.Operations$.

If, however, a is a derived attribute, then by OMT-14 $\langle a.Name$, $[\mathcal{C}.Name]$, $[a.Datatype]\rangle$ $\in \mathbb{C}.Operation$ and by Equation 7.60, $\langle a.Name$, $[\mathbb{C}.Name]$, $[a.Datatype]\rangle = \langle a.Name$, $[\mathcal{C}.Name]$, $[a.Datatype]\rangle \in S'.Operations.$

(b) If the source of $o \in S.Operations$ is Equation 7.42 then $o = \langle HAS\text{-}PART$, $[\mathcal{C}.Name$, $c.Name]$, $[Boolean]\rangle$ or $o = \langle c.Role$, $[\mathcal{C}.Name$, $c.Name]$, $[Boolean]\rangle$ for some $c \in \mathcal{C}.Connection$ based on whether $c.Role$ is defined.

  i. If $c.Role$ is defined, then by OMT-5, $\langle c.Role$, $[\mathcal{C}.Name]$, $[c.Name\text{-}CLASS]\rangle \in \mathbb{C}.Attribute$ and by ω Equation 7.61, $\langle c.Role$, $[\mathbb{C}.Name, c.Name]$, $[Boolean]\rangle =$ $\langle c.Role$, $[\mathcal{C}.Name, c.Name]$, $[Boolean]\rangle = o \in S'.Operations.$

  ii. If $c.Role$ is not defined, then by OMT-5, $\langle c.Name\text{-}OBJ$, $[\mathcal{C}.Name]$, $[c.Name\text{-}CLASS]\rangle \in \mathbb{C}.Attribute$ and by ω Equation 7.61, $\langle HAS\text{-}PART$, $[\mathbb{C}.Name$, $c.Name]$, $[Boolean]\rangle = \langle HAS\text{-}PART$, $[\mathcal{C}.Name$, $c.Name]$, $[Boolean]\rangle = o \in S'.Operations.$

(c) If the source of $o \in S.Operations$ is Equation 7.48 then $o = \langle SIMULATES$, $[\mathcal{C}.Name]$, $[c]\rangle$ for some $c \in \mathcal{C}.Superclass$. Then by OMT-4, $c \in \mathcal{C}.Superclass$ implies $c \in \mathbb{C}.Class\text{-}Sort.Inherited\text{-}Sort\text{-}Id$ and by Equation 7.67 in ω, $\langle SIMULATES$, $[\mathbb{C}.Name]$, $[c]\rangle =$ $\langle SIMULATES$, $[\mathcal{C}.Name]$, $[c]\rangle = o \in S'.Operations.$

(d) If the source of $o \in S.Operations$ is Equation 7.50 then $o = \langle \mathcal{A}.Name$, dom, $[Boolean]\rangle$ where $dom = [c.Name \mid c \in \mathcal{A}.Connection]$. By OMT-46, for each $c \in \mathcal{A}.Connection$, there must be an operation $\langle IMAGE$, $[\mathcal{A}.Name$, $c.Name]$, $[c_2.Name\text{-}CLASS]\rangle \in \mathbb{A}.Operation$. Then, by ω Equation 7.69, the second sort in each $IMAGE$ operation in $\mathbb{A}$, $Range\text{-}Ident(2)$ (which equals $c.Name$ for some $c \in \mathcal{C}.Connection$) becomes part of the domain, dom, of the operation $\langle \mathbb{A}.Name$, dom, $[Boolean]\rangle = \langle \mathcal{A}.Name$, dom, $[Boolean]\rangle = o \in S'.Operations.$

Therefore, since for each possible source of o in S (Equations 7.41, 7.42, 7.48, and 7.50), if o was generated by that equation then $o \in S.Operations \Rightarrow o' \in S'.Operations$ and $S.Operations \subseteq S'.Operations$.

$\underline{o \in S'.Operations \Rightarrow o \in S.Operations.}$ If $o \in S'.Operations$, there are five possible sources – Equations 7.59, 7.60, 7.61, 7.67, or 7.69. Each is analyzed below.

(a) If the source of $o \in S'.Operations$ is Equation 7.59 then there must be some attribute $a \in \mathbb{C}.Attribute$ such that $a = o = \langle o.Name, \mathbb{C}.Name, o.Range\text{-}Ident \rangle$. The only τ transformation that could create such an attribute in $\mathbb{C}$ is OMT-13. Thus by OMT-13 there must be some a' in $\mathcal{C}.Attribute$ such that $a'.Name = o.Name$ and $[a'.Datatype] = o.Range\text{-}Ident$. Therefore, by Equation 7.41 in transformation φ, $\langle a'.Name, [\mathcal{C}.Name], [a'.Datatype] \rangle = \langle o.Name, [\mathbb{C}.Name], o.Range\text{-}Ident \rangle = o \in S.Operations$.

(b) If the source of $o \in S'.Operations$ is Equation 7.60 then $o = \langle o.Name, [\mathbb{C}.Name]$ $o.Range\text{-}Ident \rangle \in \mathbb{C}.Operation$ and $o.Name \neq ATTR\text{-}EQUAL$. The only transformation that creates such an operation in $\mathbb{C}$ from $\mathcal{G}$ is OMT-14. Thus by OMT-14 there must be some $a = \langle a.Name, [\mathcal{C}.Name], [a.Datatype] \rangle$ in $\mathcal{C}.Attribute$ such that $a.Name = o.Name$ and $a.Datatype = o.Range\text{-}Ident$. Then, by Equation 7.41 in φ, $\langle a.Name, [\mathcal{C}.Name], [a.Datatype] \rangle = \langle o.Name, [\mathbb{C}.Name], o.Range\text{-}Ident \rangle = o \in S.Operations$.

(c) If the source of $o \in S'.Operations$ is Equation 7.61 then $o = \langle o.Name, [\mathbb{C}.Name, name]$ $[Boolean] \rangle \in S'_C.Operations$ and, since Equation 7.61 is the only equation in ω capable of producing such an operation, there exists an attribute $a \in \mathbb{C}.Attribute$ such that $a.Range\text{-}Ident = [name\text{-}CLASS]$ and, by the definition of *om-pred*, $a.Name = xxx\text{-}OBJ$ if $o.Name = HAS\text{-}PART$ or $a.Name = o.Name$ if $o.Name \neq HAS\text{-}PART$.

Since the only transformation that can create an attribute with this signature in $\mathbb{C}$ from $\mathcal{G}$ is OMT-5, there must exist some $c \in \mathcal{C}.Connection$ such that $c.Name = name$ and *attr-name*(c) = *a.Name*. This means that if *o.Name* = *HAS-PART* then *c.Role* is undefined, otherwise *c.Role* = *a.Name* = *o.Name*.

Thus by Equation 7.42 in φ,

⟨*comp-pred*(c), [$\mathcal{C}$*.Name*, *c.Name*], [*Boolean*]⟩ = ⟨*o.Name*, [*S.Name*, *name*], [*Boolean*]⟩ = ⟨*o.Name*, [*S′.Name*, *name*], [*Boolean*]⟩ = $o \in S_C$*.Operations*. Therefore, $o \in$ *S′.Operations* $\Rightarrow o \in$ *S.Operations* for operations in S' generated by Equation 7.61.

(d) If the source of $o \in$ *S′.Operations* is Equation 7.67 then o = ⟨*SIMULATES*, [$\mathbb{C}$*.Name*], [c]⟩ and c must be a sort in $\mathbb{C}$*.Class-Sort.Inherited-Sort-Id*. Since OMT-4 is the only transformation that generates sorts in $\mathbb{C}$*.Class-Sort.Inherited-Sort-Id*, it must be the case that c is in $\mathcal{C}$*.Superclass* and thus, by φ Equation 7.48, ⟨*SIMULATES*, [$\mathcal{C}$*.Name*], [c]⟩ = ⟨*SIMULATES*, [$\mathbb{C}$*.Name*], [c]⟩ = $o \in$ *S.Operations*.

(e) Finally, if the source of $o \in$ *S′.Operations* is Equation 7.69 then o = ⟨[$\mathbb{A}$*.Name*], *dom*, [*Boolean*]⟩ where each sort name, $s \in$ *dom*, must be *o.Domain-Ident*(2) in some operation named *IMAGE* in $\mathbb{A}$*.Operation*. Since OMT-46 is the only transformation in τ capable of placing an *IMAGE* operation in $\mathbb{A}$*.Operation* and (OMT-49 defines *qualified* operations that, by assumption, are not included in $\mathcal{G}$), s must be the name of some connection, c, in $\mathcal{A}$ and thus s = *c.Name*. Then by Equation 7.50, ⟨[$\mathcal{A}$*.Name*], *dom*, [*Boolean*]⟩ = ⟨[$\mathbb{A}$*.Name*], *dom*, [*Boolean*]⟩ = $o \in$ *S.Operations*.

Therefore, since for each possible source of o in S' (Equations 7.59, 7.60, 7.61, 7.67, and 7.50), if o was generated by that equation then $o \in$ *S′.Operations* $\Rightarrow o \in$ *S.Operations* and *S′.Operations* $\subseteq$ *S.Operations*. Also, since I already established that *S.Operations* $\subseteq$ *S′.Operations*, it must be the case that *S.Operations* = *S′.Operations*.

5. Axioms

Initially, the number of transformations equations in φ (7.43, 7.44, 7.45, 7.46, 7.47, 7.51, 7.52, 7.53, 7.54, and 7.55) along with the number of transformations equations in ω (7.62, 7.63, 7.64, 7.65, 7.66, 7.70, 7.71, 7.72, 7.73, and 7.74) make it appear though proving that the axioms generated by one equation are unique and can only be transformed into an exact duplicate might be exhausting if not impossible; however, by making a few observations the proof can be dramatically simplified.

First, note that the axioms generated by classes by equations 7.43– 7.47 and 7.62 – 7.66 are unique from those generated for associations by equations 7.51 – 7.55 and 7.70 – 7.74. All class axioms are based on a predicate in the class whose name is determined by the functions *comp-pred* or *om-pred.* The resulting predicate name is either *HAS-PART* or the *role* name of the component class. However, in an association, the name of the predicate is the name of the association. Therefore, assuming unique names in the object model, φ and ω generate unique axioms in S_C and S_A.

Second, note that inside a given class or association, the axioms generated by separate rules are unique with a few minor, non-critical exceptions. For example, in the set of equations 7.43– 7.47, Equation 7.43 generates the axiom

$$\text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = 1\text{”}$$

that may also be generated by Equation 7.47. This is because the same OMT notation can be represented in two ways in the generic OMT abstract syntax tree. This is not a problem since both representations result in the same axioms being generated. Also, the axiom generated by Equation 7.44 may also be generated by Equation 7.45 while the axiom generated by Equation 7.46 may also be generated by Equation 7.47. In both these cases, there are two distinct methods of representing the same semantics in the OMT object model; however, both representations generate the exact same axiom. Since I am only concerned about preserving the semantics of the object model, the internal representation in the generic OMT AST is of no importance as long as the semantics are equivalent.

I first look at an axiom a in a class S_C followed by axioms in an association S_A.

Aggregate Axioms.

$\underline{a \in S.Axioms \Rightarrow a' \in S'.Axioms.}$ Since a is in a class, it must have been generated by one of the Equations 7.43 – 7.47. Each possibility is discussed below.

(a) If a in $S_C.Axioms$ is of the form generated by Equation 7.43 such that

$$a = \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = 1\text{”}$$

then there exists a $c \in \mathcal{C}.Connection$ such that (1) by Equation 7.43, $c.Mult = One$, $c.Name = s$, and if $p = HAS\text{-}PART$ then $c.Role = undefined$ else $c.Role = p$, or (2) by Equation 7.47 $c.Mult = Specified$, $c.Name = s$, if $p = HAS\text{-}PART$ then $c.Role = undefined$ else $c.Role = p$, and $c.Specified = \{\langle 1, undefined\rangle\}$.

i. If $c.Mult = One$, then by OMT-8 there exists an axiom $ax_1 \in \mathbb{C}.Axiom$ such that

$$ax_1 = \text{“}SIZE(IMAGE(attr\text{-}name(c)(X), Q)) = 1\text{”}$$

where $attr\text{-}name(c) = c.Role$ (if defined) or $c.Name\text{-}CLASS$.

ii. If $c.Mult = Specified$, then by OMT-12 there exists an axiom $ax_2 \in \mathbb{C}.Axiom$ such that

$$ax_2 = \text{“}SIZE(IMAGE(attr\text{-}name(c)(X), Q)) = s.value1\text{”}$$

Thus it is obvious that $ax_1 = ax_2 = ax \in \mathbb{C}.Axiom$. From here, there are two paths to S'_C, via Equation 7.62 or 7.66.

i. Given ax, by Equation 7.62 there exists an axiom $a'_1 \in S'_C.Axioms$ such that

$$\begin{aligned} a'_1 = \text{“}&X \in sort\text{-}of(attr\text{-}name(c)) \\ &\Rightarrow SIZE(\{Y \mid om\text{-}pred(attr\text{-}name(c))(X,Y)\}) = 1\text{”}\end{aligned}$$

where if $c.Role$ was defined then $sort\text{-}of(attr\text{-}name(c)) = sort\text{-}of(c.Role) = c.Name = s$ (by OMT-5) and $om\text{-}pred(attr\text{-}name(c)) = om\text{-}pred(c.Role) = c.Role = p$, or if $c.Role$ was not defined then $sort\text{-}of(attr\text{-}name(c)) = sort\text{-}of(c.Name\text{-}OBJ) =$

$c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Name\text{-}OBJ) =$ $HAS\text{-}PART = p$. Thus $a = a'_1$.

ii. Given ax, by Equation 7.66 there exists an axioms $a'_2 \in S'_C.Axioms$ such that

$$a'_2 = \text{"}X \in \textit{sort-of}(\textit{attr-name}(c)) \\ \Rightarrow SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) = s.value1\text{"}$$

where $s.value1 = 1$, and if $c.Role$ was defined, $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Role) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Role) = c.Role = p$, or if $c.Role$ was not defined then $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Name\text{-}OBJ) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Name\text{-}OBJ) = HAS\text{-}PART = p$. Thus $a = a'_2$.

Again it is clear that $a'_1 = a'_2 = a$ and thus any axiom in S_C generated by Equation 7.43 (or by Equation 7.47 in the form of Equation 7.43) also exists in S'_C.

(b) If a in $S_C.Axioms$ is of the form generated by Equation 7.44 such that

$$a = \text{"}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) \geq 0\text{"}$$

then there exists a $c \in \mathcal{C}.Connection$ such that (1) by Equation 7.44, $c.Mult = Many$, $c.Name = s$, and if $p = HAS\text{-}PART$ then $c.Role = undefined$ else $c.Role = p$, or (2) by Equation 7.45, $c.Mult = Plus$, $c.Name = s$, if $p = HAS\text{-}PART$ then $c.Role = undefined$ else $c.Role = p$, and $c.Plus.Integer = 0$.

i. If $c.Mult = Many$, then by OMT-9 there exists an axiom $ax_1 \in \mathbb{C}.Axiom$ such that

$$ax_1 = \text{"}SIZE(IMAGE(\textit{attr-name}(c)(X), Q)) \geq 0\text{"}$$

where $\textit{attr-name}(c) = c.Role$ (if defined) or $c.Name\text{-}CLASS$.

ii. If $c.Mult = Plus$, then by OMT-10 there exists an axiom $ax_2 \in \mathbb{C}.Axiom$ such that

$$ax_2 = \text{"}SIZE(IMAGE(\textit{attr-name}(c)(X), Q)) \geq c.Plus.Integer\text{"}$$

Thus it is obvious that $ax_1 = ax_2 = ax \in \mathbb{C}.Axiom$. Then by Equation 7.63 there exists an axiom $a'_1 \in S'_C.Axioms$ such that

$$a_1' = \text{“}X \in \textit{sort-of}(\textit{attr-name}(c)) \\ \Rightarrow SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) \geq 0\text{”}$$

where if $c.Role$ was defined then $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Role) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Role) = c.Role = p$, or if $c.Role$ was not defined then $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Name\text{-}OBJ) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Name\text{-}OBJ) = HAS\text{-}PART = p$. Thus $a = a_1'$.

Again it is clear that $a_1' = a_2' = a$ and thus any axiom in S_C generated by Equation 7.44 (or by Equation 7.45 in the same form) also exists in S_C'.

(c) If a in $S_C.Axioms$ is of the form generated by Equation 7.45 such that

$$a = \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) \geq x\text{”}$$

then by Equation 7.45 there exists a $c \in C.Connection$ such that $c.Mult = Plus$, $c.Name = s$, if $p = HAS\text{-}PART$ then $c.Role = undefined$ else $c.Role = p$, $c.Plus.Integer = x$. (Note: If x = 0 then a is of the form generated by Equation 7.44 that was previously shown to be in OM'; therefore, in this section of the proof, I assume x > 0.)

Then by OMT-10 there exists an axiom $ax \in \mathbb{C}.Axiom$ of the form

$$ax = \text{“}SIZE(IMAGE(\textit{attr-name}(c)(X), Q)) \geq c.Plus.Integer\text{”}$$

and by Equation 7.64 there exists an axiom $a' \in S_C'.Axioms$ such that

$$a' = \text{“}X \in \textit{sort-of}(\textit{attr-name}(c)) \\ \Rightarrow SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) \geq c.Plus.Integer\text{”}$$

where $c.Plus.Integer = x$, if $c.Role$ was defined, $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Role) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Role) = c.Role = p$, or if $c.Role$ was not defined then $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Name\text{-}OBJ) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Name\text{-}$

$OBJ) = \textit{HAS-PART} = p$. Thus $a = a'$ and thus any axiom in S_C generated by Equation 7.45 also exists in S'_C.

(d) If a in $S_C.Axioms$ is of the form generated by Equation 7.46 such that

$$a = \text{“}X \in s \Rightarrow (SIZE(\{Y \mid p(X,Y)\}) = 0 \ \vee \ SIZE(\{Y \mid p(X,Y)\}) = 1)\text{”}$$

then there exists a $c \in \mathcal{C}.Connection$ such that (1) by Equation 7.46, $c.Mult = Optional$, $c.Name = s$, and if $p = \textit{HAS-PART}$ then $c.Role = undefined$ else $c.Role = p$, or (2) by Equation 7.47, $c.Mult = Specified$, $c.Name = s$, if $p = \textit{HAS-PART}$ then $c.Role = undefined$ else $c.Role = p$, and $c.Specified = \{sr_1, sr_2\}$, $sr_1 = \langle 0, undefined\rangle$, and $sr_2 = \langle 1, undefined\rangle$.

i. If $c.Mult = Optional$, then by OMT-11 there exists an axiom $ax_1 \in \mathbb{C}.Axiom$ such that

$$\begin{aligned} ax_2 = \text{“}&SIZE(IMAGE(\textit{attr-name}(c)(X), Q)) = 0 \\ &\vee SIZE(IMAGE(\textit{attr-name}(c)(X), Q)) = 1\text{”} \end{aligned}$$

where $\textit{attr-name}(c) = c.Role$ (if defined) or $c.Name\text{-}CLASS$.

ii. If $c.Mult = Specified$, then by OMT-12 there exists an axiom $ax_2 \in \mathbb{C}.Axiom$ such that

$$\begin{aligned} ax_2 = \text{“}&SIZE(IMAGE(\textit{attr-name}(c)(X), Q)) = sr_1.value1 \\ &\vee SIZE(IMAGE(\textit{attr-name}(c)(X), Q)) = sr_2.value2\text{”} \end{aligned}$$

where $s.value1 = 0$ and $s.value2 = 1$.

Thus it is obvious that $ax_1 = ax_2 = ax \in \mathbb{C}.Axiom$. From here, there are two paths to S'_C, via Equation 7.65 or 7.66.

i. Given ax, by Equation 7.65 there exists an axiom $a'_1 \in S'_C.Axioms$ such that

$$\begin{aligned} a'_1 = \text{“}X \in \textit{sort-of}(\textit{attr-name}(c)) \\ \Rightarrow (SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) = 0 \\ \vee\ SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) = 1)\text{”} \end{aligned}$$

where if *c.Role* was defined then $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Role) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Role) = c.Role = p$, or if *c.Role* was not defined then $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Name\text{-}OBJ) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Name\text{-}OBJ) = HAS\text{-}PART = p$. Thus $a = a'_1$.

ii. Given ax, by Equation 7.66 there exists an axioms $a'_2 \in S'_C.Axioms$ such that

$$\begin{aligned} a'_2 = \text{“}X \in \textit{sort-of}(\textit{attr-name}(c)) \\ \Rightarrow (SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) = s.value1 \\ \vee\ SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) = s.value2)\text{”} \end{aligned}$$

where $s.value1 = 0$, $s.value2 = 1$, and if *c.Role* was defined, $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Role) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Role) = c.Role = p$, or if *c.Role* was not defined then $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Name\text{-}OBJ) = c.Name = s$ (by OMT-5) and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Name\text{-}OBJ) = HAS\text{-}PART = p$. Thus $a = a'_2$.

Again it is clear that $a'_1 = a'_2 = a$ and thus any axiom in S_C generated by Equation 7.46 also exists in S'_C.

(e) If $a \in S_C.Axioms$ consists of the logical disjunction of n subaxioms of either of two forms

$$\text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = v_1\text{”}$$

or

$$\text{“}X \in s \Rightarrow (SIZE(\{Y \mid p(X,Y)\}) \geq v_1\ \vee\ SIZE(\{Y \mid p(X,Y)\}) \leq v_2)\text{”}$$

and assuming a is not of the form generated by Equations 7.43 or 7.46, then by Equation 7.47 there exists $c \in \mathcal{C}.Connection$ such that $c.Mult = Specified$, $c.Name = s$, and $c.Role = p$ if $p \neq HAS\text{-}PART$.

Then for each subaxiom $a_1 ... a_n$ in a, there exists some $s \in c.Specified$ (where s is of type $SPEC\text{-}RANGE$) such that $s.value1 = v_1$ and $s.value2 = v_2$.

Then by OMT-12 there exists an axiom $ax \in \mathbb{C}.Axiom$ where ax is the logical disjunction of the set of subaxioms generated for each $s_i \in c.Specified$ such that

$$s_i = \text{“}SIZE(attr\text{-}name(c)(X)) = s.value1\text{”}$$

or

$$\begin{aligned} s_i = \text{“}(&SIZE(attr\text{-}name(c)(X)) \geq s.value1 \\ &\vee SIZE(attr\text{-}name(c)(X)) \leq s.value2)\text{”} \end{aligned}$$

where $attr\text{-}name(c) = c.Role$, if defined, or $c.Name$ otherwise.

Then by Equation 7.66 there exists an a' in $S'_C.Axioms$ such that for each subaxiom of ax of one of the forms given above for s_i there exists a subaxiom in a' of the form

$$\begin{aligned} \text{“}X &\in sort\text{-}of(attr\text{-}name(c)) \\ &\Rightarrow SIZE(\{Y \mid om\text{-}pred(attr\text{-}name(c))(X,Y)\}) = s.value1\text{”} \end{aligned}$$

or

$$\begin{aligned} \text{“}X \in sort\text{-}of(attr\text{-}name(c)) \Rightarrow (&SIZE(\{Y \mid om\text{-}pred(attr\text{-}name(c))(X,Y)\}) \geq s.value1 \\ &\vee SIZE(\{Y \mid om\text{-}pred(attr\text{-}name(c))(X,Y)\}) \leq s.value2)\text{”} \end{aligned}$$

where if $c.Role$ is defined then $sort\text{-}of(attr\text{-}name(c)) = sort\text{-}of(c.Role) = c.Name = s$ and $om\text{-}pred(attr\text{-}name(c)) = om\text{-}pred(c.Role) = c.Role = p$. And, if $c.Role$ is not defined, then $sort\text{-}of(attr\text{-}name(c)) = sort\text{-}of(c.Name\text{-}OBJ) = c.Name = s$ and $om\text{-}pred(attr\text{-}name(c)) = om\text{-}pred(c.Name\text{-}OBJ) = HAS\text{-}PART = p$. Therefore, each subaxiom in $a' \in S'_C.Axioms$ is of the form

$$\text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = v_1\text{”}$$

or

$$\text{“}X \in s \Rightarrow (SIZE(\{Y \mid p(X,Y)\}) \geq v_1 \ \vee \ SIZE(\{Y \mid p(X,Y)\}) \leq v_2)\text{”}$$

and thus for each subaxiom in a there exists an equivalent subaxiom in a' and thus $a = a'$.

Therefore, since for each type of axiom in $S_C.Axioms$ there is an equivalent axiom in $S'_C.Axiom$ then $S_C.Axioms \subseteq S'_C.Axioms$.

$\underline{a \in S'.Axioms \Rightarrow a \in S.Axioms}$. The six possible sources of axioms in S'_C are analyzed below.

(a) If a in $S'_C.Axioms$ is of the form generated by Equation 7.62 such that

$$a = \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = 1\text{”}$$

then there exists an axiom, $ax \in \mathbb{C}.Axiom$ of the form

$$ax = \text{“}SIZE(IMAGE(n(X),Q)) = 1\text{”}$$

generated by either Equation 7.62 or 7.66 where $sort\text{-}of(n) = s$ and $om\text{-}pred(n) = p$. Given ax, there are two paths from C, OMT-8 or OMT-12.

i. If ax is generated by OMT-8 then

A. If $c.Role$ is defined then there exists a $c \in C.Connection$ such that

$$\begin{array}{l} c.Mult = ONE \\ n = attr\text{-}name(c) = c.Role \\ s = sort\text{-}of(n) = sort\text{-}(c.Role) = c.Name \\ p = om\text{-}pred(n) = om\text{-}pred(c.Role) = c.Role \\ comp\text{-}pred(c) = c.Role = p \end{array}$$

Thus by Equation 7.43 there exists an axiom $a'_1 \in S_C.Axiom$ such that

$$\begin{array}{rcl} a'_1 & = & \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = 1\text{”} \\ & = & \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = 1\text{”} \end{array}$$

B. If $c.Role$ is not defined then there exists a $c \in C.Connection$ such that

$$\begin{aligned}
&c.Mult = ONE \\
&n = attr\text{-}name(c) = c.Name - OBJ \\
&s = sort\text{-}of(n) = sort\text{-}(c.Name\text{-}OBJ) = c.Name \\
&p = om\text{-}pred(n) = om\text{-}pred(c.Name\text{-}OBJ) = HAS\text{-}PART \\
&comp\text{-}pred(c) = HAS\text{-}PART = p
\end{aligned}$$

Thus by Equation 7.43 there exists an axiom $a'_2 \in S_C.Axiom$ such that

$$\begin{aligned}
a'_2 &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = 1\text{”} \\
&= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = 1\text{”}
\end{aligned}$$

Thus $a'_1 = a'_2 \in S_C.Axioms = a \in S'_C.Axioms.$

ii. If ax is generated by OMT-12

A. If $c.Role$ is defined then there exists a $c \in C.Connection$ such that

$$\begin{aligned}
&c.Mult = Specified \\
&c.Mult = \{\langle 1, undefined\rangle\} \\
&n = attr\text{-}name(c) = c.Role \\
&s = sort\text{-}of(n) = sort\text{-}(c.Role) = c.Name \\
&p = om\text{-}pred(n) = om\text{-}pred(c.Role) = c.Role \\
&comp\text{-}pred(c) = c.Role = p
\end{aligned}$$

Thus by Equation 7.47 there exists an axiom $a'_1 \in S_C.Axiom$ such that

$$\begin{aligned}
a'_1 &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = x.value1\text{”} \\
&= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = 1\text{”}
\end{aligned}$$

B. If $c.Role$ is not defined then there exists a $c \in C.Connection$ such that

$$\begin{aligned}
&c.Mult = Specified \\
&c.Mult = \{\langle 1, undefined\rangle\} \\
&n = attr\text{-}name(c) = c.Name - OBJ \\
&s = sort\text{-}of(n) = sort\text{-}(c.Name\text{-}OBJ) = c.Name \\
&p = om\text{-}pred(n) = om\text{-}pred(c.Name\text{-}OBJ) = HAS\text{-}PART \\
&comp\text{-}pred(c) = HAS\text{-}PART = p
\end{aligned}$$

Thus by Equation 7.47 there exists an axiom $a'_2 \in S_C.Axiom$ such that

$$\begin{aligned} a_2' &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = x.value1\text{”} \\ &= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = 1\text{”} \end{aligned}$$

Thus $a_1' = a_2' \in S_C.Axioms = a \in S_C'.Axioms$.

Again it is clear that $a_1 = a_2 = a$ and thus any axiom in S_C' generated by Equation 7.62 also exists in S_C.

(b) If a in $S_C'.Axioms$ is of the form generated by Equation 7.63 such that

$$a = \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) \geq 0\text{”}$$

then there exists an axiom, $ax \in \mathbb{C}.Axiom$ of the form

$$ax = \text{“}SIZE(IMAGE(n(X),Q)) \geq 0\text{”}$$

generated by Equation 7.63 where $sort\text{-}of(n) = s$ and $om\text{-}pred(n) = p$.

Given ax, there are two paths from $\mathcal{C}$, OMT-9 or OMT-10.

i. If ax is generated by OMT-9 then

A. If $c.Role$ is defined then there exists a $c \in \mathcal{C}.Connection$ such that

$$\begin{aligned} &c.Mult = Many \\ &n = attr\text{-}name(c) = c.Role \\ &s = sort\text{-}of(n) = sort\text{-}(c.Role) = c.Name \\ &p = om\text{-}pred(n) = om\text{-}pred(c.Role) = c.Role \\ &comp\text{-}pred(c) = c.Role = p \end{aligned}$$

Thus by Equation 7.44 there exists an axiom $a_1' \in S_C.Axiom$ such that

$$\begin{aligned} a_1' &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) \geq 0\text{”} \\ &= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) \geq 0\text{”} \end{aligned}$$

B. If $c.Role$ is not defined then there exists a $c \in \mathcal{C}.Connection$ such that

$$\begin{aligned}
&c.Mult = Many \\
&n = attr\text{-}name(c) = c.Name - OBJ \\
&s = sort\text{-}of(n) = sort\text{-}(c.Name\text{-}OBJ) = c.Name \\
&p = om\text{-}pred(n) = om\text{-}pred(c.Name\text{-}OBJ) = HAS\text{-}PART \\
&comp\text{-}pred(c) = HAS\text{-}PART = p
\end{aligned}$$

Thus by Equation 7.44 there exists an axiom $a_2' \in S_C.Axiom$ such that

$$\begin{aligned}
a_2' &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) \geq 0\text{”} \\
&= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) \geq 0\text{”}
\end{aligned}$$

Thus $a_1' = a_2' \in S_C.Axioms = a \in S_C'.Axioms.$

ii. If ax is generated by OMT-10

A. If $c.Role$ is defined then there exists a $c \in C.Connection$ such that

$$\begin{aligned}
&c.Mult = Plus \\
&c.Plus.integer = 0 \\
&n = attr\text{-}name(c) = c.Role \\
&s = sort\text{-}of(n) = sort\text{-}(c.Role) = c.Name \\
&p = om\text{-}pred(n) = om\text{-}pred(c.Role) = c.Role \\
&comp\text{-}pred(c) = c.Role = p
\end{aligned}$$

Thus by Equation 7.45 there exists an axiom $a_1' \in S_C.Axiom$ such that

$$\begin{aligned}
a_1' &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) \geq x.Plus.integer\text{”} \\
&= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) \geq 0\text{”}
\end{aligned}$$

B. If $c.Role$ is not defined then there exists a $c \in C.Connection$ such that

$$\begin{aligned}
&c.Mult = Plus \\
&c.Plus.integer = 0 \\
&n = attr\text{-}name(c) = c.Name - OBJ \\
&s = sort\text{-}of(n) = sort\text{-}(c.Name\text{-}OBJ) = c.Name \\
&p = om\text{-}pred(n) = om\text{-}pred(c.Name\text{-}OBJ) = HAS\text{-}PART \\
&comp\text{-}pred(c) = HAS\text{-}PART = p
\end{aligned}$$

Thus by Equation 7.45 there exists an axiom $a_2' \in S_C.Axiom$ such that

$$\begin{aligned} a'_2 &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) \geq c.Plus.integer\text{”} \\ &= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) \geq 0\text{”} \end{aligned}$$

Thus $a'_1 = a'_2 \in S_C.Axioms = a \in S'_C.Axioms$.

Therefore, it is clear that $a_1 = a_2 = a$ and thus any axiom in S'_C generated by Equation 7.62 also exists in S_C.

(c) If a in $S'_C.Axioms$ is of the form generated by Equation 7.64 (assuming x > 0) such that

$$a = \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) > x\text{”}$$

then there exists an axiom, $ax \in \mathbb{C}.Axiom$ of the form

$$ax = \text{“}SIZE(IMAGE(n(X),Q)) > x\text{”}$$

generated by either Equation 7.64 where $sort\text{-}of(n) = s$ and $om\text{-}pred(n) = p$.

Given ax, by OMT-10 there exists a $c \in \mathcal{C}.Connection$ such that

i. If *c.Role* is defined then

$$\begin{aligned} &c.Mult = Plus \\ &c.Plus.integer = x \\ &n = attr\text{-}name(c) = c.Role \\ &s = sort\text{-}of(n) = sort\text{-}(c.Role) = c.Name \\ &p = om\text{-}pred(n) = om\text{-}pred(c.Role) = c.Role \\ &comp\text{-}pred(c) = c.Role = p \end{aligned}$$

Thus by Equation 7.45 there exists an axiom $a'_1 \in S_C.Axiom$ such that

$$\begin{aligned} a'_1 &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) \geq c.Plus.integer\text{”} \\ &= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) \geq x\text{”} \end{aligned}$$

ii. If *c.Role* is not defined then

$$\begin{aligned}
&c.Mult = PLUS\\
&c.Plus.integer = x\\
&n = attr\text{-}name(c) = c.Name - OBJ\\
&s = sort\text{-}of(n) = sort\text{-}(c.Name\text{-}OBJ) = c.Name\\
&p = om\text{-}pred(n) = om\text{-}pred(c.Name\text{-}OBJ) = HAS\text{-}PART\\
&comp\text{-}pred(c) = HAS\text{-}PART = p
\end{aligned}$$

Thus by Equation 7.45 there exists an axiom $a'_2 \in S_C.Axiom$ such that

$$\begin{aligned}
a'_2 &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) \geq c.Plus.integer\text{”}\\
&= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) \geq x\text{”}
\end{aligned}$$

Thus $a'_1 = a'_2 \in S_C.Axioms = a \in S'_C.Axioms.$

Again it is clear that $a_1 = a_2 = a$ and thus any axiom in S'_C generated by Equation 7.64 also exists in S_C.

(d) If a in $S'_C.Axioms$ is of the form generated by Equation 7.65 such that

$$a = \text{“}X \in s \Rightarrow (SIZE(\{Y \mid p(X,Y)\}) = 0 \vee SIZE(\{Y \mid p(X,Y)\}) = 1)\text{”}$$

then there exists an axiom, $ax \in \mathbb{C}.Axiom$ of the form

$$ax = \text{“}SIZE(IMAGE(n(X),Q)) = 0 \vee SIZE(IMAGE(n(X),Q)) = 1\text{”}$$

generated by either Equation 7.65 or 7.66 where $sort\text{-}of(n) = s$ and $om\text{-}pred(n) = p$. Given ax, there are two paths from $\mathcal{C}$, OMT-11 or OMT-12.

i. If ax is generated by OMT-11 then

A. If $c.Role$ is defined then there exists a $c \in \mathcal{C}.Connection$ such that

$$\begin{aligned}
&c.Mult = Optional\\
&n = attr\text{-}name(c) = c.Role\\
&s = sort\text{-}of(n) = sort\text{-}(c.Role) = c.Name\\
&p = om\text{-}pred(n) = om\text{-}pred(c.Role) = c.Role\\
&comp\text{-}pred(c) = c.Role = p
\end{aligned}$$

Thus by Equation 7.46 there exists an axiom $a'_1 \in S_C.Axiom$ such that

$$\begin{aligned} a'_1 &= \text{“}X \in c.Name \Rightarrow (SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = 0 \\ &\qquad \vee\ SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = 1)\text{”} \\ &= \text{“}X \in s \Rightarrow (SIZE(\{Y \mid p(X,Y)\}) = 0 \vee\ SIZE(\{Y \mid p(X,Y)\}) = 1)\text{”} \end{aligned}$$

B. If *c.Role* is not defined then there exists a $c \in \mathcal{C}.Connection$ such that

$$\begin{aligned} &c.Mult = Specified \\ &n = attr\text{-}name(c) = c.Name - OBJ \\ &s = sort\text{-}of(n) = sort\text{-}(c.Name\text{-}OBJ) = c.Name \\ &p = om\text{-}pred(n) = om\text{-}pred(c.Name\text{-}OBJ) = HAS\text{-}PART \\ &comp\text{-}pred(c) = HAS\text{-}PART = p \end{aligned}$$

Thus by Equation 7.46 there exists an axiom $a'_2 \in S_C.Axiom$ such that

$$\begin{aligned} a'_2 &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = 1\text{”} \\ &= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = 1\text{”} \end{aligned}$$

Thus $a'_1 = a'_2 \in S_C.Axioms = a \in S'_C.Axioms.$

ii. If *ax* is generated by OMT-12

A. If *c.Role* is defined then there exists a $c \in \mathcal{C}.Connection$ such that

$$\begin{aligned} &c.Mult = Specified \\ &c.Mult = \{\langle 0, undefined\rangle, \langle 1, undefined\rangle\} \\ &n = attr\text{-}name(c) = c.Role \\ &s = sort\text{-}of(n) = sort\text{-}(c.Role) = c.Name \\ &p = om\text{-}pred(n) = om\text{-}pred(c.Role) = c.Role \\ &comp\text{-}pred(c) = c.Role = p \end{aligned}$$

Thus by Equation 7.47 there exists an axiom $a'_1 \in S_C.Axiom$ such that

$$\begin{aligned} a'_1 &= \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = x.value1\text{”} \\ &= \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = 1\text{”} \end{aligned}$$

B. If *c.Role* is not defined then there exists a $c \in \mathcal{C}.Connection$ such that

$c.Mult = Specified$
$c.Mult = \{\langle 0, undefined\rangle, \langle 1, undefined\rangle\}$
$n = attr\text{-}name(c) = c.Name - OBJ$
$s = sort\text{-}of(n) = sort\text{-}(c.Name\text{-}OBJ) = c.Name$
$p = om\text{-}pred(n) = om\text{-}pred(c.Name\text{-}OBJ) = HAS\text{-}PART$
$comp\text{-}pred(c) = HAS\text{-}PART = p$

Thus by Equation 7.47 there exists an axiom $a'_2 \in S_C.Axiom$ such that

$$\begin{array}{rcl} a'_1 & = & \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid comp\text{-}pred(c)(X,Y)\}) = x.value1\text{”} \\ & = & \text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = 1\text{”} \end{array}$$

Thus $a'_1 = a'_2 \in S_C.Axioms = a \in S'_C.Axioms.$

Again it is clear that $a_1 = a_2 = a$ and thus any axiom in S'_C generated by Equation 7.65 also exists in S_C.

(e) If a' in $S'_C.Axioms$ is of the form generated by Equation 7.66 such that it consists of the logical disjunction of n subaxioms of either of two forms

$$\text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = v_1\text{”}$$

or

$$\text{“}X \in s \Rightarrow (SIZE(\{Y \mid p(X,Y)\}) \geq v_1 \ \vee \ SIZE(\{Y \mid p(X,Y)\}) \leq v_2)\text{”}$$

and assuming a' is not of the form generated by Equations 7.62 or 7.65, then by Equation 7.66, there exists an axiom ax in $\mathbb{C}.Axiom$ where ax is the logical disjunction of the set of subaxioms generated for each s_i in a' such that

$$s_i = \text{“}SIZE(n(X)) = v_1\text{”}$$

or

$$s_i = \text{“}(SIZE(n(X)) \geq v_1 \ \vee \ SIZE(n(X)) \leq v_2)\text{”}$$

where $s = sort\text{-}of(n)$ and $p = om\text{-}pred(c)$. Then by OMT-12 there exists a $c \in \mathcal{C}.Connection$ such that $c.Mult = Specified$, there exists some $\langle v_1, v_2\rangle \in c.Specified$ (v_2 may be undefined) for each s_i in ax, and the $attr\text{-}name(c) = n$.

i. By the definition of the function *om-pred* in Equation 7.75, if $p = \textit{om-pred}(c) = \textit{HAS-PARTS}$ then *c.Role* is undefined and

$$\begin{aligned}
&n = \textit{attr-name}(c) = c.Name - OBJ\\
&s = \textit{sort-of}(n) = \textit{sort-of}(c.Name\text{-}OBJ) = c.Name\\
&p = \textit{om-pred}(n) = \textit{om-pred}(c.Name\text{-}OBJ) = \textit{HAS-PART}\\
&\textit{comp-pred}(c) = \textit{HAS-PART} = p
\end{aligned}$$

ii. By the definition of the function *om-pred* in Equation 7.75, if $p = \textit{om-pred}(c) \neq \textit{HAS-PARTS}$ then *c.Role* is defined and

$$\begin{aligned}
&n = \textit{attr-name}(c) = c.Role\\
&s = \textit{sort-of}(n) = \textit{sort-of}(c.Role) = c.Name\\
&p = \textit{om-pred}(n) = \textit{om-pred}(c.Role) = c.Role\\
&\textit{comp-pred}(c) = c.Role = p
\end{aligned}$$

Then by Equation 7.47 there exists an a in $S_C.Axioms$ such that for each $\textit{spec-range} \in c.Specified$ there exists a subaxiom in a of the form

$$\begin{aligned}
&\text{“}X \in \textit{sort-of}(\textit{attr-name}(c))\\
&\quad \Rightarrow SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) = s.value1\text{”}
\end{aligned}$$

or

$$\begin{aligned}
&\text{“}X \in \textit{sort-of}(\textit{attr-name}(c)) \Rightarrow (SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) \geq s.value1\\
&\quad \vee\ SIZE(\{Y \mid \textit{om-pred}(\textit{attr-name}(c))(X,Y)\}) \leq s.value2)\text{”}
\end{aligned}$$

where if *c.Role* is defined then $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Role) = c.Name = s$ and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Role) = c.Role = p$. And, if *c.Role* is not defined, then $\textit{sort-of}(\textit{attr-name}(c)) = \textit{sort-of}(c.Name\text{-}OBJ) = c.Name = s$ and $\textit{om-pred}(\textit{attr-name}(c)) = \textit{om-pred}(c.Name\text{-}OBJ) = \textit{HAS-PART} = p$. Therefore, each subaxiom in $a \in S_C.Axioms$ is of the form

$$\text{“}X \in s \Rightarrow SIZE(\{Y \mid p(X,Y)\}) = v_1\text{”}$$

or

$$\text{“}X \in s \Rightarrow (SIZE(\{Y \mid p(X,Y)\}) \geq v_1 \;\vee\; SIZE(\{Y \mid p(X,Y)\}) \leq v_2)\text{”}$$

and thus for each subaxiom in a' there exists an equivalent subaxiom in a and thus $a = a'$.

Therefore, since for each type of axiom in $S'_C.Axioms$ there is an equivalent axiom in $S_C.Axiom$ then $S'_C.Axioms \subseteq S_C.Axioms$, and, since I have previously shown that $S_C.Axioms \subseteq S'_C.Axioms$, it is clear that $S_C.Axioms = S'_C.Axioms$

Association Axioms. Since a is in an association specification, it must have been generated by one of the Equations 7.51 – 7.55. Each possibility is discussed below.

(a) If a in $S_A.Axioms$ is of the form generated by Equation 7.51 such that

$$a = \text{“}X \in s \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) = 1\text{”}$$

then there exists a $c \in C.Connection$ such that (1) by Equation 7.51, $c.Mult = One$ and $c.Name = s$, or (2) by Equation 7.55 $c.Mult = Specified$, $c.Name = s$, $c.Specified = \{sr\}$, and sr $= \langle 1, undefined \rangle$.

i. If $c.Mult = One$, then by OMT-50 there exists an axiom $ax_1 \in \mathbb{C}.Axiom$ such that

$$ax_1 = \text{“}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) = 1\text{”}$$

ii. If $c.Mult = Specified$, then by OMT-54 there exists an axiom $ax_2 \in \mathbb{C}.Axiom$ such that

$$ax_1 = \text{“}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) = sr.value1\text{”}$$

Thus, since $sr.value1 = 1$, $ax_1 = ax_2 = ax \in \mathbb{C}.Axiom$, then by Equation 7.70 there exists an axiom $a' \in S'_A.Axioms$ such that

$$a' = \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid S'_A.Name(X,Y)\}) = 1\text{”}$$

Thus $a = a'$ since $c.Name = s$ and $S_A.Name = S'_A.Name$; therefore any axiom in S_A generated by Equation 7.51 (or by Equation 7.55 in the form of Equation 7.51) also exists in S'_A.

(b) If a in $S_A.Axioms$ is of the form generated by Equation 7.52 such that

$$a = \text{"}X \in s \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq 0\text{"}$$

then there exists a $c \in \mathcal{C}.Connection$ such that (1) by Equation 7.52, $c.Mult = Many$ and $c.Name = s$, or (2) by Equation 7.53, $c.Mult = Plus$, $c.Name = s$, and $c.Plus.Integer = 0$.

i. If $c.Mult = Many$, then by OMT-51 there exists an axiom $ax_1 \in \mathbb{C}.Axiom$ such that

$$ax_1 = \text{"}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) \geq 0\text{"}$$

where $attr\text{-}name(c) = c.Role$ (if defined) or $c.Name\text{-}CLASS$.

ii. If $c.Mult = Plus$, then by OMT-52 there exists an axiom $ax_2 \in \mathbb{C}.Axiom$ such that

$$ax_1 = \text{"}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) \geq c.Plus.integer\text{"}$$

Thus, since $c.Plus.Integer = 0$, it is obvious that $ax_1 = ax_2 = ax \in \mathbb{C}.Axiom$. Then by Equation 7.71 there exists an axiom $a' \in S'_A.Axioms$ such that

$$a' = \text{"}X \in c.Name \Rightarrow SIZE(\{Y \mid S'_A.Name(X,Y)\}) \geq c.Plus.integer\text{"}$$

and since $c.Name = s$ and $S_A.Name = S'_A.Name$, $a = a'$ and any axiom in S_A generated by Equation 7.52 (or by Equation 7.53 in the same form) also exists in S'_A.

(c) If a in $S_A.Axioms$ is of the form generated by Equation 7.53 such that

$$a = \text{"}X \in s \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq x\text{"}$$

then by Equation 7.53 there exists a $c \in \mathcal{C}.Connection$ such that $c.Mult = Plus$, $c.Name = s$, and $c.Plus.Integer = x$. (Note: If x = 0 then a is of the form generated by Equation 7.52 that was previously shown to be in OM'; therefore, in this section of the proof, I assume x > 0.)

Then by OMT-52 there exists an axiom $ax \in \mathbb{C}.Axiom$ of the form

$$ax = \text{"}X \in c.Name \Rightarrow SIZE(IMAGE(A,X) \geq c.Plus.Integer\text{"}$$

and thus by Equation 7.72 there exists an axiom $a' \in S'_A.Axioms$ such that

$$a' = \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid S'_A.Name(X,Y)\}) \geq c.Plus.Integer\text{”}$$

and since $c.Name = s$, $c.Plus.Integer = x$, and $S_A.Name = S'_A.Name$, it must be true that $a = a'$ and thus any axiom in S_A generated by Equation 7.53 also exists in S'_A.

(d) If a in $S_A.Axioms$ is of the form generated by Equation 7.54 such that

$$a = \text{“}X \in s \Rightarrow (SIZE(\{Y \mid S_A.Name(X,Y)\}) = 0 \\ \vee\ SIZE(\{Y \mid S_A.Name(X,Y)\}) = 1)\text{”}$$

then there exists a $c \in C.Connection$ such that (1) by Equation 7.54, $c.Mult = Optional$ and $c.Name = s$, or (2) by Equation 7.55, $c.Mult = Specified$, $c.Name = s$, $c.Specified = \{sr_1, sr_2\}$, $sr_1 = \langle 0, undefined\rangle$, and $sr_2 = \langle 1, undefined\rangle$.

i. If $c.Mult = Optional$, then by OMT-53 there exists an axiom $ax_1 \in \mathbb{C}.Axiom$ such that

$$ax_1 = \text{“}X \in c.Name \Rightarrow (SIZE(IMAGE(A,X) = 0 \\ \vee\ SIZE(IMAGE(A,X) = 1)\text{”}$$

ii. If $c.Mult = Specified$, then by OMT-54 there exists an axiom $ax_2 \in \mathbb{C}.Axiom$ such that

$$ax_1 = \text{“}X \in c.Name \Rightarrow (SIZE(IMAGE(A,X) = sr_1.value1 \\ \vee\ SIZE(IMAGE(A,X) = sr_2.value1)\text{”}$$

Thus it is obvious that $ax_1 = ax_2 = ax \in \mathbb{C}.Axiom$ and by Equation 7.73 there exists an axiom $a' \in S'_A.Axioms$ such that

$$a' = \text{“}X \in c.Name \Rightarrow (SIZE(\{Y \mid S'_A.Name(X,Y)\}) = 0 \\ \vee\ SIZE(\{Y \mid S'_A.Name(X,Y)\}) = 1)\text{”}$$

and since $c.Name = s$ and $S'_A Name = S_A.Name$, $a = a'$.

Again it is clear that $a'_1 = a'_2 = a$ and thus any axiom in S_A generated by Equation 7.54 also exists in S'_A.

(e) If $a \in S_A.Axioms$ is of the form generated by Equation 7.55 such that a consists of the logical disjunction of n subaxioms of either of two forms

$$\text{“}X \in s \Rightarrow SIZE(\{Y \mid pred(X,Y)\}) = v_1\text{”}$$

or

$$\text{“}X \in s \Rightarrow (SIZE(\{Y \mid pred(X,Y)\}) \geq v_1 \ \wedge \ SIZE(\{Y \mid p(X,Y)\}) \leq v_2)\text{”}$$

and assuming a is not of the form generated by Equations 7.51 or 7.54, then by Equation 7.55 there exists $c \in \mathcal{C}.Connection$ such that $c.Mult = Specified$, $c.Name = s$, and $pred = S_A.Name$.

Then for each subaxiom $a_1...a_n$ in a, there exists some $s \in c.Specified$ (where s is of type $SPEC\text{-}RANGE$) such that $s.value1 = v_1$ and $s.value2 = v_2$ (v_2 may be undefined).

Then by OMT-54 there exists an axiom $ax \in \mathbb{C}.Axiom$ such that ax is the logical disjunction of the set of subaxioms generated for each $s_i \in c.Specified$ such that

$$s_i = \text{“}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) = s.value1\text{”}$$

or

$$s_i = \text{“}X \in c.Name \Rightarrow (SIZE(IMAGE(A,X)) \geq s.value1 \\ \wedge SIZE(IMAGE(A,X)) \leq s.value2)\text{”}$$

Then by Equation 7.74 there exists an a' in $S'_A.Axioms$ such that for each subaxiom of ax of one of the forms given above for s_i there exists a subaxiom in a' of the form

$$\text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid S'_A.Name(X,Y)\}) = s.value1\text{”}$$

or

$$\text{“}X \in c.Name \Rightarrow (SIZE(\{Y \mid S'_A.Name(X,Y)\}) \geq s.value1 \\ \wedge SIZE(\{Y \mid S'_A.Name(X,Y)\}) \leq s.value2)\text{”}$$

where $c.Name = s$ and $S'_A.Name = S_A.Name$. Therefore, each subaxiom in $a' \in S'_A.Axioms$ is of the form

$$\text{“}X \in s \Rightarrow SIZE(\{Y \mid pred(X,Y)\}) = v_1\text{”}$$

or

$$\text{“}X \in s \Rightarrow (SIZE(\{Y \mid pred(X,Y)\}) \geq v_1 \ \vee \ SIZE(\{Y \mid pred(X,Y)\}) \leq v_2)\text{”}$$

and thus for each subaxiom in a there exists an equivalent subaxiom in a' and thus $a = a'$.

Therefore, since for each type of axiom in $S_A.Axioms$ there is an equivalent axiom in $S'_A.Axiom$ it must be true that $S_A.Axioms \subseteq S'_A.Axioms$.

<u>$a \in S'.Axioms \Rightarrow a \in S.Axioms$.</u> The six possible sources of axioms in S'_A are analyzed below.

(a) If a in $S'_A.Axioms$ is of the form generated by Equation 7.70 such that

$$a = \text{“}X \in s \Rightarrow SIZE(\{Y \mid S'_A.Name(X,Y)\}) = 1\text{”}$$

then there exists an axiom, $ax \in \mathbb{C}.Axiom$ of the form

$$ax = \text{“}X \in sSIZE(IMAGE(A,X)) = 1\text{”}$$

generated by either Equation 7.70 or 7.74.

Given ax, there are two paths from $\mathcal{C}$, OMT-50 or OMT-54.

i. If ax is generated by OMT-50 then $c.Mult = ONE$ and $c.Name = s$, and by Equation 7.51 there exists an axiom $a'_1 \in S_A.Axiom$ such that

$$a' = \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) = 1\text{”}$$

Since $c.Name = s$ and $S_A.Name = S'_A.Name$, $a' = a$.

ii. If ax is generated by OMT-54 then there exists a $c \in \mathcal{C}.Connection$ such that $c.Mult = Specified$, $c.Specified = \{sr\}$, and $sr = \langle 1, undefined \rangle$. Thus by Equation 7.55 there exists an axiom $a' \in S_A.Axiom$ such that

$$a' = \text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) = sr.value1\text{”}$$

Since $c.Name = s$, $sr.value1 = 1$, and $S_A.Name = S'_A.Name$, $a' \in S_A.Axioms = a \in S'_A.Axioms$ and thus any axiom in S'_A generated by Equation 7.70 also exists in S_A.

(b) If a in $S'_A.Axioms$ is of the form generated by Equation 7.71 such that

$$a = \text{``}X \in s \Rightarrow SIZE(\{Y \mid S'_A.Name(X,Y)\}) \geq 0\text{''}$$

then there exists an axiom, $ax \in \mathbb{C}.Axiom$ of the form

$$ax = \text{``}X \in sSIZE(IMAGE(A,X)) \geq 0\text{''}$$

generated by Equation 7.71.

Given ax, there are two paths from $\mathcal{C}$, OMT-51 or OMT-52.

i. If ax is generated by OMT-51 then there exists a $c \in \mathcal{C}.Connection$ such that $c.Mult = Many$ and $c.Name = s$; therefore, by Equation 7.52 there exists an axiom $a' \in S_A.Axiom$ such that

$$a = \text{``}X \in c.Name \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq 0\text{''}$$

Therefore since $s = c.Name$ and $S_A.Name = S'_A.Name$, $a' = a$.

ii. If ax is generated by OMT-52 then there exists a $c \in \mathcal{C}.Connection$ such that $c.Mult = Plus$, $c.Plus.integer = 0$, and $s = c.Name$ and by Equation 7.53 there exists an axiom $a' \in S_A.Axiom$ such that

$$a = \text{``}X \in c.Name \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq 0\text{''}$$

Since $s = c.Name$ and $S_A.Name = S'_A.Name$, $a' = a$; therefore any axiom in S'_A generated by Equation 7.70 also exists in S_A.

(c) If a in $S'_A.Axioms$ is of the form generated by Equation 7.72 (assuming x > 0) such that

$$a = \text{``}X \in s \Rightarrow SIZE(\{Y \mid S'_A.Name(X,Y)\}) > x\text{''}$$

then there is an axiom, $ax \in \mathbb{C}.Axiom$

$$X \in S \Rightarrow ax = \text{``}SIZE(IMAGE(A,X)) > x\text{''}$$

generated by Equation 7.72.

Given ax, by OMT-52 there exists a $c \in \mathcal{C}.Connection$ such that$c.Mult = Plus$, $c.Plus.integer = x$, and $c.Name = s$. Therefore, by Equation 7.53 there exists an axiom $a' \in S_A.Axiom$ such that

$$a = \text{"}X \in c.Name \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) > c.Plus.integer\text{"}$$

Again, since $c.Plus.integer = x$ and $c.Name = s$, it is clear that $a = a$ and thus any axiom in S'_A generated by Equation 7.72 also exists in S_A.

(d) If a in $S'_A.Axioms$ is of the form generated by Equation 7.73 such that

$$a = \text{"}X \in s \Rightarrow (SIZE(\{Y \mid S'_A.Name(X,Y)\}) = 0$$
$$\vee SIZE(\{Y \mid S'_A.Name(X,Y)\}) = 1)\text{"}$$

then there os an axiom, $ax \in \mathbb{C}.Axiom$

$$ax = \text{"}X \in s \Rightarrow (SIZE(IMAGE(A,X)) = 0 \vee SIZE(IMAGE(A,X)) = 1)\text{"}$$

generated by either Equation 7.73 or 7.74.

Given ax, there are two paths from $\mathcal{C}$, OMT-53 or OMT-54.

i. If ax is generated by OMT-53 then there exists a $c \in \mathcal{C}.Connection$ such that $c.Mult = Optional$ and $c.Name = s$. Thus by Equation 7.54 there exists an axiom $a' \in S_A.Axiom$ such that

$$a = \text{"}X \in c.Name \Rightarrow (SIZE(\{Y \mid S_A.Name(X,Y)\}) = 0$$
$$\vee SIZE(\{Y \mid S_A.Name(X,Y)\}) = 1)\text{"}$$

Therefore since $s = c.Name$ and $S_A.Name = S'_A.Name$, $a' = a$.

ii. If ax is generated by OMT-54 then there exists a $c \in \mathcal{C}.Connection$ such that$c.Mult = Specified$, $c.Name = s$, $c.Specified = \{sr_1, sr_2\}$, $sr_1 = \langle 0, undefined\rangle$, and $sr_2 = \langle 1, undefined\rangle$.

Thus by Equation 7.55 there exists an axiom $a' \in S_A.Axiom$ such that

$$a = \text{"}X \in c.Name \Rightarrow (SIZE(\{Y \mid S_A.Name(X,Y)\}) = 0$$
$$\vee SIZE(\{Y \mid S_A.Name(X,Y)\}) = 1)\text{"}$$

Therefore since $s = c.Name$ and $S_A.Name = S'_A.Name$, $a' = a$.

Again it is clear that $a' = a$ in both cases and, therefore, any axiom in S'_A generated by Equation 7.73 also exists in S_A.

(e) If a' in $S'_A.Axioms$ is of the form generated by Equation 7.74 such that it consists of the logical disjunction of n subaxioms of either of two forms

$$\text{“}X \in s \Rightarrow SIZE(\{Y \mid S'_A.Name(X,Y)\}) = v_1\text{”}$$

or

$$\text{‘}X \in s \Rightarrow (SIZE(\{Y \mid S'_A.Name(X,Y)\}) \geq v_1 \\ \vee\ SIZE(\{Y \mid S'_A.Name(X,Y)\}) \leq v_2)\text{”}$$

and assuming a' is not of the form generated by Equations 7.70 or 7.73, then by Equation 7.74 there exists an axiom ax in $\mathbb{C}.Axiom$ where ax is the logical disjunction of the set of subaxioms generated for each s_i in a' such that

$$s_i = \text{“}X \in c.Name \Rightarrow SIZE(IMAGE(A,X)) = v_1\text{”}$$

or

$$s_i = \text{“}X \in c.Name \Rightarrow (SIZE(IMAGE(A,X)) \geq v_1 \\ \vee\ SIZE(IMAGE(A,X)) \leq v_2)\text{”}$$

Thus by OMT-54 there exists a $c \in \mathcal{C}.Connection$ such that $c.Mult = Specified$, and there exists some $\langle v_1, v_2 \rangle \in c.Specified$ (v_2 may be undefined) for each s_i in ax.

Then by Equation 7.55 there exists an a in $S_A.Axioms$ such that for each *spec-range* $\in$ $c.Specified$ there exists a subaxiom in a of the form

$$\text{“}X \in c.Name \Rightarrow SIZE(\{Y \mid S_A.Name(X,Y)\}) = s.value1\text{”}$$

or

$$\text{“}X \in c.Name \Rightarrow (SIZE(\{Y \mid S_A.Name(X,Y)\}) \geq s.value1 \\ \vee\ SIZE(\{Y \mid S_A.Name(X,Y)\}) \leq s.value2)\text{”}$$

Since $c.Name = s$ and $S_A.Name = S'_A.Name$, it must be true that for each subaxiom in a' there exists an equivalent subaxiom in a, and thus $a = a'$. Therefore, any axiom in S'_A generated by Equation 7.74 also exists in S_A.

Therefore, since for each type of axiom in $S'_A.Axioms$, there is an equivalent axiom in $S_A.Axiom$, then $S'_A.Axioms \subseteq S_A.Axioms$, and, since I have previously shown that $S_A.Axioms \subseteq S'_A.Axioms$ it is clear that $S_A.Axioms = S'_A.Axioms$

Therefore, since each specification in OM is in OM', each specification in OM' is in OM, and each component within each specification in OM is in the associated specification in OM' and vice versa, it is obvious that $OM = OM'$. □

F.2 Dynamic Model Correctness Proof

In this section, Theorem VII.2 is proved.

Proof. Preservation of the dynamic model semantics by τ is established by showing the equivalence of two sets of dynamic model semantics, M and M', created from a generic OMT domain theory, $\mathcal{G}$. M is the dynamic model semantics defined by transforming $\mathcal{G}$ by φ while M' is the dynamic model semantics defined by transforming $\mathcal{G}$ by τ, into an O-SLANG domain theory $\mathbb{O}$, and then by ω. In this proof, I assume that $\mathcal{G}$ has a well defined dynamic model in which $\mathcal{C}$ is a class.

I prove the theorem by showing that, given a valid generic OMT domain theory $\mathcal{G}$, each component defined in M (Q, Σ, Δ and δ) exist in M' and that each component defined in M' exists in M.

1. $Q = Q'$.

To show that $Q = Q'$, I first show that $Q \subseteq Q'$ and then that $Q' \subseteq Q$. By assumption, $\mathcal{C}$ is some class in $\mathcal{G}$ with a well-defined dynamic model. Therefore, for each state s in $\mathcal{C}.State$, by φ, $s.Name$ is in Q.

If s in $C.State$, then by rule OMT-68 $\langle s.Name, , x\rangle$ is in $\mathbb{C}.State$ (with the exception of *Initial-State-Marker* by function π_s as defined in Equation 7.19) and by ω, $s.Name$ is in Q'. The definition of Q' in ω also explicitly re-inserts the *Initial-State-Marker*; therefore, $Q \subseteq Q'$.

By ω, if $s.Name$ is in Q' then $s.Name$ must be the name of some state s in $\mathbb{C}.State$ or the *Initial-State-Marker*. Rules OMT-68 and OMT-72 are the only translation rules in τ that create elements of $\mathbb{C}.State$. However, since OMT-72 translates substates of some state s in the states of C and, by assumption the dynamic model has been translated into a simple finite state machine automata as defined in Section 5.4.4 without substates, if s is in $\mathbb{C}.State$ it had to be placed there by OMT-68, and thus $s.Name$ is the name of some state in $C.State$. Since the *Initial-State-Marker* is present in $C.State$, it is in Q as well. Thus $Q' \subseteq Q$ and therefore $Q = Q'$.

2. $\Sigma = \Sigma'$.

To show that $\Sigma = \Sigma'$, I first show that $\Sigma \subseteq \Sigma'$ and then that $\Sigma' \subseteq \Sigma$. By definition of Σ in φ, for $t.Name$ to be in the set Σ, the corresponding transition, t, must exist in $C.Transition$. Then, if $t \in C.Transition$, by OMT-74, an event $\langle t.Name, x, y\rangle$ is in $\mathbb{C}.Event$ that is mapped by ω to Σ'. Thus $\Sigma \subseteq \Sigma'$.

Assume $t.Name$ is in Σ'. Then by definition of Σ' in ω, there must be some $\langle t.Name, x', y'\rangle$ in $\mathbb{C}.Event$. And, since Rule OMT-74 is the only rule in τ that translates components in $\mathcal{G}$ to $\mathbb{C}.Event$ (OMT-17 only creates events in $\mathbb{C}.Event$ when there is *no* dynamic model defined by the user), $t.Name$ must be the name of some transition $t \in C.Transition$, which by φ implies $t.Name \in \Sigma$. Therefore $\Sigma' \subseteq \Sigma$ and knowing that $\Sigma = \Sigma'$ from above, implies $\Sigma' = \Sigma$.

3. $\Delta = \Delta'$

To show that $\Delta = \Delta'$, I first show that $\Delta \subseteq \Delta'$ and then that $\Delta' \subseteq \Delta$. Assume t is some transition in $C.Transition$, then by definition of Δ in φ, for each a in $t.Action$, $msig(a)$, as defined in Equation 7.77, is in Δ if and only if $a.Name \neq SEND$ and $msig(a.Action)$ is in Δ if and only if $a.Name = SEND$.

Assume $a.Name \neq SEND$ and thus $msig(a) = \langle a.Name, [\mathcal{C}.Name] \parallel domain(a.Parameter), [\mathcal{C}.Name] \rangle$ is in Δ by definition of φ. By Rule OMT-75, $m = \langle a.Name, [\mathcal{C}.Name] \parallel domain(a.Parameter), [\mathcal{C}.Name] \rangle \in \mathbb{C}.Method$, by Rule OMT-74 $\langle t.Name, [\mathcal{C}.Name] \parallel domain(t.Parameter), [\mathcal{C}.Name] \rangle \in \mathbb{C}.Event$, and by OMT-82 an axiom of the form

$$\text{“}oldstate(t.FromState) \wedge guard \Rightarrow newstate(t)$$
$$\wedge \ldots ATTR\text{-}EQUAL(t.Name(\ \ldots\ ), a.Name(\ \ldots\ ))\ \ldots\text{”}$$

is in $\mathbb{C}.Axiom$. Therefore, by definition of Δ' in ω, $m \in \Delta'$, and $msig(a) \in \Delta = \langle a.Name, [\mathcal{C}.Name] \| domain(a.Parameter), [\mathcal{C}.Name\rangle = m \in \Delta'$.

If $a.Name = SEND$, by definition of φ, $esig(a.Action) = \langle a.Action.Name, [a.Action.Name\text{-}SORT] \parallel domain(a.Action.Parameter), [a.Action.Name\text{-}SORT] \rangle \in \Delta$. By Rule OMT-76 and OMT-77, and event theory $\mathbb{C}_{\mathbb{E}}$ is created and $\mathbb{C}_{\mathbb{E}}.Name = a.Action.Name$, by Rule OMT-79 $e = \langle a.Action.Name, [a.Action.Name] \parallel domain(a.Action), [a.Action.Name\text{-}SORT] \rangle \in \mathbb{C}_{\mathbb{E}}.Event$, and by Rule OMT-80 $a.Action.Name \in \mathbb{C}.Import$. Then by definition of Δ' in ω, $e \in \Delta'$ and $esig(a.Action) \in \Delta = \langle a.Action.Name, [a.Action.Name\text{-}SORT] \parallel domain(a.Action.Parameter), [a.Action.Name\text{-}SORT] \rangle = e \in \Delta'$. Therefore, $\Delta \subseteq \Delta'$.

An event theory can only be created by Rules OMT-76 through OMT-79, or Rule OMT-40; however, since the event theories defined in Rule OMT-40 are *broadcast* theories and are only referenced in aggregate specifications, an event theory whose name, $\mathbb{C}_{\mathbb{E}}.Name$ is found in the import block of some class specification $\mathbb{C}$ (as required by the definition of Δ' in ω), must have been created by Rules OMT-76 through OMT-79. So, if there exists an event theory $\mathbb{C}_{\mathbb{E}}$ such that $\mathbb{C}_{\mathbb{E}}.Name = n$ and $n \in \mathbb{C}.Import$ of some class specification $\mathbb{C}$, then it must be the case that there exists some $t \in \mathcal{C}.Transistion$ with an action $a \in t.Action$ such that $a.Name = SEND$, $a.Action.Name = n$, and $esig(a) = e \in \mathbb{C}_{\mathbb{E}}.Event$ (i.e., e is the event generated by a via Rule OMT-79). Therefore, by definition of φ, $esig(a) \in \Delta$, and by defintion of ω, $e \in \Delta'$.

Assume that there exists some $m \in \Delta'$ such that $m \in \mathbb{C}.Method$, $e \in \mathbb{C}.Event$, and an axiom $a \in \mathbb{C}.Axiom$ of the form

$$\begin{aligned}&\mathbb{C}.Name\text{-}STATE(x) = q \ldots \Rightarrow \\ &\quad \mathbb{C}.Name\text{-}STATE(event(\ldots)) = q_2 \ldots ATTR\text{-}EQUAL(e.Name(\ldots), m.Name(\ldots)) \ldots\end{aligned}$$

Since, (1) as defined in Section 6.2.7 a user may not modify a state attribute except through definition of a transition in the class dynamic model and (2) the assumption for this proof that a dynamic model for class C has been defined, an axiom of the form of a,

$$\text{“}\mathbb{C}.Name\text{-}STATE(x) = q \ldots \Rightarrow \mathbb{C}.Name\text{-}STATE(event(\ldots)) = q_2 \ldots\text{”}$$

must have been generated by Rule OMT-82. Therefore, by definition of the *method-invocation* function that implements Rule OMT-82, it must also be true that an axiom of the form of a with the substring

$$\text{“}ATTR\text{-}EQUAL(e(\ldots), m(\ldots))\text{”}$$

must have been generated from a transition $t \in \mathcal{C}.Transition$ such that $e = t.Name$ and $m = a.Name$ for some $a \in t.Action$ where $a.Name \neq SEND$. This same t and a also generated $m = \langle a.Name, [\mathcal{C}.Name] \parallel domain(a.Parameter), [\mathcal{C}.Name]\rangle \in \mathbb{C}.Method$ by OMT-75 and $msig(a) = \langle a.Name, [\mathcal{C}.Name] \| domain(a.Parameter), [\mathcal{C}.Name]\rangle \in \Delta$ by definition of φ. Thus $\Delta' \subseteq \Delta$ and, already knowing that $\Delta \subseteq \Delta'$, implies $\Delta = \Delta'$.

4. $\delta = \delta'$

To show that $\delta = \delta'$, I show that if I pick an object in any class $\mathcal{C}$ with a well-defined dynamic model such that the object is in some arbitrary state $q \in Q$ and receive input event $\sigma \in \Sigma$, $\delta(q, \sigma) = \delta'(q, \sigma)$.

First, pick some object in $\mathcal{C}$ in state $q \in Q$. Upon receipt of an input event $\sigma \in \Sigma$, if there exists a transition $t \in \mathcal{C}.Transition$ such that $t.FromState = q$, $t.Name = \sigma$, and $t.Axiom$ evaluates to true, then by φ, $\delta(q,\sigma) = t.ToState$. Otherwise, if there is no transition $t \in \mathcal{C}.Transition$ such that $t.FromState = q$, $t.Name = \sigma$, and $t.Axiom$ evaluates to true, then by φ, $\delta(q,\sigma) = q$.

By Rule OMT-82, for all $t \in \mathcal{C}.Transition$, t generates an axiom in $\mathbb{C}.Axiom$ of the form

$$\mathbb{C}.Name\text{-}STATE(x) = q \wedge guard \Rightarrow \mathbb{C}.Name\text{-}STATE(\sigma(...)) = q_2 \; ... \tag{F.1}$$

where $q = t.FromState$, the $guard$ is simply $t.Axiom$, and the $q_2 = t.ToState$. And, as shown above in Item 3, no other axioms of this form can be entered into $\mathbb{C}.Axiom$ using any other transformation rules.

Then, knowing that $Q = Q'$ and $\Sigma = \Sigma'$ from Items 1 and 2 above, I choose some object in class $\mathbb{C}$ in state $q \in Q$. Upon receipt of an input event $\sigma \in \Sigma$, if there exists an axiom in $\mathbb{C}.Axiom$ of the form shown in Equation F.1, such that the $guard$ ($t.Axiom$) evaluates to true then by ω, $\delta'(q,\sigma) = q_2 = t.ToState$. Otherwise, if there is no axiom of the appropriate form such that the $guard$ ($t.Axiom$) evaluates to true then by ω, $\delta'(q,\sigma) = q$.

Therefore, since for all possible inputs, (q,σ), $\delta(q,\sigma) = \delta'(q,\sigma)$, it must be the case that $\delta = \delta'$.

5. $\lambda = \lambda'$

To show that $\lambda = \lambda'$, I show that, for any class $\mathcal{C}$ with a well-defined dynamic model, if I pick an object in the class in any arbitrary state $q \in Q$ and receive input event $\sigma \in \Sigma$, $\lambda(q,\sigma) = \lambda'(q,\sigma)$.

First, choose some object in class $\mathcal{C}$ with state $q \in Q$. Upon receipt of an input event $\sigma \in \Sigma$, if there exists a transition $t \in \mathcal{C}.Transition$ such that $t.FromState = q$ and $t.Name = \sigma$, and $t.Axiom$ evaluates to true then by φ, $\lambda(q, \sigma) = sig(t.Action)$ which is the set of all action/event signatures invoked/sent in response to the transition as defined in Equation 7.79. Otherwise, if there is no transition $t \in C.Transition$ such that $t.FromState = q$ and $t.Name = \sigma$, and $t.Axiom$ evaluates to true then by φ, $\lambda(q, \sigma) = \{\}$.

By Rule OMT-82, for all $t \in C.Transition$, t generates an axiom in $ax \in \mathbb{C}.Axiom$ of the form given in Equation F.1 where $q = t.FromState$, the *guard* is $t.Axiom$, and the $q_2 = t.ToState$. The *method-invocations* function translates each non-send actions $a \in t.Action$ to a subsequence of the form

$$ATTR\text{-}EQUAL(t.Name(\ \dots\ ), a.Name(\ \dots\ ))\ \dots$$

and the function *event-sends* translates each send event $e \in t.Action$ to a subsequence of ax of the form

$$e.Action.Name\text{-}OBJ(t.Name(\dots)) = e.Action.Name(e.Action.Name\text{-}OBJ(x)\dots)$$

As discussed above in Item 3, only Rule OMT-82 is capable of placing axioms of this form into $\mathbb{C}.Axiom$.

Since $Q = Q'$ and $\Sigma = \Sigma'$ from Items 1 and 2 above, I may choose some object in class $\mathbb{C}$ in state $q \in Q$. Upon receipt of an input event $\sigma \in \Sigma$, if there exists an axiom, ax, in $\mathbb{C}.Axiom$ of the form shown in Equation F.1, such that the *guard* ($t.Axiom$) evaluates to true then by ω, $\lambda'(q, \sigma) = action\text{-}set(ax)$. By Rule OMT-82 all actions in t are translated into either a method invocation or an event send subsequence in ax. The function *action-set* as defined in Equation 7.80 returns the set of all method and event signatures for each method/event subsequence found in ax; therefore, $action\text{-}set(ax) = sig(t.Action)$. If, on the other hand, there are no axioms in $\mathbb{C}.Axiom$ of the form shown in Equation F.1, such that the *guard* ($t.Axiom$) evaluates to true then by ω, $\lambda'(q, \sigma) = \{\}$. Therefore, since for all possible inputs, (q, σ), $\lambda(q, \sigma) = \lambda'(q, \sigma)$, it must be true that $\lambda = \lambda'$.

6. $q_0 = q_0'$

The initial state is explicitly represented in each $\mathcal{G}$ dynamic model with a unique identifier *Initial-State-Marker*; however, by Rule OMT-83, a transition, t, from the initial state (by a *new* event) results in an axiom of the form

$$\begin{gathered} \mathit{C.Name\text{-}STATE(new\text{-}C.Name(...)} = \mathit{t.ToState} \wedge \\ \mathit{method\text{-}invocations}(t) \wedge \mathit{event\text{-}sends}(t) \end{gathered}$$

Therefore, there is no explicit representation of q_0. However, Q, δ', λ', and q_0' explicitly incorporate the *Initial-State-Marker* into ω and thus $q_0 = q_0'$.

Therefore, since $Q = Q'$, $\Sigma = \Sigma'$, $\Delta = \Delta'$, $\delta = \delta'$, $\lambda = \lambda'$ and $q_0 = q_0'$ it can be concluded that $M = M'$. $\square$

F.3 Functional Model Correctness Proof

In this section, Theorem VII.3 is proved.

Proof. Preservation of the functional model semantics by τ is established by showing the equivalence of two sets of functional model semantics, D and D', created from a generic OMT domain theory, $\mathcal{G}$. D is the functional model semantics defined by transforming $\mathcal{G}$ by φ while D' is the functional model semantics defined by transforming $\mathcal{G}$ by τ, into an O-SLANG domain theory $\mathbb{O}$, and then by ω. In this proof, I assume that $\mathcal{G}$ has a well defined functional model in which $\mathcal{C}$ is a class.

I prove the theorem by showing that, given a valid generic OMT domain theory $\mathcal{G}$, each component defined in D (C, F, K, and R) exist in D' and that each component defined in D' exists in D.

1. $C = C'$.

I show $C = C'$ by showing that $C \subseteq C'$ and then showing $C' \subseteq C'$.

- $C \subseteq C'$.

  For any $c \in C$, by the definition of C, the name of c must be the name of a process of $\mathcal{C}$, a datastore of $\mathcal{C}$, or the symbol *Extern*. If c refers to a process in $\mathcal{C}$ then by Rule OMT-85 or Rule OMT-86 the process referred to by c is mapped to either a method or an operation in $\mathbb{C}$ and therefore, by definition of C', $c \in C'$.

  If c is the name of a datastore then by definition of a valid dataflow diagram, there must be a dataflow in $\mathcal{C}$.DataFlow such that it is either the target or source. Also, by Assumptions V.12 and V.13 there exists a process, p, such that p only accesses or modifies datastore c. Therefore, by Rule OMT-87 there must be an axiom in $\mathbb{C}$.Axiom such that the following is a substring.

  "... *op*(*datastore*(X) ...) ..."

  This substring is extracted by function *datastores* as defined in Equation 7.83 and thus, by definition of C', $c \in C'$.

  If c is *Extern* then by definition of C', $c \in C'$. Therefore $C \subseteq C'$.

- $C' \subseteq C$.

  Now, assume $c \in C'$. If $c \in C'$ then by the definition of C', $c \in \mathbb{C}.Method$, $c \in \mathbb{C}.Operation$, $c \in datastores(\mathbb{C})$, or $c = Extern$. Since, by assumption, all methods in $\mathbb{C}$ are defined from processes in $\mathcal{C}$ via Rule OMT-85 there must be a process in $\mathcal{C}$ whose name is c and, by definition of C, $c \in C$.

  If c is the name of an operation in $\mathbb{C}$ then by the assumption that all methods in $\mathbb{C}$ are defined from processes in $\mathcal{C}$ via Rule OMT-86 there must be a process in $\mathcal{C}$ whose name is c and thus, by definition of C, $c \in C$.

  If c is a datastore, then $c \in datastores(ax)$ for some $ax \in \mathbb{C}.Axiom$ that contains the following substring.

  "... *op*(c(X) ...) ..."

Since axioms of this form are only generated by Rule OMT-87, then c must have been in $datastore(\mathcal{C})$ that, by definition of C, implies $c \in C$.

If c is $Extern$ then by definition of C, $c \in C$ and $C' \subseteq C$. Therefore since $C' \subseteq C$ and $C \subseteq C'$, $C = C'$.

2. $F = F'$.

I show that $F = F'$ by showing that for any $\mathcal{C}$ in $\mathcal{G}$, for each f *in* $\mathcal{C}.Dataflow$, $f \in F$ and $f \in F'$ and nothing else is in F or F'.

If $f \in \mathcal{C}.Dataflow$, then trivially, by the definition of F, $f \in F$ and nothing else can be in F.

If $f \in \mathcal{C}.Dataflow$, it is a dataflow in a functional model. Given a valid dataflow diagram, as shown in Table 7.1, these dataflows can be (a) an input to a process (lines 1, 2, 10, and 14), (b) an output to a datastore (line 11), or (c) an output from a process to $Extern$ (lines 8 and 12). I discuss each of these possibilities below.

(a) There are two unique cases to consider when a dataflow is an input to a process. The input may be to a top-level process (line 1 in Table 7.1) or a subprocess(lines 2, 10, and 14 in Table 7.1). Each of these is discussed below.

If $f = \langle f.Name, f.Type, f.Target, f.Source \rangle$ is an input to a process, p, where $p.Name = f.Target$ and $\langle f.Name, f.Type \rangle =\in p.InFlows$, then by Rule OMT-85 or Rule OMT-86 there is an operation or method defined in $\mathbb{C}$ as shown below.

$$\langle f.Target, [...f.Type...], [...] \rangle \in (\mathbb{C}.Method \cup \mathbb{C}.Operation)$$

Thus by Rule OMT-87 there exists an axiom, ax, in $\mathbb{C}.Axiom$ of the form

$$ax = \text{“}m(i_1...i_m) = o_1...o_n... \wedge r_1...r_o = sp(d_1...d_p)...\text{”}$$

such that $m = f.Target$, or, $sp = f.Target$.

i. If $m = f.Target$, then some parameter $i_1...i_m = f.Name$ such that the variables in the function *dataflows-of(ax)* take on the following values.

$$\begin{aligned} Extern &= f.Source \\ t.Name &= f.Target \\ p_3 &= f.Name \\ itype(p_3, t) &= f.Type \end{aligned}$$

Thus by the defintions of *dataflows-of* and F', $\langle p_3, itype(p_3, t), Extern, t.Name\rangle = \langle f.Name, f.Type, f.Source, f.Target\rangle \in F'$

ii. If $sp = f.Target$, then some parameter in $d_1 ... d_m = f.Name$ such that the variables in the function *dataflows-of(ax)* take on the following values.

$$\begin{aligned} op_2.Name &= f.Source \\ op_1.Name &= f.Target \\ p_1 &= f.Name \\ itype(p_1, op_1) &= f.Type \end{aligned}$$

Therefore, by the defintions of *dataflows-of* and F', $\langle p_1, itype(p_1, op_1), op_2.Name, op_1.Name\rangle = \langle f.Name, f.Type, f.Source, f.Target\rangle \in F'$

(b) If $f = \langle f.Name, f.Type, f.Target, f.Source\rangle$ is an output to a datastore, d, then by Assumptions V.12 and V.13 there exists a process p such that f is the only datastore output of p. Therefore, p is a method such that $p.Name = f.Source$ and by Rule OMT-85 there is a method defined in $\mathbb{C}$ as shown below.

$$\langle f.Source, [f.Target...], [f.Target]\rangle \in \mathbb{C}.Method$$

Thus by Rule OMT-87 there exists an axiom, ax, in $\mathbb{C}.Axiom$ of the form

$$ax = \text{“}m(i_1 ... i_m) = o_1 ... o_n ... \wedge r_1 ... r_n = sp(d_1 ... d_p) ...\text{”}$$

such that $f.Source = sp$ and $r_1 ... r_n = r_1 = f.Target$ and the variables in the function *dataflows-of(ax)* take on the following values.

$$\begin{aligned} op_1.Name &= f.Source \\ o &= f.Target \\ dsname(o) &= f.Name \\ dstype(o) &= f.Type \end{aligned}$$

Therefore, by the defintions of *dataflows-of* and F', $\langle dsname(o),\ dstype(o),\ op_1.Name,\ o\rangle = \langle f.Name,\ f.Type,\ f.Source,\ f.Target\rangle \in F'$

(c) There are two unique cases to consider when an output goes to *Extern*. The output may come from a subprocess, or it may come from a top-level object. Each of these is discussed separately.

i. If $f = \langle f.Name, f.Type, f.Target, f.Source\rangle$ is an output from a subprocess p, with $p.Name = f.Source$, to an external object such that $f.Target = Extern$, then p must be an operation since the only output from a method is the class sort which is implied and not actually part of the dataflow diagram. Thus by Rule OMT-86 there is an operation defined in $\mathbb{C}$ as shown below.

$$\langle f.Source, [...], [...f.Type...]\rangle \in \mathbb{C}.Operation$$

Thus by Rule OMT-87 there exists an axiom, ax, in $\mathbb{C}.Axiom$ of the form

$$ax = \text{“}m(i_1...i_m) = o_1...o_n... \wedge r_1...r_o = sp(d_1...d_p)...\text{”}$$

such that $sp = f.Source$ and some output of sp, $r_1...r_m = f.Name$ and the variables in the function *dataflows-of(ax)* take on the following values.

$$\begin{aligned} Extern &= f.Target \\ op_1.Name &= f.Source \\ o &= f.name \\ otype(o, op_1) &= f.Type \end{aligned}$$

Then by the defintions of *dataflows-of* and F', $\langle o,\ otype(o, sp),\ op_1.Name,\ Extern\rangle = \langle f.Name,\ f.Type,\ f.Source,\ f.Target\rangle \in F'$

ii. If $f = \langle f.Name, f.Type, f.Target, f.Source\rangle$ is an output from a top-level process p, where $p.Name = f.Source$, then p must be an operation since the only output from a method is the class sort which is implied and not actually part of the dataflow diagram. Thus by Rule OMT-86 there is an operation defined in $\mathbb{C}$ as shown below.

$$\langle f.Source, [...], [...f.Type...]\rangle \in \mathbb{C}.Operation$$

Thus by Rule OMT-87 there exists an axiom, ax, in $\mathbb{C}.Axiom$ of the form

$$ax = \text{``}m(i_1...i_m) = o_1...o_n... \wedge r_1...r_o = sp(d_1...d_p)...\text{''}$$

such that $m = f.Source$ and some output of m, $o_1...o_m = f.Name$ and the variables in the function *dataflows-of(ax)* take on the following values.

$$\begin{aligned} Extern &= f.Target \\ t.Name &= f.Source \\ p_3 &= f.name \\ otype(p_3, t) &= f.Type \end{aligned}$$

Thus by the defintions of *dataflows-of* and F', $\langle p_3, otype(p_3, t), t.Name, Extern\rangle = \langle f.Name, f.Type, f.Source, f.Target\rangle \in F'$

Therefore, since if $f \in \mathcal{C}.Dataflow$ is either (a) an input to a process, (b) an output to a datastore, or (c) an output from the top-level process of a dataflow diagram, then $f \in F'$. Also, given the initial assumption that all processes p in $\mathcal{C}$ are processes in a functional model defined in $\mathcal{C}$, all axioms produced by OMT-87 use only processes from a functional model of $\mathcal{C}$. Since OMT-87 is the only rule capable of producing axioms in $\mathbb{C}.Axiom$ of the form required by the function *dataflows-of*, all sources and targets extracted from some axiom in $\mathbb{C}.Axiom$ must be processes in a functional model of $\mathcal{C}$. Finally, given the uniqueness of dataflow names by Assumption V.19, the fact that there must be in inflow and an outflow for each input and output of a process in a valid dataflow diagram (by Assumption V.18), and the definition of OMT-87, it is clear that there are no dataflows in F' not derived from a dataflows in a $\mathcal{C}$ functional model.

3. $K = K'$.

   Since I showed in Item 1 that $C = C'$ and by definition their definitions, each includes the symbol *Extern*, it is obvious that $C \setminus \{Extern\} = C' \setminus \{Extern\}$ and thus $K = K'$.

4. $R = R'$.

   Since the definition of R uses the set *dfmerge*($\mathcal{C}$.*DataFlow*), which is the definition F, its definition is equivalent to Equation F.2.

$$\begin{aligned}\{\langle x, y\rangle \mid\ & (x, y \in F \land\ x.Target = y.Source \land x.Target \neq Extern) \\ & \lor (\langle x, z\rangle \in R \land \langle z, y\rangle \in R)\}\end{aligned} \tag{F.2}$$

   And, since the defintion of R' uses the set *dfmerge*($\{f \mid a \in \mathbb{C}.Axiom \land f \in dataflows\text{-}of(a)\}$, which is the definition of F', its definition is equivalent to Equation F.3.

$$\begin{aligned}\{\langle x, y\rangle \mid\ & (x, y \in F') \\ & \land\ x.Target = y.Source \land\ x.Target \neq Extern) \\ & \lor (\langle x, z\rangle \in R' \land \langle z, y\rangle \in R')\}\end{aligned} \tag{F.3}$$

   And finally, since $F = F'$, Equations F.2 and F.3 are equivalent and thus $R = R'$.

Since, as shown above, $C = C'$, $F = F'$, $K = K$, and $R = R'$, then $D = D'$ and therefore, the translation τ preserves the semantics of the generic OMT functional model. □

F.4 Summary

This appendix presents the proofs of Theorems VII.1, VII.2, and VII.3. These theorems show that the transformation rules as defined in Chapter VII preserve the semantics of the the object model, the dynamic model, and the functional model as defined Chapter V.

Appendix G. Feasibility Demonstration O-SLANG *Output*

G.1 Pump O-SLANG

```
class CLUTCH is
 class-sort CLUTCH
 import START-FUEL
 sort CLUTCH-STATE
 attributes START-FUEL-OBJ: CLUTCH -> START-FUEL-SORT
 state-attributes CLUTCH-STATE: CLUTCH -> CLUTCH-STATE
 methods CREATE-CLUTCH: -> CLUTCH
 states
   CLUTCH-DISABLED: -> CLUTCH-STATE
   CLUTCH-FREE: -> CLUTCH-STATE
   CLUTCH-ENGAGED: -> CLUTCH-STATE
 events
   FREE-CLUTCH: CLUTCH -> CLUTCH
   DISABLE-CLUTCH: CLUTCH -> CLUTCH
   ENGAGE-CLUTCH: CLUTCH -> CLUTCH
   NEW-CLUTCH: -> CLUTCH
 axioms
   CLUTCH-DISABLED <> CLUTCH-FREE;
   CLUTCH-DISABLED <> CLUTCH-ENGAGED;
   CLUTCH-FREE <> CLUTCH-ENGAGED;
   ATTR-EQUAL(C1, C2) <=> (START-FUEL-OBJ(C1) = START-FUEL-OBJ(C2));
   (CLUTCH-STATE(NEW-CLUTCH(C)) = CLUTCH-DISABLED
                  & ATTR-EQUAL(NEW-CLUTCH(C), CREATE-CLUTCH(C)));
   (CLUTCH-STATE(C) = CLUTCH-FREE) =>(CLUTCH-STATE(ENGAGE-CLUTCH(C)) = CLUTCH-ENGAGED
        & START-FUEL-OBJ(ENGAGE-CLUTCH(C)) = START-FUEL(START-FUEL-OBJ(C)));
   (CLUTCH-STATE(C) = CLUTCH-FREE) =>(CLUTCH-STATE(DISABLE-CLUTCH(C)) = CLUTCH-DISABLED);
   (CLUTCH-STATE(C) = CLUTCH-FREE) =>(CLUTCH-STATE(FREE-CLUTCH(C)) = CLUTCH-ENGAGED);
   (CLUTCH-STATE(C) = CLUTCH-DISABLED) =>(CLUTCH-STATE(FREE-CLUTCH(C)) = CLUTCH-FREE);
   CLUTCH-STATE(C) = CLUTCH-DISABLED => CLUTCH-STATE(ENGAGE-CLUTCH(C)) = CLUTCH-DISABLED;
   CLUTCH-STATE(C) = CLUTCH-DISABLED => CLUTCH-STATE(DISABLE-CLUTCH(C)) = CLUTCH-DISABLED;
   CLUTCH-STATE(C) = CLUTCH-DISABLED => CLUTCH-STATE(FREE-CLUTCH(C)) = CLUTCH-DISABLED;
   CLUTCH-STATE(C) = CLUTCH-FREE => CLUTCH-STATE(FREE-CLUTCH(C)) = CLUTCH-FREE;
   CLUTCH-STATE(C) = CLUTCH-ENGAGED => CLUTCH-STATE(ENGAGE-CLUTCH(C)) = CLUTCH-ENGAGED;
   CLUTCH-STATE(C) = CLUTCH-ENGAGED => CLUTCH-STATE(DISABLE-CLUTCH(C)) = CLUTCH-ENGAGED;
   CLUTCH-STATE(C) = CLUTCH-ENGAGED => CLUTCH-STATE(FREE-CLUTCH(C)) = CLUTCH-ENGAGED;
   CLUTCH-STATE(C) = CLUTCH-ENGAGED => CLUTCH-STATE(FREE-CLUTCH(C)) = CLUTCH-ENGAGED
end-class


class CLUTCH-CLASS is
 class-sort CLUTCH-CLASS
 contained-class CLUTCH
 events
   FREE-CLUTCH: CLUTCH-CLASS -> CLUTCH-CLASS
   DISABLE-CLUTCH: CLUTCH-CLASS -> CLUTCH-CLASS
   ENGAGE-CLUTCH: CLUTCH-CLASS -> CLUTCH-CLASS
   NEW-CLUTCH-CLASS: -> CLUTCH-CLASS
 axioms
   NEW-CLUTCH-CLASS() = EMPTY-SET;
   fa (C:CLUTCH, CC:CLUTCH-CLASS) in(C, CC) <=> in(ENGAGE-CLUTCH(C), ENGAGE-CLUTCH(CC));
   fa (C:CLUTCH, CC:CLUTCH-CLASS) in(C, CC) <=> in(DISABLE-CLUTCH(C), DISABLE-CLUTCH(CC));
   fa (C:CLUTCH, CC:CLUTCH-CLASS) in(C, CC) <=> in(FREE-CLUTCH(C), FREE-CLUTCH(CC));
   fa (C:CLUTCH, CC:CLUTCH-CLASS) in(C, CC) <=> in(FREE-CLUTCH(C), FREE-CLUTCH(CC))
 end-class


 class HOLSTER is
  class-sort HOLSTER
  sort HOLSTER-STATE
  state-attributes HOLSTER-STATE: HOLSTER -> HOLSTER-STATE
  methods CREATE-HOLSTER: -> HOLSTER
  states
    HOLSTER-WAIT: -> HOLSTER-STATE
    HOLSTER-WORKING: -> HOLSTER-STATE
  events
    CLOSE-HOLSTER-SWITCH: HOLSTER -> HOLSTER
    RELEASE-HOLSTER-SWITCH: HOLSTER -> HOLSTER
    NEW-HOLSTER: -> HOLSTER
  axioms
    HOLSTER-WAIT <> HOLSTER-WORKING;
    (HOLSTER-STATE(NEW-HOLSTER(H)) = HOLSTER-WAIT
                 & ATTR-EQUAL(NEW-HOLSTER(H), CREATE-HOLSTER(H)));
    (HOLSTER-STATE(H) = HOLSTER-WAIT)
         => (HOLSTER-STATE(RELEASE-HOLSTER-SWITCH(H)) = HOLSTER-WORKING);
    (HOLSTER-STATE(H) = HOLSTER-WORKING)
         => (HOLSTER-STATE(CLOSE-HOLSTER-SWITCH(H)) = HOLSTER-WAIT);
    HOLSTER-STATE(H) = HOLSTER-WAIT
         => HOLSTER-STATE(CLOSE-HOLSTER-SWITCH(H)) = HOLSTER-WAIT;
    HOLSTER-STATE(H) = HOLSTER-WORKING
         => HOLSTER-STATE(RELEASE-HOLSTER-SWITCH(H)) = HOLSTER-WORKING
 end-class


 class HOLSTER-CLASS is
  class-sort HOLSTER-CLASS
  contained-class HOLSTER
  events
    CLOSE-HOLSTER-SWITCH: HOLSTER-CLASS -> HOLSTER-CLASS
    RELEASE-HOLSTER-SWITCH: HOLSTER-CLASS -> HOLSTER-CLASS
    NEW-HOLSTER-CLASS: -> HOLSTER-CLASS
  axioms
    NEW-HOLSTER-CLASS() = EMPTY-SET;
    fa ((H: HOLSTER), HC: HOLSTER-CLASS) in(H, HC)
      <=> in(RELEASE-HOLSTER-SWITCH(H), RELEASE-HOLSTER-SWITCH(HC));
    fa ((H: HOLSTER), HC: HOLSTER-CLASS) in(H, HC)
      <=> in(CLOSE-HOLSTER-SWITCH(H), CLOSE-HOLSTER-SWITCH(HC))
 end-class


 class MOTOR is
  class-sort MOTOR
  import FREE-CLUTCH, DISABLE-CLUTCH
  sort MOTOR-STATE
  attributes
    DISABLE-CLUTCH-OBJ: MOTOR -> DISABLE-CLUTCH-SORT
    FREE-CLUTCH-OBJ: MOTOR -> FREE-CLUTCH-SORT
  state-attributes
    MOTOR-STATE: MOTOR -> MOTOR-STATE
 methods
   CREATE-MOTOR: -> MOTOR
 states
   MOTOR-DISABLED: -> MOTOR-STATE
   MOTOR-RUNNING: -> MOTOR-STATE
 events
   STOP-MOTOR: MOTOR -> MOTOR
   START-PUMP-MOTOR: MOTOR -> MOTOR
   NEW-MOTOR: -> MOTOR
 axioms
   MOTOR-DISABLED <> MOTOR-RUNNING;
   ATTR-EQUAL(M1, M2) <=> (DISABLE-CLUTCH-OBJ(M1) = DISABLE-CLUTCH-OBJ(M2)
                       & FREE-CLUTCH-OBJ(M1) = FREE-CLUTCH-OBJ(M2));
   (MOTOR-STATE(NEW-MOTOR(M)) = MOTOR-DISABLED
             & ATTR-EQUAL(NEW-MOTOR(M), CREATE-MOTOR(M)));
   (MOTOR-STATE(M) = MOTOR-DISABLED)
       => (MOTOR-STATE(START-PUMP-MOTOR(M)) = MOTOR-RUNNING
             & FREE-CLUTCH-OBJ(START-PUMP-MOTOR(M))
                     = FREE-CLUTCH(FREE-CLUTCH-OBJ(M)));
   (MOTOR-STATE(M) = MOTOR-RUNNING)
       => (MOTOR-STATE(STOP-MOTOR(M)) = MOTOR-DISABLED
             & DISABLE-CLUTCH-OBJ(STOP-MOTOR(M))
                     = DISABLE-CLUTCH(DISABLE-CLUTCH-OBJ(M)));
   MOTOR-STATE(M) = MOTOR-DISABLED
       => MOTOR-STATE(STOP-MOTOR(M)) = MOTOR-DISABLED;
   MOTOR-STATE(M) = MOTOR-RUNNING
       => MOTOR-STATE(START-PUMP-MOTOR(M)) = MOTOR-RUNNING
end-class


class MOTOR-CLASS is
 class-sort MOTOR-CLASS
   contained-class MOTOR
 events
   STOP-MOTOR: MOTOR-CLASS -> MOTOR-CLASS
   START-PUMP-MOTOR: MOTOR-CLASS -> MOTOR-CLASS
   NEW-MOTOR-CLASS: -> MOTOR-CLASS
 axioms
   NEW-MOTOR-CLASS() = EMPTY-SET;
   fa (M:MOTOR, MC:MOTOR-CLASS) in(M, MC)
            <=> in(START-PUMP-MOTOR(M), START-PUMP-MOTOR(MC));
   fa (M:MOTOR, MC:MOTOR-CLASS) in(M, MC) <=> in(STOP-MOTOR(M), STOP-MOTOR(MC))
end-class


class GUN is
 class-sort GUN
 import START-TIMER, DISABLE-PUMP, CLOSE-HOLSTER-SWITCH,
        FREE-CLUTCH, ENGAGE-CLUTCH, RELEASE-HOLSTER-SWITCH
 sort GUN-STATE
 attributes
   RELEASE-HOLSTER-SWITCH-OBJ:
   GUN -> RELEASE-HOLSTER-SWITCH-SORT
   ENGAGE-CLUTCH-OBJ: GUN -> ENGAGE-CLUTCH-SORT
   FREE-CLUTCH-OBJ: GUN -> FREE-CLUTCH-SORT
   CLOSE-HOLSTER-SWITCH-OBJ: GUN -> CLOSE-HOLSTER-SWITCH-SORT
```

```
    DISABLE-PUMP-OBJ: GUN -> DISABLE-PUMP-SORT
    START-TIMER-OBJ: GUN -> START-TIMER-SORT
  state-attributes
    GUN-STATE: GUN -> GUN-STATE
  methods
    CREATE-GUN: -> GUN
  states
    GUN-DISABLED: -> GUN-STATE
    GUN-ENABLED: -> GUN-STATE
    GUN-ON: -> GUN-STATE
  events
    REMOVE-GUN: GUN -> GUN
    RELEASE-TRIGGER: GUN -> GUN
    DEPRESS-TRIGGER: GUN -> GUN
    CUT-OFF-SUPPLY: GUN -> GUN
    REPLACE-GUN: GUN -> GUN
    NEW-GUN: -> GUN
  axioms
    GUN-DISABLED <> GUN-ENABLED;
    GUN-DISABLED <> GUN-ON;
    GUN-ENABLED <> GUN-ON;
    ATTR-EQUAL(G1, G2) <=> (RELEASE-HOLSTER-SWITCH-OBJ(G1) = RELEASE-HOLSTER-SWITCH-OBJ(G2)
                            & ENGAGE-CLUTCH-OBJ(G1) = ENGAGE-CLUTCH-OBJ(G2)
                            & FREE-CLUTCH-OBJ(G1) = FREE-CLUTCH-OBJ(G2)
                            & CLOSE-HOLSTER-SWITCH-OBJ(G1) = CLOSE-HOLSTER-SWITCH-OBJ(G2)
                            & DISABLE-PUMP-OBJ(G1) = DISABLE-PUMP-OBJ(G2)
                            & START-TIMER-OBJ(G1) = START-TIMER-OBJ(G2));
    (GUN-STATE(NEW-GUN(G)) = GUN-DISABLED & ATTR-EQUAL(NEW-GUN(G), CREATE-GUN(G)));

    (GUN-STATE(G) = GUN-ENABLED) => (GUN-STATE(REPLACE-GUN(G)) = GUN-DISABLED
                      & START-TIMER-OBJ(REPLACE-GUN(G)) = START-TIMER(START-TIMER-OBJ(G))
                      & DISABLE-PUMP-OBJ(REPLACE-GUN(G)) = DISABLE-PUMP(DISABLE-PUMP-OBJ(G))
                      & CLOSE-HOLSTER-SWITCH-OBJ(REPLACE-GUN(G))
                                 = CLOSE-HOLSTER-SWITCH(CLOSE-HOLSTER-SWITCH-OBJ(G)));

    (GUN-STATE(G) = GUN-ON) => (GUN-STATE(CUT-OFF-SUPPLY(G)) = GUN-ENABLED
                  & FREE-CLUTCH-OBJ(CUT-OFF-SUPPLY(G)) = FREE-CLUTCH(FREE-CLUTCH-OBJ(G)));
    (GUN-STATE(G) = GUN-ENABLED) => (GUN-STATE(DEPRESS-TRIGGER(G)) = GUN-ON
                  & ENGAGE-CLUTCH-OBJ(DEPRESS-TRIGGER(G))
                                       = ENGAGE-CLUTCH(ENGAGE-CLUTCH-OBJ(G)));
    (GUN-STATE(G) = GUN-ENABLED) => (GUN-STATE(RELEASE-TRIGGER(G)) = GUN-ON
                  & FREE-CLUTCH-OBJ(RELEASE-TRIGGER(G)) = FREE-CLUTCH(FREE-CLUTCH-OBJ(G)));
    (GUN-STATE(G) = GUN-DISABLED) => (GUN-STATE(REMOVE-GUN(G)) = GUN-ENABLED
                  & RELEASE-HOLSTER-SWITCH-OBJ(REMOVE-GUN(G))
                                  = RELEASE-HOLSTER-SWITCH(RELEASE-HOLSTER-SWITCH-OBJ(G)));
    GUN-STATE(G) = GUN-DISABLED => GUN-STATE(REPLACE-GUN(G)) = GUN-DISABLED;
    GUN-STATE(G) = GUN-DISABLED => GUN-STATE(CUT-OFF-SUPPLY(G)) = GUN-DISABLED;
    GUN-STATE(G) = GUN-DISABLED => GUN-STATE(DEPRESS-TRIGGER(G)) = GUN-DISABLED;
    GUN-STATE(G) = GUN-DISABLED => GUN-STATE(RELEASE-TRIGGER(G)) = GUN-DISABLED;
    GUN-STATE(G) = GUN-ENABLED => GUN-STATE(CUT-OFF-SUPPLY(G)) = GUN-ENABLED;
    GUN-STATE(G) = GUN-ENABLED => GUN-STATE(REMOVE-GUN(G)) = GUN-ENABLED;
    GUN-STATE(G) = GUN-ON => GUN-STATE(REPLACE-GUN(G)) = GUN-ON;
    GUN-STATE(G) = GUN-ON => GUN-STATE(DEPRESS-TRIGGER(G)) = GUN-ON;
    GUN-STATE(G) = GUN-ON => GUN-STATE(RELEASE-TRIGGER(G)) = GUN-ON;
    GUN-STATE(G) = GUN-ON => GUN-STATE(REMOVE-GUN(G)) = GUN-ON
end-class
```

```
class GUN-CLASS is
 class-sort GUN-CLASS contained-class GUN
 events
   REMOVE-GUN: GUN-CLASS -> GUN-CLASS
   RELEASE-TRIGGER: GUN-CLASS -> GUN-CLASS
   DEPRESS-TRIGGER: GUN-CLASS -> GUN-CLASS
   CUT-OFF-SUPPLY: GUN-CLASS -> GUN-CLASS
   REPLACE-GUN: GUN-CLASS -> GUN-CLASS
   NEW-GUN-CLASS: -> GUN-CLASS
 axioms
   NEW-GUN-CLASS() = EMPTY-SET;
   fa ((G: GUN), GC: GUN-CLASS) in(G, GC) <=> in(REPLACE-GUN(G), REPLACE-GUN(GC));
   fa ((G: GUN), GC: GUN-CLASS) in(G, GC) <=> in(CUT-OFF-SUPPLY(G), CUT-OFF-SUPPLY(GC));
   fa ((G: GUN), GC: GUN-CLASS) in(G, GC) <=> in(DEPRESS-TRIGGER(G), DEPRESS-TRIGGER(GC));
   fa ((G: GUN), GC: GUN-CLASS) in(G, GC) <=> in(RELEASE-TRIGGER(G), RELEASE-TRIGGER(GC));
   fa ((G: GUN), GC: GUN-CLASS) in(G, GC) <=> in(REMOVE-GUN(G), REMOVE-GUN(GC))
end-class


class DISPLAY is
 class-sort DISPLAY
 import GRADE, AMOUNT, VOLUME
 sort DISPLAY-STATE
 operations ATTR-EQUAL: DISPLAY, DISPLAY -> BOOLEAN
 attributes
   COST: DISPLAY -> AMOUNT
   VOLUME: DISPLAY -> VOLUME
   PPG: DISPLAY -> AMOUNT
   GRADE: DISPLAY -> GRADE
 state-attributes DISPLAY-STATE: DISPLAY -> DISPLAY-STATE
 methods
   CREATE-DISPLAY: -> DISPLAY
   UPDATE-DISPLAY: DISPLAY, COST, VOLUME -> DISPLAY
   ZERO-OUT-DISPLAY: DISPLAY -> DISPLAY
 states
   ZERO-DISPLAY: -> DISPLAY-STATE
   INCREMENT-DISPLAY: -> DISPLAY-STATE
 events
   RESET-DISPLAY: DISPLAY -> DISPLAY
   PULSE: DISPLAY -> DISPLAY
   NEW-DISPLAY: -> DISPLAY
 axioms
   ZERO-DISPLAY <> INCREMENT-DISPLAY;
   ATTR-EQUAL(D1, D2) <=> (GRADE(D1) = GRADE(D2) & VOLUME(D1) = VOLUME(D2)
                           & COST(D1) = COST(D2));
   DISPLAY-STATE(D) = ZERO-DISPLAY => COST(D) = 0 & VOLUME(D) = 0;
   DISPLAY-STATE(D) = INCREMENT-DISPLAY => COST(D) >= 0 & VOLUME(D) >= 0;
   COST(D) >= 0;
   VOLUME(D) >= 0;
   PPG(D) = COST(D) / VOLUME(D);
   VOLUME(CREATE-DISPLAY(D)) = 0;
   COST(CREATE-DISPLAY(D)) = 0;
   GRADE(UPDATE-DISPLAY(D)) = GRADE(D);
   VOLUME(UPDATE-DISPLAY(D)) = VOLUME(D) + 1;
   COST(UPDATE-DISPLAY(D)) = COST(D) + 1;
   GRADE(ZERO-OUT-DISPLAY(D)) = GRADE(D);
   COST(ZERO-OUT-DISPLAY(D)) = 0;
   VOLUME(ZERO-OUT-DISPLAY(D)) = 0;
   (DISPLAY-STATE(NEW-DISPLAY(D)) = ZERO-DISPLAY
             & ATTR-EQUAL(NEW-DISPLAY(D), CREATE-DISPLAY(D)));
   (DISPLAY-STATE(D) = ZERO-DISPLAY) => (DISPLAY-STATE(PULSE(D)) = INCREMENT-DISPLAY
                     & ATTR-EQUAL(PULSE(D), UPDATE-DISPLAY(D)));
   (DISPLAY-STATE(D) = INCREMENT-DISPLAY)
       => (DISPLAY-STATE(RESET-DISPLAY(D)) = ZERO-DISPLAY
                     & ATTR-EQUAL(RESET-DISPLAY(D), ZERO-OUT-DISPLAY(D)));
   (DISPLAY-STATE(D) = INCREMENT-DISPLAY) => (DISPLAY-STATE(PULSE(D)) = INCREMENT-DISPLAY
                     & ATTR-EQUAL(PULSE(D), UPDATE-DISPLAY(D)));
   DISPLAY-STATE(D) = ZERO-DISPLAY => DISPLAY-STATE(RESET-DISPLAY(D)) = ZERO-DISPLAY;
   DISPLAY-STATE(D) = ZERO-DISPLAY => DISPLAY-STATE(PULSE(D)) = ZERO-DISPLAY;
   DISPLAY-STATE(D) = INCREMENT-DISPLAY => DISPLAY-STATE(PULSE(D)) = INCREMENT-DISPLAY;
end-class


class DISPLAY-CLASS is
 class-sort DISPLAY-CLASS
 contained-class DISPLAY
 events
   RESET-DISPLAY: DISPLAY-CLASS -> DISPLAY-CLASS
   PULSE: DISPLAY-CLASS -> DISPLAY-CLASS
   NEW-DISPLAY-CLASS: -> DISPLAY-CLASS
 axioms
   NEW-DISPLAY-CLASS() = EMPTY-SET;
   fa (D:DISPLAY, DC:DISPLAY-CLASS) in(D, DC) <=> in(PULSE(D), PULSE(DC));
   fa (D:DISPLAY, DC:DISPLAY-CLASS) in(D, DC) <=> in(RESET-DISPLAY(D), RESET-DISPLAY(DC));
end-class


class CLUTCH-MOTOR-ASSEMBLY is
 class-sort CLUTCH-MOTOR-ASSEMBLY
 import CLUTCH-MOTOR-ASSEMBLY-AGGREGATE
 attributes
   MOTOR-OBJ: CLUTCH-MOTOR-ASSEMBLY -> MOTOR-CLASS
   CLUTCH-OBJ: CLUTCH-MOTOR-ASSEMBLY -> CLUTCH-CLASS
 methods
   CREATE-CLUTCH-MOTOR-ASSEMBLY: -> CLUTCH-MOTOR-ASSEMBLY
 events NEW-CLUTCH-MOTOR-ASSEMBLY: -> CLUTCH-MOTOR-ASSEMBLY
 axioms
   ATTR-EQUAL(C1, C2) <=> (CLUTCH-OBJ(C1) = CLUTCH-OBJ(C2)
                          & MOTOR-OBJ(C1) = MOTOR-OBJ(C2));
   ATTR-EQUAL(NEW-CLUTCH-MOTOR-ASSEMBLY(), CREATE-CLUTCH-MOTOR-ASSEMBLY());
   SIZE(CLUTCH-OBJ(C)) = 1;
   SIZE(MOTOR-OBJ(C)) = 1
end-class


class CLUTCH-MOTOR-ASSEMBLY-CLASS is
 class-sort CLUTCH-MOTOR-ASSEMBLY-CLASS
 contained-class CLUTCH-MOTOR-ASSEMBLY
 events
   NEW-CLUTCH-MOTOR-ASSEMBLY-CLASS: -> CLUTCH-MOTOR-ASSEMBLY-CLASS
 axioms NEW-CLUTCH-MOTOR-ASSEMBLY-CLASS() = EMPTY-SET
end-class


aggregate CLUTCH-MOTOR-ASSEMBLY-AGGREGATE is
 nodes DISABLE-CLUTCH, FREE-CLUTCH, MOTOR-CLASS, MOTOR, START-FUEL,
       CLUTCH-CLASS, CLUTCH
 arcs DISABLE-CLUTCH -> CLUTCH: {DISABLE-CLUTCH-SORT -> CLUTCH},
      FREE-CLUTCH -> CLUTCH: {FREE-CLUTCH-SORT -> CLUTCH},
      MOTOR -> MOTOR-CLASS: {}, CLUTCH -> CLUTCH-CLASS: {},
      DISABLE-CLUTCH -> MOTOR: {}, FREE-CLUTCH -> MOTOR: {},
      START-FUEL -> CLUTCH: {}
end-aggregate


class GUN-HOLSTER-ASSEMBLY is
 class-sort GUN-HOLSTER-ASSEMBLY
 import GUN-HOLSTER-ASSEMBLY-AGGREGATE
 attributes
   GUN-OBJ: GUN-HOLSTER-ASSEMBLY -> GUN-CLASS
   HOLSTER-OBJ: GUN-HOLSTER-ASSEMBLY -> HOLSTER-CLASS
 methods CREATE-GUN-HOLSTER-ASSEMBLY: -> GUN-HOLSTER-ASSEMBLY
 events NEW-GUN-HOLSTER-ASSEMBLY: -> GUN-HOLSTER-ASSEMBLY
 axioms
   ATTR-EQUAL(G1, G2) <=> (HOLSTER-OBJ(G1) = HOLSTER-OBJ(G2) & GUN-OBJ(G1) = GUN-OBJ(G2));
   ATTR-EQUAL(NEW-GUN-HOLSTER-ASSEMBLY(), CREATE-GUN-HOLSTER-ASSEMBLY());
   SIZE(HOLSTER-OBJ(G)) = 1;
   SIZE(GUN-OBJ(G)) = 1
end-class


class GUN-HOLSTER-ASSEMBLY-CLASS is
 class-sort GUN-HOLSTER-ASSEMBLY-CLASS
 contained-class GUN-HOLSTER-ASSEMBLY
 events NEW-GUN-HOLSTER-ASSEMBLY-CLASS: -> GUN-HOLSTER-ASSEMBLY-CLASS
 axioms NEW-GUN-HOLSTER-ASSEMBLY-CLASS() = EMPTY-SET
end-class


aggregate GUN-HOLSTER-ASSEMBLY-AGGREGATE is
 nodes RELEASE-HOLSTER-SWITCH, FREE-CLUTCH, ENGAGE-CLUTCH,
       CLOSE-HOLSTER-SWITCH, DISABLE-PUMP, START-TIMER, GUN-CLASS,
       GUN, HOLSTER-CLASS, HOLSTER
 arcs RELEASE-HOLSTER-SWITCH -> HOLSTER: {RELEASE-HOLSTER-SWITCH-SORT -> HOLSTER},
      CLOSE-HOLSTER-SWITCH -> HOLSTER: {CLOSE-HOLSTER-SWITCH-SORT -> HOLSTER},
      GUN -> GUN-CLASS: {},
      HOLSTER -> HOLSTER-CLASS: {},
      RELEASE-HOLSTER-SWITCH -> GUN: {},
      FREE-CLUTCH -> GUN: {},
      ENGAGE-CLUTCH -> GUN: {},
      CLOSE-HOLSTER-SWITCH -> GUN: {},
      DISABLE-PUMP -> GUN: {},
 START-TIMER -> GUN: {}
end-aggregate
```

```
class PUMP is
 class-sort PUMP
 import PUMP-ID, RESET-DISPLAY, START-PUMP-MOTOR, PUMP-AGGREGATE
 sort PUMP-STATE operations ATTR-EQUAL: PUMP, PUMP -> BOOLEAN
 attributes
   PUMP-ID: PUMP -> PUMP-ID
   GUN-HOLSTER-ASSEMBLY-OBJ: PUMP -> GUN-HOLSTER-ASSEMBLY-CLASS
   CLUTCH-MOTOR-ASSEMBLY-OBJ:
   PUMP -> CLUTCH-MOTOR-ASSEMBLY-CLASS
   DISPLAY-OBJ: PUMP -> DISPLAY-CLASS
   START-PUMP-MOTOR-OBJ: PUMP -> START-PUMP-MOTOR-SORT
   RESET-DISPLAY-OBJ: PUMP -> RESET-DISPLAY-SORT
 state-attributes PUMP-STATE: PUMP -> PUMP-STATE
 methods
   CREATE-PUMP: PUMP-ID -> PUMP
   ENABLE-PUMP: PUMP -> PUMP
 states
   PUMP-DISABLED: -> PUMP-STATE
   PUMP-ENABLED: -> PUMP-STATE
 events
   ENABLE-PUMP: PUMP, PUMP-ID -> PUMP
   NEW-PUMP: PUMP-ID -> PUMP
   DISABLE-PUMP: PUMP -> PUMP
 axioms
   PUMP-DISABLED <> PUMP-ENABLED;
   ATTR-EQUAL(P1, P2) <=>
   (PUMP-ID(P1) = PUMP-ID(P2) & RESET-DISPLAY-OBJ(P1) = RESET-DISPLAY-OBJ(P2)
                  & START-PUMP-MOTOR-OBJ(P1) = START-PUMP-MOTOR-OBJ(P2)
                  & DISPLAY-OBJ(P1) = DISPLAY-OBJ(P2)
                  & CLUTCH-MOTOR-ASSEMBLY-OBJ(P1) = CLUTCH-MOTOR-ASSEMBLY-OBJ(P2)
                  & GUN-HOLSTER-ASSEMBLY-OBJ(P1) = GUN-HOLSTER-ASSEMBLY-OBJ(P2));
   (PUMP-STATE(P) = PUMP-ENABLED) => (PUMP-STATE(DISABLE-PUMP(P)) = PUMP-DISABLED);
   (PUMP-STATE(NEW-PUMP(P, A)) = PUMP-DISABLED
                  & ATTR-EQUAL(NEW-PUMP(P, A), CREATE-PUMP(P, A)));
   (PUMP-STATE(P) = PUMP-DISABLED &(X = PUMP-ID(P)))
      => (PUMP-STATE(ENABLE-PUMP(P, X)) = PUMP-ENABLED
          & RESET-DISPLAY-OBJ(ENABLE-PUMP(P, X)) = RESET-DISPLAY(RESET-DISPLAY-OBJ(P))
          & START-PUMP-MOTOR-OBJ(ENABLE-PUMP(P, X))
                         = START-PUMP-MOTOR(START-PUMP-MOTOR-OBJ(P)));
   (PUMP-STATE(P) = PUMP-DISABLED &(X <> PUMP-ID(P)))
      => PUMP-STATE(ENABLE-PUMP(P, X)) = PUMP-DISABLED;
   PUMP-STATE(P) = PUMP-DISABLED => PUMP-STATE(DISABLE-PUMP(P)) = PUMP-DISABLED;
   PUMP-STATE(P) = PUMP-ENABLED => PUMP-STATE(ENABLE-PUMP(P, X)) = PUMP-ENABLED
end-class
```

```
class REGULAR is
 class-sort REGULAR < PUMP
 import PUMP
 methods CREATE-REGULAR: -> REGULAR
 events NEW-REGULAR: -> REGULAR
 axioms ATTR-EQUAL(NEW-REGULAR(), CREATE-REGULAR())
end-class
```

```
class REGULAR-CLASS is
 class-sort REGULAR-CLASS
 contained-class REGULAR
 import PUMP-CLASS
 events NEW-REGULAR-CLASS: -> REGULAR-CLASS
 axioms NEW-REGULAR-CLASS() = EMPTY-SET
end-class


class SOPHISTICATED is
 class-sort SOPHISTICATED < PUMP
 import PUMP
 operations ATTR-EQUAL: SOPHISTICATED, SOPHISTICATED -> BOOLEAN
 attributes
   VOLUME-SELECT: SOPHISTICATED -> VOLUME
   AMOUNT-SELECT: SOPHISTICATED -> AMOUNT
 methods
  CREATE-SOPHISTICATED: -> SOPHISTICATED
 events
  NEW-SOPHISTICATED: -> SOPHISTICATED
 axioms
   ATTR-EQUAL(S1, S2) <=> (PUMP.ATTR-EQUAL(S1, S2)
       & AMOUNT-SELECT(S1) = AMOUNT-SELECT(S2) & VOLUME-SELECT(S1) = VOLUME-SELECT(S2));
   AMOUNT-SELECT(CREATE-SOPHISTICATED(S)) = 0;
   VOLUME-SELECT(CREATE-SOPHISTICATED(S)) = 0;
   ATTR-EQUAL(NEW-SOPHISTICATED(), CREATE-SOPHISTICATED())
end-class


class SOPHISTICATED-CLASS is
 class-sort SOPHISTICATED-CLASS
 contained-class SOPHISTICATED
 import PUMP-CLASS
 events NEW-SOPHISTICATED-CLASS: -> SOPHISTICATED-CLASS
 axioms NEW-SOPHISTICATED-CLASS() = EMPTY-SETend-class


class PUMP-CLASS is
 class-sort PUMP-CLASS contained-class PUMP
 events
   ENABLE-PUMP: PUMP-CLASS, PUMP-ID -> PUMP-CLASS
   DISABLE-PUMP: PUMP-CLASS -> PUMP-CLASS
   NEW-PUMP-CLASS: -> PUMP-CLASS
 axioms
   NEW-PUMP-CLASS() = EMPTY-SET;
   fa ((P: PUMP), PC: PUMP-CLASS) in(P, PC) <=> in(DISABLE-PUMP(P), DISABLE-PUMP(PC));
   fa ((P: PUMP), (PC: PUMP-CLASS), X: PUMP-ID) in(P, PC)
       <=> in(ENABLE-PUMP(P, X), ENABLE-PUMP(PC, X))
end-class


aggregate PUMP-AGGREGATE is
 nodes FREE-CLUTCH, ENGAGE-CLUTCH, GUN-HOLSTER-ASSEMBLY-AGGREGATE,
       GUN-HOLSTER-ASSEMBLY-CLASS, GUN-HOLSTER-ASSEMBLY,
       CLUTCH-MOTOR-ASSEMBLY-AGGREGATE, CLUTCH-MOTOR-ASSEMBLY-CLASS,
       CLUTCH-MOTOR-ASSEMBLY, GRADE, AMOUNT, VOLUME, DISPLAY-CLASS,
       DISPLAY
  arcs FREE-CLUTCH -> GUN-HOLSTER-ASSEMBLY-AGGREGATE: {},
       FREE-CLUTCH -> CLUTCH-MOTOR-ASSEMBLY-AGGREGATE: {},
       FREE-CLUTCH -> CLUTCH-MOTOR-ASSEMBLY-AGGREGATE: {FREE-CLUTCH-SORT -> CLUTCH},
       ENGAGE-CLUTCH -> GUN-HOLSTER-ASSEMBLY-AGGREGATE: {},
       ENGAGE-CLUTCH -> CLUTCH-MOTOR-ASSEMBLY-AGGREGATE: {ENGAGE-CLUTCH-SORT -> CLUTCH},
       GUN-HOLSTER-ASSEMBLY-AGGREGATE -> GUN-HOLSTER-ASSEMBLY: {},
       GUN-HOLSTER-ASSEMBLY -> GUN-HOLSTER-ASSEMBLY-CLASS: {},
       CLUTCH-MOTOR-ASSEMBLY-AGGREGATE -> CLUTCH-MOTOR-ASSEMBLY: {},
       CLUTCH-MOTOR-ASSEMBLY -> CLUTCH-MOTOR-ASSEMBLY-CLASS: {},
       DISPLAY -> DISPLAY-CLASS: {},
       GRADE -> DISPLAY: {},
       AMOUNT -> DISPLAY: {},
       VOLUME -> DISPLAY: {}
end-aggregate


aggregate DOMAIN-THEORY-AGGREGATE is
 nodes DISABLE-PUMP, REGULAR-CLASS, REGULAR, SOPHISTICATED-CLASS,
       SOPHISTICATED, PUMP-ID, START-PUMP-MOTOR, RESET-DISPLAY,
       PUMP-AGGREGATE, PUMP-CLASS, PUMP
 arcs DISABLE-PUMP -> PUMP: {DISABLE-PUMP-SORT -> PUMP},
      DISABLE-PUMP -> PUMP-AGGREGATE: {},
      REGULAR -> REGULAR-CLASS: {},
      SOPHISTICATED -> SOPHISTICATED-CLASS: {},
      START-PUMP-MOTOR -> PUMP-AGGREGATE: {START-PUMP-MOTOR-SORT -> MOTOR},
      RESET-DISPLAY -> PUMP-AGGREGATE: {RESET-DISPLAY-SORT -> DISPLAY},
      PUMP-AGGREGATE -> PUMP: {},
      PUMP -> PUMP-CLASS: {},
      PUMP-ID -> PUMP: {},
      START-PUMP-MOTOR -> PUMP: {},
      RESET-DISPLAY -> PUMP: {},
      PUMP-CLASS -> REGULAR-CLASS: {},
      PUMP -> REGULAR: {},
      PUMP-CLASS -> SOPHISTICATED-CLASS: {},
      PUMP -> SOPHISTICATED: {}
end-aggregate


event START-FUEL is
 class-sort START-FUEL-SORT
 events START-FUEL: START-FUEL-SORT -> START-FUEL-SORT
end-event


event FREE-CLUTCH is
 class-sort FREE-CLUTCH-SORT
 events FREE-CLUTCH: FREE-CLUTCH-SORT -> FREE-CLUTCH-SORT
end-event


event DISABLE-CLUTCH is
 class-sort DISABLE-CLUTCH-SORT
 events DISABLE-CLUTCH: DISABLE-CLUTCH-SORT -> DISABLE-CLUTCH-SORT
end-event
```

```
event ENGAGE-CLUTCH is
 class-sort ENGAGE-CLUTCH-SORT
 events ENGAGE-CLUTCH: ENGAGE-CLUTCH-SORT -> ENGAGE-CLUTCH-SORT
end-event


event START-TIMER is
 class-sort START-TIMER-SORT
 events START-TIMER: START-TIMER-SORT -> START-TIMER-SORT
end-event


event CLOSE-HOLSTER-SWITCH is
 class-sort CLOSE-HOLSTER-SWITCH-SORT
 events
   CLOSE-HOLSTER-SWITCH: CLOSE-HOLSTER-SWITCH-SORT -> CLOSE-HOLSTER-SWITCH-SORT
end-event


event RELEASE-HOLSTER-SWITCH is
 class-sort RELEASE-HOLSTER-SWITCH-SORT
 events RELEASE-HOLSTER-SWITCH: RELEASE-HOLSTER-SWITCH-SORT -> RELEASE-HOLSTER-SWITCH-SORT
end-event


event RESET-DISPLAY is
 class-sort RESET-DISPLAY-SORT
 events RESET-DISPLAY: RESET-DISPLAY-SORT -> RESET-DISPLAY-SORT
end-event


event START-PUMP-MOTOR is
 class-sort START-PUMP-MOTOR-SORT
 events START-PUMP-MOTOR: START-PUMP-MOTOR-SORT -> START-PUMP-MOTOR-SORT
end-event


event DISABLE-PUMP is
 class-sort DISABLE-PUMP-SORT
 events DISABLE-PUMP: DISABLE-PUMP-SORT -> DISABLE-PUMP-SORT
end-event
```

G.2 Faculty Student Database – Faculty Workload Functional Model

Figures G.1 through G.4 define the *Faculty Workload* process. This functional model is translated into the *Faculty-Workload* class in Section G.3.

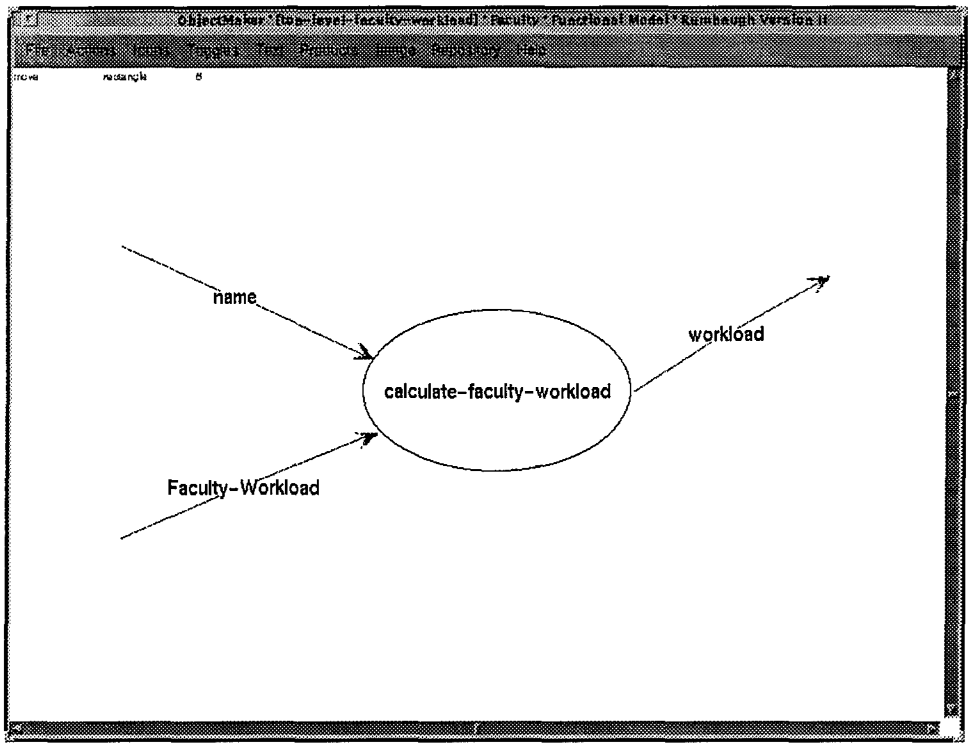

Figure G.1 Faculty Workload Functional Model

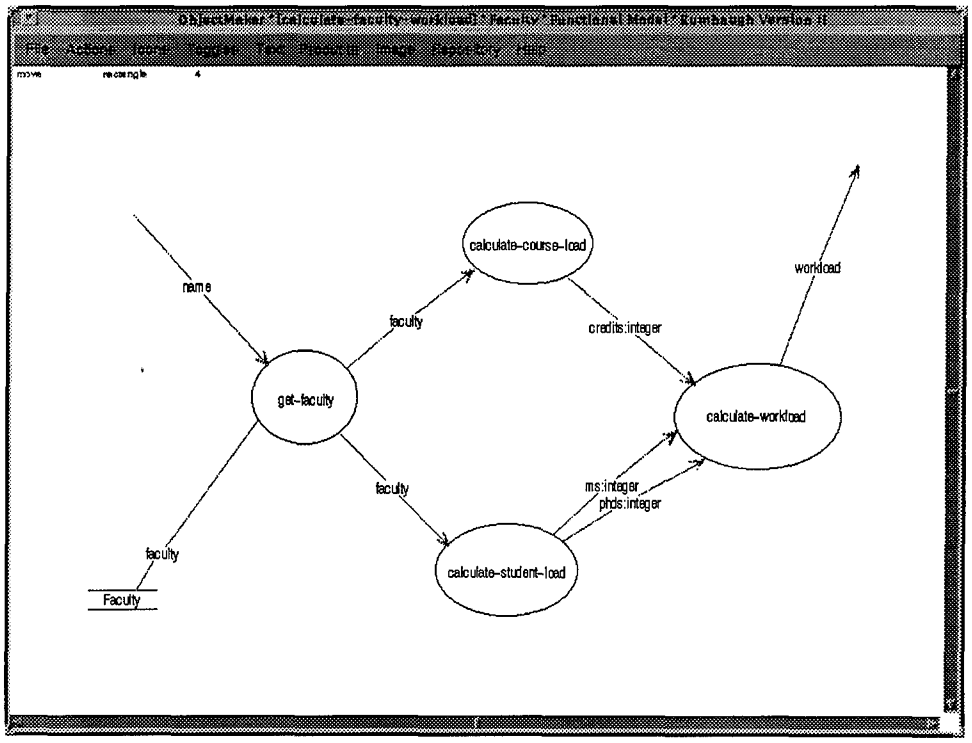

Figure G.2 Calculate-Faculty-Workload Functional Model

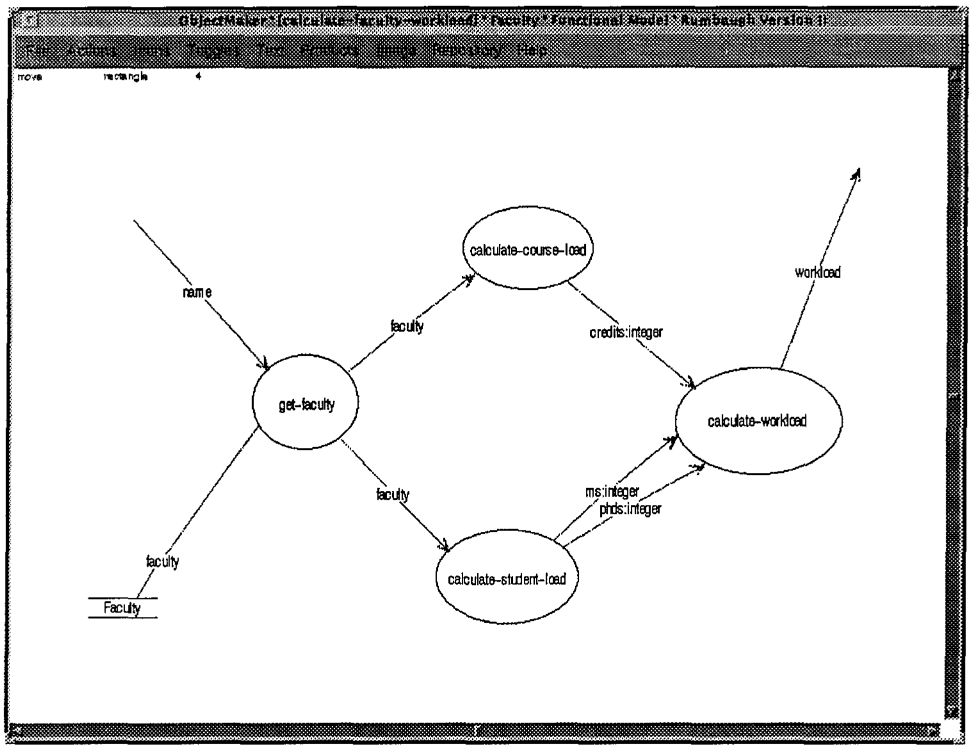

Figure G.3 Calculate-Student-Load Functional Model

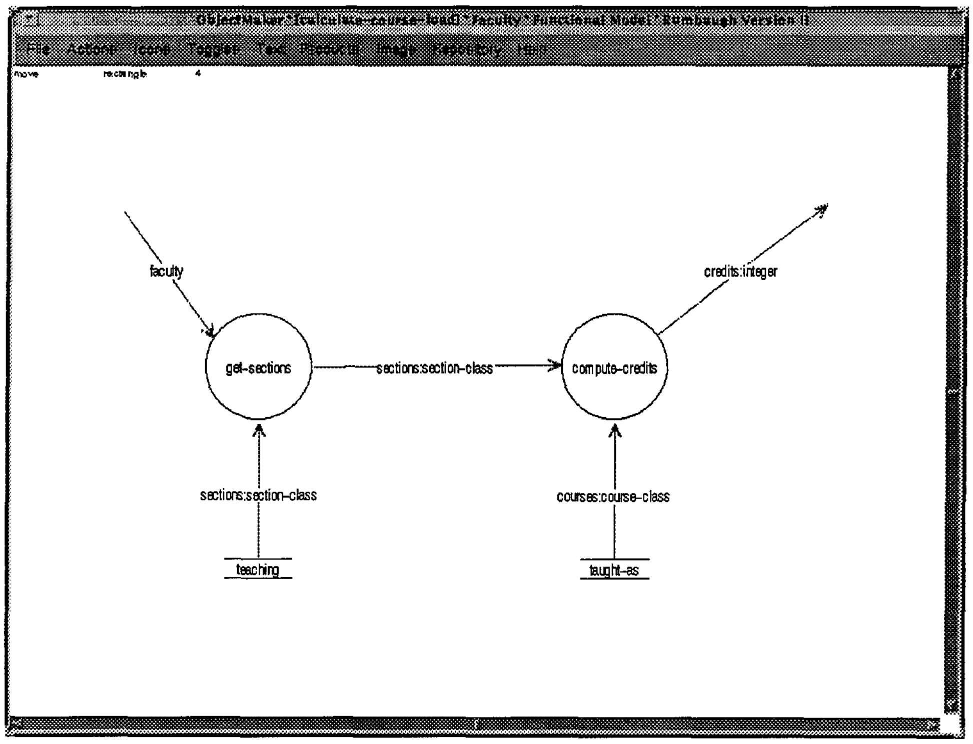

Figure G.4 Calculate-Course-Load Functional Model

G.3 Faculty Student Database O-SLANG

```
class SECTION is
 class-sort SECTION
 import NUMBER
 operations ATTR-EQUAL: SECTION, SECTION -> BOOLEAN
 attributes NUMBER: SECTION -> NUMBER
 methods CREATE-SECTION: -> SECTION
 events NEW-SECTION: -> SECTION
 axioms
   ATTR-EQUAL(S1, S2) <=> (NUMBER(S1) = NUMBER(S2));
   ATTR-EQUAL(NEW-SECTION(), CREATE-SECTION())
end-class


class SECTION-CLASS is
 class-sort SECTION-CLASS
 contained-class SECTION
 events NEW-SECTION-CLASS: -> SECTION-CLASS
 axioms NEW-SECTION-CLASS() = EMPTY-SET
end-class


class A-CLASS is
 class-sort A-CLASS
 import PROGRAM
 operations ATTR-EQUAL: A-CLASS, A-CLASS -> BOOLEAN
 attributes PROGRAM: A-CLASS -> PROGRAM
 methods CREATE-A-CLASS: -> A-CLASS
 events NEW-A-CLASS: -> A-CLASS
 axioms
   ATTR-EQUAL(A1, A2) <=> (PROGRAM(A1) = PROGRAM(A2));
   ATTR-EQUAL(NEW-A-CLASS(), CREATE-A-CLASS())
end-class


class A-CLASS-CLASS is
 class-sort A-CLASS-CLASS
 contained-class A-CLASS
 events NEW-A-CLASS-CLASS: -> A-CLASS-CLASS
 axioms NEW-A-CLASS-CLASS() = EMPTY-SET
end-class


class QUARTER is
 class-sort QUARTER
 import END-DATE, START-DATE, QUATER-YEAR, QUARTER-NAME
 operations ATTR-EQUAL: QUARTER, QUARTER -> BOOLEAN
 attributes
   QUARTER-NAME: QUARTER -> QUARTER-NAME
   QUATER-YEAR: QUARTER -> QUATER-YEAR
   START-DATE: QUARTER -> START-DATE
   END-DATE: QUARTER -> END-DATE
 methods CREATE-QUARTER: -> QUARTER
 events NEW-QUARTER: -> QUARTER
 axioms
```

```
   ATTR-EQUAL(Q1, Q2) <=> (END-DATE(Q1) = END-DATE(Q2)
                         & START-DATE(Q1) = START-DATE(Q2)
                         & QUATER-YEAR(Q1) = QUATER-YEAR(Q2)
                         & QUARTER-NAME(Q1) = QUARTER-NAME(Q2));
   ATTR-EQUAL(NEW-QUARTER(), CREATE-QUARTER())
end-class


class QUARTER-CLASS is
 class-sort QUARTER-CLASS
 contained-class QUARTER
 events NEW-QUARTER-CLASS: -> QUARTER-CLASS
 axioms NEW-QUARTER-CLASS() = EMPTY-SET
end-class


class COURSE is
 class-sort COURSE
 import ABET-OTHER, ABET-MATH, ABET-SCIENCE, ABET-DESIGN, LAB-HOURS,
        LEXTURE-HOURS, CREDIT-HOURS, DESCRIPTION, TITLE, NUM, TYPE
 operations ATTR-EQUAL: COURSE, COURSE -> BOOLEAN
 attributes
   TYPE: COURSE -> TYPE
   NUM: COURSE -> NUM
   TITLE: COURSE -> TITLE
   DESCRIPTION: COURSE -> DESCRIPTION
   CREDIT-HOURS: COURSE -> CREDIT-HOURS
   LEXTURE-HOURS: COURSE -> LEXTURE-HOURS
   LAB-HOURS: COURSE -> LAB-HOURS
   ABET-DESIGN: COURSE -> ABET-DESIGN
   ABET-SCIENCE: COURSE -> ABET-SCIENCE
   ABET-MATH: COURSE -> ABET-MATH
   ABET-OTHER: COURSE -> ABET-OTHER
 methods CREATE-COURSE: -> COURSE
 events NEW-COURSE: -> COURSE
 axioms
   ATTR-EQUAL(C1, C2) <=> (ABET-OTHER(C1) = ABET-OTHER(C2)
                         & ABET-MATH(C1) = ABET-MATH(C2)
                         & ABET-SCIENCE(C1) = ABET-SCIENCE(C2)
                         & ABET-DESIGN(C1) = ABET-DESIGN(C2)
                         & LAB-HOURS(C1) = LAB-HOURS(C2)
                         & LEXTURE-HOURS(C1) = LEXTURE-HOURS(C2)
                         & CREDIT-HOURS(C1) = CREDIT-HOURS(C2)
                         & DESCRIPTION(C1) = DESCRIPTION(C2) & TITLE(C1) = TITLE(C2)
                         & NUM(C1) = NUM(C2) & TYPE(C1) = TYPE(C2));
   ATTR-EQUAL(NEW-COURSE(), CREATE-COURSE())
end-class


class COURSE-CLASS is
 class-sort COURSE-CLASS contained-class COURSE
 events NEW-COURSE-CLASS: -> COURSE-CLASS
 axioms NEW-COURSE-CLASS() = EMPTY-SET
end-class
```

```
class FACULTY is
 class-sort FACULTY
 import ACADEMIC-RANK, SEX, SSAN, AGE, BIRTHDATE, FIRSTNAME,
        INITIAL, LAST-NAME
 operations ATTR-EQUAL: FACULTY, FACULTY -> BOOLEAN
 attributes
   LAST-NAME: FACULTY -> LAST-NAME
   INITIAL: FACULTY -> INITIAL
   FIRSTNAME: FACULTY -> FIRSTNAME
   BIRTHDATE: FACULTY -> BIRTHDATE
   AGE: FACULTY -> AGE
   SSAN: FACULTY -> SSAN
   SEX: FACULTY -> SEX
   ACADEMIC-RANK: FACULTY -> ACADEMIC-RANK
 methods CREATE-FACULTY: -> FACULTY
 events NEW-FACULTY: -> FACULTY
 axioms
   ATTR-EQUAL(F1, F2) <=> (ACADEMIC-RANK(F1) = ACADEMIC-RANK(F2)
                        & SEX (F1) = SEX (F2) & SSAN(F1) = SSAN(F2)
                        & BIRTHDATE(F1) = BIRTHDATE(F2) & FIRSTNAME(F1) = FIRSTNAME(F2)
                        & INITIAL(F1) = INITIAL(F2) & LAST-NAME(F1) = LAST-NAME(F2));
 ATTR-EQUAL(NEW-FACULTY(), CREATE-FACULTY())
end-class


class FACULTY-CLASS is
 class-sort FACULTY-CLASS
 contained-class FACULTY
 events NEW-FACULTY-CLASS: -> FACULTY-CLASS
 axioms NEW-FACULTY-CLASS() = EMPTY-SET
end-class


class STUDENT is
 class-sort STUDENT
 import GPA, WEIGHT, HEIGHT, SEX, SSAN, AGE, BIRTH-DATE, FIRST-NAME,
        INIT, LASTNAME
 operations ATTR-EQUAL: STUDENT, STUDENT -> BOOLEAN
 attributes
   LASTNAME: STUDENT -> LASTNAME
   INIT: STUDENT -> INIT
   FIRST-NAME: STUDENT -> FIRST-NAME
   BIRTH-DATE: STUDENT -> BIRTH-DATE
   AGE: STUDENT -> AGE
   SSAN: STUDENT -> SSAN
   SEX: STUDENT -> SEX
   HEIGHT: STUDENT -> HEIGHT
   WEIGHT: STUDENT -> WEIGHT
   GPA: STUDENT -> GPA
 methods CREATE-STUDENT: -> STUDENT
 events NEW-STUDENT: -> STUDENT
 axioms
   ATTR-EQUAL(S1, S2) <=> (GPA(S1) = GPA(S2) & WEIGHT(S1) = WEIGHT(S2)
                        & HEIGHT(S1) = HEIGHT(S2) & SEX (S1) = SEX (S2)
                        & SSAN(S1) = SSAN(S2) & BIRTH-DATE(S1) = BIRTH-DATE(S2)
                        & FIRST-NAME(S1) = FIRST-NAME(S2) & INIT(S1) = INIT(S2)
                        & LASTNAME(S1) = LASTNAME(S2));
   ATTR-EQUAL(NEW-STUDENT(), CREATE-STUDENT())
end-class


class STUDENT-CLASS is
 class-sort STUDENT-CLASS
 contained-class STUDENT
 events NEW-STUDENT-CLASS: -> STUDENT-CLASS
 axioms NEW-STUDENT-CLASS() = EMPTY-SET
end-class


class FACULTY-WORKLOAD is
  class-sort FACULTY-WORKLOAD
  import FACULTY-WORKLOAD-AGGREGATE
  operations
    ATTR-EQUAL: FACULTY-WORKLOAD, FACULTY-WORKLOAD -> BOOLEAN
    GET-SECTIONS: TEACHING, FACULTY -> SECTION-CLASS
    COMPUTE-CREDITS: TAUGHT-AS, SECTION-CLASS -> INTEGER
    CALCULATE-COURSE-LOAD: FACULTY-WORKLOAD, FACULTY -> INTEGER
    GET-STUDENTS-ADVISED: ADVISES, FACULTY -> STUDENT-CLASS
    COUNT-STUDENTS: MEMBER-OF, STUDENT-CLASS -> INTEGER
    CALCULATE-STUDENT-LOAD: FACULTY-WORKLOAD, FACULTY -> INTEGER, INTEGER
    CALCULATE-WORKLOAD: INTEGER, INTEGER, INTEGER -> WORKLOAD
    CALCULATE-FACULTY-WORKLOAD: FACULTY-WORKLOAD, NAME -> WORKLOAD
    GET-FACULTY: FACULTY-CLASS, NAME, NAME -> FACULTY
    GET-COURSE: COURSE-CLASS, NUM, TYPE -> COURSE
    GET-SECTIONS-TAUGHT: SECTION-CLASS, FACULTY -> SECTION-CLASS
    GET-SECTIONS-OFFERED: SECTION-CLASS, COURSE -> SECTION-CLASS
    COMPUTE-SECTION-UNION: SECTION-CLASS, SECTION-CLASS -> TIMES-TAUGHT
    COUNT-TIMES-TAUGHT: SECTION-CLASS, COURSE, FACULTY -> TIMES-TAUGHT
    GET-TEACHES: TEACHES, FACULTY, COURSE -> TEACHES-LINK
  attributes
    STUDENT-OBJ: FACULTY-WORKLOAD -> STUDENT-CLASS
    FACULTY-OBJ: FACULTY-WORKLOAD -> FACULTY-CLASS
    SECTION-OBJ: FACULTY-WORKLOAD -> SECTION-CLASS
    COURSE-OBJ: FACULTY-WORKLOAD -> COURSE-CLASS
    QUARTER-OBJ: FACULTY-WORKLOAD -> QUARTER-CLASS
    A-CLASS-OBJ: FACULTY-WORKLOAD -> A-CLASS-CLASS
    MEMBER-OF-OBJ: FACULTY-WORKLOAD -> MEMBER-OF
    ADVISES-OBJ: FACULTY-WORKLOAD -> ADVISES
    TEACHES-OBJ: FACULTY-WORKLOAD -> TEACHES
    OFFERING-OBJ: FACULTY-WORKLOAD -> OFFERING
    TAUGHT-AS-OBJ: FACULTY-WORKLOAD -> TAUGHT-AS
    SCHEDULED-IN-OBJ: FACULTY-WORKLOAD -> SCHEDULED-IN
    TEACHING-OBJ: FACULTY-WORKLOAD -> TEACHING
  methods
    CREATE-FACULTY-WORKLOAD: -> FACULTY-WORKLOAD
    MODIFY-TEACHES: TEACHES, TIMES-TAUGHT, TEACHES-LINK -> TEACHES
    UPDATE-TEACHES: FACULTY-WORKLOAD, NUM, NAME, TYPE -> FACULTY-WORKLOAD
  events
    CALCULATE-FACULTY-WORKLOAD-EVENT: FACULTY-WORKLOAD, NAME -> WORKLOAD
    UPDATE-TEACHES-EVENT: FACULTY-WORKLOAD, NUM, NAME, TYPE -> FACULTY-WORKLOAD
    NEW-FACULTY-WORKLOAD: -> FACULTY-WORKLOAD
  axioms
```

```
    ATTR-EQUAL (F1, F2) <=> (A-CLASS-OBJ (F1) = A-CLASS-OBJ (F2)
         & QUARTER-OBJ (F1) = QUARTER-OBJ (F2)
         & COURSE-OBJ (F1) = COURSE-OBJ (F2)
         & SECTION-OBJ (F1) = SECTION-OBJ (F2)
         & FACULTY-OBJ (F1) = FACULTY-OBJ (F2)
         & STUDENT-OBJ (F1) = STUDENT-OBJ (F2));
    UPDATE-TEACHES (F, NUM, NAME, TYPE) = F1
      & TEACHES-OBJ (F1) = MODIFY-TEACHES (TEACHES-OBJ (F), TIMES-TAUGHT, TEACHES-LINK)
      & TEACHES-LINK = GET-TEACHES(TEACHES-OBJ (F), FACULTY, COURSE)
      & TIMES-TAUGHT = COUNT-TIMES-TAUGHT (SECTION-OBJ (F), COURSE, FACULTY)
      & COURSE = GET-COURSE (COURSE-OBJ (F), NUM, TYPE)
      & FACULTY = GET-FACULTY (FACULTY-OBJ (F), NAME, NAME);
    COUNT-TIMES-TAUGHT (SECTION-OBJ (F), COURSE, FACULTY) = TIMES-TAUGHT
      & TIMES-TAUGHT = COMPUTE-SECTION-UNION (C, F)
      & C = GET-SECTIONS-OFFERED (SECTION-OBJ (F), COURSE)
      & F = GET-SECTIONS-TAUGHT (SECTION-OBJ (F), FACULTY);
    CALCULATE-FACULTY-WORKLOAD (F, NAME) = WORKLOAD
      & WORKLOAD = CALCULATE-WORKLOAD (PHDS, MS, CREDITS)
      & <MS, PHDS> = CALCULATE-STUDENT-LOAD (F, FACULTY)
      & CREDITS = CALCULATE-COURSE-LOAD (F, FACULTY);
    CALCULATE-STUDENT-LOAD (F, FACULTY) = <MS, PHDS>
      & MS = COUNT-STUDENTS (MEMBER-OF-OBJ (F), STUDENTS)
      & STUDENTS = GET-STUDENTS-ADVISED (ADVISES-OBJ (F), FACULTY);
    CALCULATE-COURSE-LOAD (F, FACULTY) = CREDITS
      & CREDITS = COMPUTE-CREDITS (TAUGHT-AS-OBJ (F), SECTIONS)
      & SECTIONS = GET-SECTIONS (TEACHING-OBJ (F), FACULTY);
    ATTR-EQUAL (NEW-FACULTY-WORKLOAD (), CREATE-FACULTY-WORKLOAD ());
    ATTR-EQUAL (UPDATE-TEACHES-EVENT (F, N, A, T), UPDATE-TEACHES (F, N, A, T));
    ATTR-EQUAL (CALCULATE-FACULTY-WORKLOAD-EVENT (F, N),
                CALCULATE-FACULTY-WORKLOAD (F, N))
end-class


class FACULTY-WORKLOAD-CLASS is
 class-sort FACULTY-WORKLOAD-CLASS
 contained-class FACULTY-WORKLOAD
 events NEW-FACULTY-WORKLOAD-CLASS: -> FACULTY-WORKLOAD-CLASS
 axioms NEW-FACULTY-WORKLOAD-CLASS() = EMPTY-SET
end-class


aggregate FACULTY-WORKLOAD-AGGREGATE is
 nodes MEMBER-OF, MEMBER-OF-LINK, ADVISES, ADVISES-LINK, TEACHES,
       TEACHES-LINK, OFFERING, OFFERING-LINK, TAUGHT-AS,
       TAUGHT-AS-LINK, SCHEDULED-IN, SCHEDULED-IN-LINK, TEACHING,
       TEACHING-LINK, GPA, WEIGHT, HEIGHT, BIRTH-DATE, FIRST-NAME,
       INIT, LASTNAME, STUDENT-CLASS, STUDENT, ACADEMIC-RANK, SEX,
       SSAN, AGE, BIRTHDATE, FIRSTNAME, INITIAL, LAST-NAME,
       FACULTY-CLASS, FACULTY, NUMBER, SECTION-CLASS, SECTION,
       ABET-OTHER, ABET-MATH, ABET-SCIENCE, ABET-DESIGN, LAB-HOURS,
       LEXTURE-HOURS, CREDIT-HOURS, DESCRIPTION, TITLE, NUM, TYPE,
       COURSE-CLASS, COURSE,END-DATE, START-DATE, QUATER-YEAR,
       QUARTER-NAME, QUARTER-CLASS, QUARTER, PROGRAM, A-CLASS-CLASS,
       A-CLASS, TRIV-127: TRIV, TRIV-128: TRIV, TRIV-129: TRIV,
       TRIV-130: TRIV, TRIV-131: TRIV, TRIV-132: TRIV, TRIV-133: TRIV,
       TRIV-134: TRIV, TRIV-135: TRIV, TRIV-136: TRIV, TRIV-137: TRIV,
      TRIV-138: TRIV, TRIV-139: TRIV, TRIV-140: TRIV
arcs TRIV-140 -> MEMBER-OF-LINK: { E -> STUDENT-OBJ},
     TRIV-140 -> STUDENT: { E -> STUDENT},
     TRIV-139 -> MEMBER-OF-LINK: { E -> A-CLASS-OBJ},
     TRIV-139 -> A-CLASS: { E -> A-CLASS},
     MEMBER-OF-LINK -> MEMBER-OF: {},
     TRIV-138 -> ADVISES-LINK: { E -> FACULTY-OBJ},
     TRIV-138 -> FACULTY: { E -> FACULTY},
     TRIV-137 -> ADVISES-LINK: { E -> STUDENT-OBJ},
     TRIV-137 -> STUDENT: { E -> STUDENT},
     ADVISES-LINK -> ADVISES: {},
     TRIV-136 -> TEACHES-LINK: { E -> COURSE-OBJ},
     TRIV-136 -> COURSE: { E -> COURSE},
     TRIV-135 -> TEACHES-LINK: { E -> FACULTY-OBJ},
     TRIV-135 -> FACULTY: { E -> FACULTY},
     TEACHES-LINK -> TEACHES: {},
     TRIV-134 -> OFFERING-LINK: { E -> QUARTER-OBJ},
     TRIV-134 -> QUARTER: { E -> QUARTER},
     TRIV-133 -> OFFERING-LINK: { E -> COURSE-OBJ},
     TRIV-133 -> COURSE: { E -> COURSE},
     OFFERING-LINK -> OFFERING: {},
     TRIV-132 -> TAUGHT-AS-LINK: { E -> SECTION-OBJ},
     TRIV-132 -> SECTION: { E -> SECTION},
     TRIV-131 -> TAUGHT-AS-LINK: { E -> COURSE-OBJ},
     TRIV-131 -> COURSE: { E -> COURSE},
     TAUGHT-AS-LINK -> TAUGHT-AS: {},
     TRIV-130 -> SCHEDULED-IN-LINK: { E -> SECTION-OBJ},
     TRIV-130 -> SECTION: { E -> SECTION},
     TRIV-129 -> SCHEDULED-IN-LINK: { E -> QUARTER-OBJ},
     TRIV-129 -> QUARTER: { E -> QUARTER},
     SCHEDULED-IN-LINK -> SCHEDULED-IN: {},
     TRIV-128 -> TEACHING-LINK: { E -> SECTION-OBJ},
     TRIV-128 -> SECTION: { E -> SECTION},
     TRIV-127 -> TEACHING-LINK: { E -> FACULTY-OBJ},
     TRIV-127 -> FACULTY: { E -> FACULTY},
     TEACHING-LINK -> TEACHING: {},
     STUDENT -> STUDENT-CLASS: {},
     FACULTY -> FACULTY-CLASS: {},
     SECTION -> SECTION-CLASS: {},
     COURSE -> COURSE-CLASS: {},
     QUARTER -> QUARTER-CLASS: {},
     A-CLASS -> A-CLASS-CLASS: {},
     GPA -> STUDENT: {},
     WEIGHT -> STUDENT: {},
     HEIGHT -> STUDENT: {},
     BIRTH-DATE -> STUDENT: {},
     FIRST-NAME -> STUDENT: {},
     INIT -> STUDENT: {},
     LASTNAME -> STUDENT: {},
     ACADEMIC-RANK -> FACULTY: {},
     SEX -> STUDENT: {},
     SEX -> FACULTY: {},
     SSAN -> STUDENT: {},
     SSAN -> FACULTY: {},
     AGE -> STUDENT: {},
     AGE -> FACULTY: {},
        BIRTHDATE -> FACULTY: {},
        FIRSTNAME -> FACULTY: {},
        INITIAL -> FACULTY: {},
        LAST-NAME -> FACULTY: {},
        NUMBER -> SECTION: {},
        ABET-OTHER -> COURSE: {},
        ABET-MATH -> COURSE: {},
        ABET-SCIENCE -> COURSE: {},
        ABET-DESIGN -> COURSE: {},
        LAB-HOURS -> COURSE: {},
        LEXTURE-HOURS -> COURSE: {},
        CREDIT-HOURS -> COURSE: {},
        DESCRIPTION -> COURSE: {},
        TITLE -> COURSE: {},
        NUM -> COURSE: {},
        TYPE -> COURSE: {},
        END-DATE -> QUARTER: {},
        START-DATE -> QUARTER: {},
        QUATER-YEAR -> QUARTER: {},
        QUARTER-NAME -> QUARTER: {},
        PROGRAM -> A-CLASS: {}
  end-aggregate


  link TEACHING-LINK is
   class-sort TEACHING-LINK
   sort SECTION, FACULTY
   operations ATTR-EQUAL: TEACHING-LINK, TEACHING-LINK -> BOOLEAN
   attributes
     FACULTY-OBJ: TEACHING-LINK -> FACULTY
     SECTION-OBJ: TEACHING-LINK -> SECTION
   methods CREATE-TEACHING-LINK: FACULTY, SECTION -> TEACHING-LINK
   events NEW-TEACHING-LINK: FACULTY, SECTION -> TEACHING-LINK
   axioms
     ATTR-EQUAL(T1, T2) <=> (SECTION-OBJ(T1) = SECTION-OBJ(T2)
                           & FACULTY-OBJ(T1) = FACULTY-OBJ(T2));
     SECTION-OBJ(CREATE-TEACHING-LINK(T, S, F)) = S;
     FACULTY-OBJ(CREATE-TEACHING-LINK(T, S, F)) = F;
     ATTR-EQUAL(NEW-TEACHING-LINK(T, S, F), (CREATE-TEACHING-LINK(T, S, F)))
  end-link


  association TEACHING is
   class-sort TEACHING
   link-class TEACHING-LINK
   sort SECTION-CLASS, FACULTY-CLASS
   operations
     IMAGE: TEACHING, SECTION -> FACULTY-CLASS
     IMAGE: TEACHING, FACULTY -> SECTION-CLASS
   events NEW-TEACHING: -> TEACHING
   axioms
     NEW-TEACHING() = EMPTY-SET;
     fa ((S: TEACHING), (T: SECTION), A: FACULTY)
        (ex (F: TEACHING-LINK) in(F, S) & TEACHING-OBJ(F) = T & TEACHING-OBJ(F) = A)
           <=> in(A, image(S, T));
     fa ((S: TEACHING), (T: SECTION), A: FACULTY)
```

```
      (ex (F: TEACHING-LINK) in(F, S) & TEACHING-OBJ(F) = A & TEACHING-OBJ(F) = T)
         <=> in(T, image(S, A))
end-association


link SCHEDULED-IN-LINK is
 class-sort SCHEDULED-IN-LINK
 sort SECTION, QUARTER
 operations ATTR-EQUAL: SCHEDULED-IN-LINK, SCHEDULED-IN-LINK -> BOOLEAN
 attributes
   QUARTER-OBJ: SCHEDULED-IN-LINK -> QUARTER
   SECTION-OBJ: SCHEDULED-IN-LINK -> SECTION
 methods
   CREATE-SCHEDULED-IN-LINK:
   QUARTER, SECTION -> SCHEDULED-IN-LINK
 events NEW-SCHEDULED-IN-LINK: QUARTER, SECTION -> SCHEDULED-IN-LINK
 axioms
   ATTR-EQUAL(S1, S2) <=> (SECTION-OBJ(S1) = SECTION-OBJ(S2)
                        & QUARTER-OBJ(S1) = QUARTER-OBJ(S2));
   SECTION-OBJ(CREATE-SCHEDULED-IN-LINK(S, A, Q)) = A;
   QUARTER-OBJ(CREATE-SCHEDULED-IN-LINK(S, A, Q)) = Q;
   ATTR-EQUAL (NEW-SCHEDULED-IN-LINK(S, A, Q), (CREATE-SCHEDULED-IN-LINK(S, A, Q)))
end-link


association SCHEDULED-IN is
 class-sort SCHEDULED-IN
 link-class SCHEDULED-IN-LINK
 sort SECTION-CLASS, QUARTER-CLASS
 operations
   IMAGE: SCHEDULED-IN, SECTION -> QUARTER-CLASS
   IMAGE: SCHEDULED-IN, QUARTER -> SECTION-CLASS
 events NEW-SCHEDULED-IN: -> SCHEDULED-IN
 axioms
   NEW-SCHEDULED-IN() = EMPTY-SET;
   fa ((S: SCHEDULED-IN), A: SECTION) SIZE(IMAGE(S, A)) = 1;
   fa ((S: SCHEDULED-IN), (A: SECTION), B: QUARTER)
      (ex (Q: SCHEDULED-IN-LINK) in(Q, S) & SCHEDULED-IN-OBJ(Q) = A
        & SCHEDULED-IN-OBJ(Q) = B) <=> in(B, image(S, A));
   fa ((S: SCHEDULED-IN), (A: SECTION), B: QUARTER)
      (ex (Q: SCHEDULED-IN-LINK) in(Q, S) & SCHEDULED-IN-OBJ(Q) = B
        & SCHEDULED-IN-OBJ(Q) = A) <=> in(A, image(S, B))
end-association


link TAUGHT-AS-LINK is
 class-sort TAUGHT-AS-LINK
 sort SECTION, COURSE
 operations ATTR-EQUAL: TAUGHT-AS-LINK, TAUGHT-AS-LINK -> BOOLEAN
 attributes
   COURSE-OBJ: TAUGHT-AS-LINK -> COURSE
   SECTION-OBJ: TAUGHT-AS-LINK -> SECTION
 methods CREATE-TAUGHT-AS-LINK: COURSE, SECTION -> TAUGHT-AS-LINK
 events NEW-TAUGHT-AS-LINK: COURSE, SECTION -> TAUGHT-AS-LINK
 axioms
   ATTR-EQUAL(T1, T2) <=> (SECTION-OBJ(T1) = SECTION-OBJ(T2)
                      & COURSE-OBJ(T1) = COURSE-OBJ(T2));
   SECTION-OBJ(CREATE-TAUGHT-AS-LINK(T, S, C)) = S;
   COURSE-OBJ(CREATE-TAUGHT-AS-LINK(T, S, C)) = C;
   ATTR-EQUAL(NEW-TAUGHT-AS-LINK(T, S, C), (CREATE-TAUGHT-AS-LINK(T, S, C)))
end-link


association TAUGHT-AS is
 class-sort TAUGHT-AS
 link-class TAUGHT-AS-LINK
 sort SECTION-CLASS, COURSE-CLASS
 operations
   IMAGE: TAUGHT-AS, SECTION -> COURSE-CLASS
   IMAGE: TAUGHT-AS, COURSE -> SECTION-CLASS
 events NEW-TAUGHT-AS: -> TAUGHT-AS
 axioms
   NEW-TAUGHT-AS() = EMPTY-SET;
   fa ((T: TAUGHT-AS), S: SECTION) SIZE(IMAGE(T, S)) = 1;
   fa ((S: TAUGHT-AS), (T: SECTION), A: COURSE)
      (ex (C: TAUGHT-AS-LINK) in(C, S) & TAUGHT-AS-OBJ(C) = T & TAUGHT-AS-OBJ(C) = A)
          <=> in(A, image(S, T));
   fa ((S: TAUGHT-AS), (T: SECTION), A: COURSE)
      (ex (C: TAUGHT-AS-LINK) in(C, S) & TAUGHT-AS-OBJ(C) = A & TAUGHT-AS-OBJ(C) = T)
          <=> in(T, image(S, A))
end-association


link OFFERING-LINK is
 class-sort OFFERING-LINK
 sort QUARTER, COURSE
 operations ATTR-EQUAL: OFFERING-LINK, OFFERING-LINK -> BOOLEAN
 attributes
   COURSE-OBJ: OFFERING-LINK -> COURSE
   QUARTER-OBJ: OFFERING-LINK -> QUARTER
 methods CREATE-OFFERING-LINK: COURSE, QUARTER -> OFFERING-LINK
 events NEW-OFFERING-LINK: COURSE, QUARTER -> OFFERING-LINK
 axioms
   ATTR-EQUAL(O1, O2) <=> (QUARTER-OBJ(O1) = QUARTER-OBJ(O2)
                          & COURSE-OBJ(O1) = COURSE-OBJ(O2));
   QUARTER-OBJ(CREATE-OFFERING-LINK(O, Q, C)) = Q;
   COURSE-OBJ(CREATE-OFFERING-LINK(O, Q, C)) = C;
   ATTR-EQUAL(NEW-OFFERING-LINK(O, Q, C), (CREATE-OFFERING-LINK(O, Q, C)))
end-link


association OFFERING is
 class-sort OFFERING
 link-class OFFERING-LINK
 sort QUARTER-CLASS, COURSE-CLASS
 operations
   IMAGE: OFFERING, QUARTER -> COURSE-CLASS
   IMAGE: OFFERING, COURSE -> QUARTER-CLASS
 events NEW-OFFERING: -> OFFERING
 axioms
   NEW-OFFERING() = EMPTY-SET;
   fa ((Q: OFFERING), (O: QUARTER), A: COURSE)
```

```
      (ex (C: OFFERING-LINK) in(C, Q) & OFFERING-OBJ(C) = O & OFFERING-OBJ(C) = A)
          <=> in(A, image(Q, O));
   fa ((Q: OFFERING), (O: QUARTER), A: COURSE)
      (ex (C: OFFERING-LINK) in(C, Q) & OFFERING-OBJ(C) = A & OFFERING-OBJ(C) = O)
          <=> in(O, image(Q, A))
end-association


link TEACHES-LINK is
 class-sort TEACHES-LINK
 sort COURSE, FACULTY
 operations ATTR-EQUAL: TEACHES-LINK, TEACHES-LINK -> BOOLEAN
 attributes
   FACULTY-OBJ: TEACHES-LINK -> FACULTY
   COURSE-OBJ: TEACHES-LINK -> COURSE
 methods CREATE-TEACHES-LINK: FACULTY, COURSE -> TEACHES-LINK
 events NEW-TEACHES-LINK: FACULTY, COURSE -> TEACHES-LINK
 axioms
   ATTR-EQUAL(T1, T2) <=>(COURSE-OBJ(T1) = COURSE-OBJ(T2)
                         & FACULTY-OBJ(T1) = FACULTY-OBJ(T2));
   COURSE-OBJ(CREATE-TEACHES-LINK(T, C, F)) = C;
   FACULTY-OBJ(CREATE-TEACHES-LINK(T, C, F)) = F;
   ATTR-EQUAL(NEW-TEACHES-LINK(T, C, F), (CREATE-TEACHES-LINK(T, C, F)))
end-link


association TEACHES is
 class-sort TEACHES
 link-class TEACHES-LINK
 sort COURSE-CLASS, FACULTY-CLASS
 operations
   IMAGE: TEACHES, COURSE -> FACULTY-CLASS
   IMAGE: TEACHES, FACULTY -> COURSE-CLASS
 events NEW-TEACHES: -> TEACHES
 axioms
   ATTR-EQUAL(T1, T2) <=>(AVERAGE-SIZE(T1) = AVERAGE-SIZE(T2)
                         & TIMES-TAUGHT(T1) = TIMES-TAUGHT(T2));
   NEW-TEACHES() = EMPTY-SET;
   fa ((C: TEACHES), (T: COURSE), A: FACULTY)
      (ex (F: TEACHES-LINK) in(F, C) & TEACHES-OBJ(F) = T & TEACHES-OBJ(F) = A)
          <=> in(A, image(C, T));
   fa ((C: TEACHES), (T: COURSE), A: FACULTY)
      (ex (F: TEACHES-LINK) in(F, C) & TEACHES-OBJ(F) = A & TEACHES-OBJ(F) = T)
          <=> in(T, image(C, A))
end-association


link ADVISES-LINK is
 class-sort ADVISES-LINK
 sort FACULTY, STUDENT
 operations ATTR-EQUAL: ADVISES-LINK, ADVISES-LINK -> BOOLEAN
 attributes
   STUDENT-OBJ: ADVISES-LINK -> STUDENT
   FACULTY-OBJ: ADVISES-LINK -> FACULTY
 methods CREATE-ADVISES-LINK: STUDENT, FACULTY -> ADVISES-LINK
 events NEW-ADVISES-LINK: STUDENT, FACULTY -> ADVISES-LINK
 axioms
   ATTR-EQUAL(A1, A2) <=> (FACULTY-OBJ(A1) = FACULTY-OBJ(A2)
                           & STUDENT-OBJ(A1) = STUDENT-OBJ(A2));
   FACULTY-OBJ(CREATE-ADVISES-LINK(A, F, S)) = F;
   STUDENT-OBJ(CREATE-ADVISES-LINK(A, F, S)) = S;
   ATTR-EQUAL(NEW-ADVISES-LINK(A, F, S), (CREATE-ADVISES-LINK(A, F, S)))
end-link


association ADVISES is
 class-sort ADVISES
 link-class ADVISES-LINK
 sort FACULTY-CLASS, STUDENT-CLASS
 operations
   IMAGE: ADVISES, FACULTY -> STUDENT-CLASS
   IMAGE: ADVISES, STUDENT -> FACULTY-CLASS
 events NEW-ADVISES: -> ADVISES
 axioms
   NEW-ADVISES() = EMPTY-SET;
   fa ((F: ADVISES), (A: FACULTY), B: STUDENT)
      (ex (S: ADVISES-LINK) in(S, F) & ADVISES-OBJ(S) = A & ADVISES-OBJ(S) = B)
          <=> in(B, image(F, A));
 fa ((F: ADVISES), (A: FACULTY), B: STUDENT)
     (ex (S: ADVISES-LINK) in(S, F) & ADVISES-OBJ(S) = B & ADVISES-OBJ(S) = A)
          <=> in(A, image(F, B))
end-association


link MEMBER-OF-LINK is
 class-sort MEMBER-OF-LINK
 sort STUDENT, A-CLASS
 operations ATTR-EQUAL: MEMBER-OF-LINK, MEMBER-OF-LINK -> BOOLEAN
 attributes
   A-CLASS-OBJ: MEMBER-OF-LINK -> A-CLASS
   STUDENT-OBJ: MEMBER-OF-LINK -> STUDENT
 methods CREATE-MEMBER-OF-LINK: A-CLASS, STUDENT -> MEMBER-OF-LINK
 events NEW-MEMBER-OF-LINK: A-CLASS, STUDENT -> MEMBER-OF-LINK
 axioms
   ATTR-EQUAL(M1, M2) <=> (STUDENT-OBJ(M1) = STUDENT-OBJ(M2)
                           & A-CLASS-OBJ(M1) = A-CLASS-OBJ(M2));
   STUDENT-OBJ(CREATE-MEMBER-OF-LINK(M, S, A)) = S;
   A-CLASS-OBJ(CREATE-MEMBER-OF-LINK(M, S, A)) = A;
   ATTR-EQUAL(NEW-MEMBER-OF-LINK(M, S, A), (CREATE-MEMBER-OF-LINK(M, S, A)))
end-link


association MEMBER-OF is
 class-sort MEMBER-OF
 link-class MEMBER-OF-LINK
 sort STUDENT-CLASS, A-CLASS-CLASS
 operations
   IMAGE: MEMBER-OF, STUDENT -> A-CLASS-CLASS
   IMAGE: MEMBER-OF, A-CLASS -> STUDENT-CLASS
 events NEW-MEMBER-OF: -> MEMBER-OF
 axioms
   NEW-MEMBER-OF() = EMPTY-SET;
   fa ((M: MEMBER-OF), S: STUDENT) SIZE(IMAGE(M, S)) = 1;
   fa ((S: MEMBER-OF), (M: STUDENT), B: A-CLASS)
       (ex (A: MEMBER-OF-LINK) in(A, S) & MEMBER-OF-OBJ(A) = M & MEMBER-OF-OBJ(A) = B)
            <=> in(B, image(S, M));
 fa ((S: MEMBER-OF), (M: STUDENT), B: A-CLASS)
     (ex (A: MEMBER-OF-LINK) in(A, S) & MEMBER-OF-OBJ(A) = B & MEMBER-OF-OBJ(A) = M)
          <=> in(M, image(S, B))
end-association


aggregate DOMAIN-THEORY-AGGREGATE is
 nodes FACULTY-WORKLOAD-AGGREGATE, FACULTY-WORKLOAD-CLASS, FACULTY-WORKLOAD
 arcs FACULTY-WORKLOAD-AGGREGATE -> FACULTY-WORKLOAD: {},
      FACULTY-WORKLOAD -> FACULTY-WORKLOAD-CLASS: {}
end-aggregate
```

Bibliography

1. Ackroyd, M. and D. Daum. "Graphical Notation for Object-Oriented Design and Programming," *Journal of Object-Oriented Programming*, *3*(5):18–28 (January 1991).
2. Adler, Mike. "An Algebra for Data Flow Diagram Process Decomposition," *IEEE Transactions on Software Engineering*, *14*(2):169–183 (February 1988).
3. Aho, Alfred and others. *Compilers: Principles, Techniques, and Tools*. Reading, Massachusetts: Addison-Wesley, 1986.
4. ALabiso, B. "Transformation of Data Flow Analysis Models to Object-Oriented Design." *Proceedings of OOPSLA '88 Conference*. 335–353. September 1988.
5. Alencar, Antonio J. and Joseph A. Gougen. "OOZE: An Object Oriented Z Environment." *ECOOP '91 Proceedings*. 180–199. New York: Springer-Verlag, July 1991.
6. Alencar, Antonio J. and Joseph A. Gougen. "OOZE." *Object Orientation in Z* edited by Rosalind Barden Susan Stepney and David Cooper, 78–94, Springer-Verlag, 1992.
7. Alencar, Antonio J. and Joseph A. Gougen. "Specification in OOZE with Examples." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Houghton, 158–183, Prentice-Hall, 1994.
8. Alexander, Perry. "Best of Both Worlds," *IEEE Potentials*, 29–32 (December 1995).
9. Arango, Guillermo. "Domain Analysis - From Art Form to Engineering Discipline." *Proceedings of the 5th International Workshop on Software Specifications and Design*. 152–159. 1989.
10. Babin, G. and others. "Specification and Design of Transactions in Information Systems: A Formal Approach," *IEEE Transactions on Software Engineering*, *17*(8):814–829 (August 1991).
11. Bailin, S.C. "An Object-Oriented Requirements Specification Method," *Communications of the ACM*, *32*(5):608–623 (May 1989).
12. Bar-David, Tvsi. "Practical Consequences of Formal Defintions of Inheritance," *Journal of Object-Oriented Programming*, *5*(4):43–49 (July/August 1992).
13. Booch, Grady. *Object-Oriented Design with Applications*. Benjamin Cummings, 1991.
14. Bourdeau, Robert H. and Betty H.C. Cheng. "A Formal Semantics for Object Model Diagrams," *IEEE Transactions on Software Engineering*, *21*(10):799–821 (October 1995).
15. Brumbaugh, David E. *Object-Oriented Development: Building Case Tools with C++*. New York: John Wiley and Sons Inc., 1994.
16. Burstall, R. M. and J. A. Goguen. "Putting Theories Together to Make Specifications." *Proceedings. 5th International Joint Conference on Artificial Intelligence*. 1045–1058. Cambridge, MA: MIT, 1977.
17. Carrington, David and others. "Object-Z: An Object-Oriented Extension to Z." *Formal Description Techniques, II: Proceedings of the IFIP Second International Conference on Formal Description Techniques for Distributed Systems and Communications Protocol*. 281–297. Amsterdam: North-Holland, December 1989.
18. Chang, Chin-Lian and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. San Diego, California: Academic Process, Inc., 1973.
19. Coad, Peter and Edward Yourdon. *Object-Oriented Analysis*. Englewood Cliffs, New Jersey: Yourdon Press, 1990.
20. Coad, Peter and Edward Yourdon. *Object-Oriented Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

21. Coleman, Dereck and others. *Object-Oriented Development: The Fusion Method*. Englewood Cliffs, New Jersey: Prentice Hall, 1994.

22. Conger, S. A. and others. "A Structured Stepwise Refinement Method for VDM." *Proceeding of the 8th Annual Conference on Ada Technology*. 311–320. Atlanta GA: ANACOST, Inc., March 1990.

23. Danforth, Scott and Chris Tomlinson. "Type Theories and Object-Oriented Programs," *ACM Computing Surveys*, *20*(1):29–72 (March 1988).

24. Debart, Francoise and others. "Multimodal Logic Programming using Equational and Order-Sorted Logic," *Theoretical Computer Science*, *105*(1):141–166 (1992).

25. DeBellis, Michael and others. *KBSA Concept Demo*. Technical Report RL-TR-93-38, Griffiss Air Force Base, New York: Rome Laboratory, April 1993.

26. D'Ippolito, Richard S. "Using Models in Software Engineering." *Tri-Ada '89*. 256–265. 1989.

27. Fiadeiro, J. and T. Maibaum. "Describing, Structuring and Implementing Objects." *Foundations of Object-Oriented Languages 489*. LNCS, edited by J.W. deBakker and W.P. deRoever, Springer-Verlag, 1990.

28. Fraser, Martin D. and others. "Informal and Formal Requirements Specification Languages: Bridging the Gap," *IEEE Transactions on Software Engineering*, *17*(5):454–466 (May 1991).

29. Fraser, Martin D., et al. "Strategies for Incorporating Formal Specifications," *Communications of the ACM*, *37*(10):74–86 (October 1994).

30. Garlan, David and Mary Shaw. "An Introduction to Software Architecture." *Advances in Software Engineering and Knowledge Engineering 1*, World Scientific Publishing Company, 1993.

31. Geller, J., et al. "Structure and Semantics in OODB Class Specifications," *SIGMOD RECORD*, *20*(4):40–43 (December 1991).

32. Gerken, Mark J. *Specification and Design Theories for Software Architectures*. PhD dissertation, Graduate School of Engineering, Air Force Institute of Technology (AU), 1995.

33. Goguen, J. A. and R. M. Burstall. "Some Fundamental Algebraic Tools for the Semantics of Computation Part I: Comma Categories, Colimits, Signatures and Theories," *Theoretical Computer Science*, *31*:175–209 (1984).

34. Goguen, J. A. and R. M. Burstall. "Some Fundamental Algebraic Tools for the Semantics of Computation Part II: Signed and Abstract Theories," *Theoretical Computer Science*, *31*:263–295 (1984).

35. Goguen, J.A., et al. "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types." *Data Structuring IV*, edited by R.T. Yeh, 80–149, Englewood Cliffs, NJ: Prentice-Hall, 1978.

36. Goguen, Joseph A. "Parameterized Programming," *IEEE Transactions on Software Engineering*, 528–543 (September 1984).

37. Goguen, Joseph A. "Reusing and Interconnecting Software Components," *IEEE Computer*, 16–28 (February 1986).

38. Goguen, Joseph A. and Jose Meseguer. "Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics." *Research Directions in Object-Oriented Programming* edited by Bruce Shriver and Peter Wegner, 417–477, MIT Press, 1987.

39. Goguen, Joseph A. and Jose Meseguer. "Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations," *Theoretical Computer Science*, *105*(2):217–273 (November 1992).

40. Goguen, Joseph A. and Timothy Winkler. *Introducing OBJ3*. Technical Report, 333 Ravenswood Ave, Menlo Park, CA: Computer Science Laboratory SRI International, August 1988.

41. Goldberg, Allen T. "Knowledge-Based Programming: A Survey of Program Design and Construction Techniques," *IEEE Transactions on Software Engineering*, *SE-12*(7):752–768 (July 1986).

42. Green, Cordell and others. "Report on a Knowledge-Based Software Assistant." *Readings in Artificial Intelligence and Software Engineering* edited by C. Rich and R.C. Waters, 377–428, San Mateo, CA: Morgan Kauffman, 1986.

43. Guttag, John and James Horning. *Larch: Languages and Tools for Formal Specification*. New York: Springer-Verlag, 1993.

44. Hall, Anthony. "Seven Myths of Formal Methods," *IEEE Software*, *7*(5):11–19 (September 1990).

45. Harel, David. "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, *8*:231–274 (1987).

46. Harel, David and others. "On the Formal Semantics of Statecharts." *Proceedings of the Symposium on Logic in Computer Science*. 54–64. Ithaca, NY: IEEE Computer Society Press, June 1987.

47. Hopcroft, John E. and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading, Massachusetts: Addison-Wesley, 1979.

48. Iscoe, Neil. "Domain-Specific Reuse: An Object-Oriented and Knowledge-Based Approach." *Tutorial on Software Reuse: Emerging Technology* edited by Will Tracz, IEEE Computer Society Press, 1988.

49. Jing, Ying and others. "A Methodology for High-level Software Specification Construction," *SIGSOFT Software Engineering Notes*, *20*(2):48–54 (1995). April.

50. Jullig, Richard and Yellamraju V. Srinivas. "Diagrams for Software Synthesis." *KBSE '93*. IEEE, 1993.

51. Kang, Kyo C., et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, Software Engineering Institute, Carnegie Mellon University, November 1990.

52. Kemmerer, R. A. "Integrating Formal Methods into the Development Process," *IEEE Software*, 37–50 (September 1990).

53. Kestrel Institute. *KIDS Manual*, September 1991.

54. Kestrel Institute. *Slang Language Manual: Specware Version Core4*, October 1994.

55. Kestrel Institute. *Specware User Manual: Specware Version Core4*, October 1994.

56. Kierburtz, R.B. and others. "Calculating Software Generators from Solution Specifications." *TAPSOFT '95: Theory and Practice of Software Development*. 546–60. Berlin, Germany: Springer-Verlag, May 1995.

57. Kung, C. H. "Conceptual Modeling in the Context of Software Development," *IEEE Transactions on Software Engineering*, *15*(10):1176–1187 (October 1989).

58. Lano, Kevin. "Z++, An Object-Oriented Extension to Z." *Z User Workshop, Oxford 1990* edited by J.E. Nicholls, 151–172, Springer-Verlag, 1990.

59. Lano, Kevin and Howard Haughton. "Reuse and Adaptation of Z Specifications." *Z User Workshop, London 1992* edited by J.P. Bowen and J.E. Nicholls, 62–90, Springer-Verlag, 1992.

60. Lano, Kevin and Howard Houghton. "A Comparative Description of Object-Oriented Specification Languages." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Houghton, 20–54, Prentice-Hall, 1994.

61. Lano, Kevin and Howard Houghton. "Object-oriented Specification Languages in the Software Life Cycle." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Houghton, 55–79, Prentice-Hall, 1994.

62. Lano, Kevin and Howard Houghton. "Specifying a Concept-recognition System in Z++." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Houghton, 137–157, Prentice-Hall, 1994.

63. Lano, Kevin and Mary Tobin. "Specification and Analysis Techniques in Object-oriented Methods." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Houghton, 1–19, Prentice-Hall, 1994.

64. Lano, Kevin C. "Z++." *Object Orientation in Z* edited by Rosalind Barden Susan Stepney and David Cooper, 105–112, Springer-Verlag, 1992.

65. Lin, Captain Catherine J. *Developing a Transformation Methodology From Object-Oriented Domain Models Into Algebraic Specifications.* MS thesis, Graduate School of Engineering, Air Force Institute of Technology (AU), 1994.

66. Liskov, B. and S.Zilles. "Specification Techniques for Data Abstraction," *IEEE Transactions on Software Engineering*, *SE-1*:7–19 (1975).

67. Liskov, Barbara. "Data Abstraction and Hierarchy." *(addendum to) Conference Proceedings, Object Oriented Programming Systems Languages and Applications (OOPSLA).* 1987.

68. Lowry, Michael R. "Software Engineering in the Twenty-First Century," *AI Magazine* (Fall 1992).

69. Lu, Xue-Miao and Tharam S. Dillon. "An Algebraic Theory of Object-Oriented Systems," *IEEE Transactions on Knowedge and Data Engineering*, *6*(3):412–419 (June 1994).

70. Lubars, Mitchell D. "Domain Analysis and Domain Engineering in IDeA." *Domain Analysis and Software Systems Modeling* edited by Rubèn Prieto-Diaz and Guillermo Arango, 163–178, IEEE Press, 1991.

71. MacLane, Saunders and Garrett Birkhoff. *Algebra.* New York, NY: Chelsea Publishing Company, 1993.

72. Martin, James. *Principles of Object-Oriented Analysis and Design.* Englewood Cliffs, New Jersey: Prentice Hall, 1993.

73. Meyer, B. *Object-Oriented Software Construction.* Englewood Cliffs, MJ: Prentice Hall, 1988.

74. Miriyala, K. and M. T. Harandi. "Automatic Derivation of Formal Software Specifications from Informal Descriptions," *IEEE Transactions on Software Engineering*, *17*(10):1126–1142 (October 1991).

75. Mitra, Swapan. "Object-oriented Specification in VDM++." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Houghton, 130–136, Prentice-Hall, 1994.

76. Monarchi, David E. and Gretchen I. Puhr. "A Research Typology for Object-Oriented Analysis and Design," *Communications of the ACM*, *35*(9):35–47 (September 1992).

77. Moore and Balin. "Domain Analysis: Framework for Reuse." *Domain Analysis and Software Systems Modeling* edited by Rubèn Prieto-Diaz and Guillermo Arango, 163–178, IEEE Press, 1991.

78. Neighbors, J. *Software Construction Using Components.* PhD dissertation, Dept. of Information and Computer Science, U. of California, Irvine, 1981.

79. Prieto-Diaz, Rubén. "Domain Analysis for Reusability." *Proceedings of COMPSAC '87*. 23–29. 1987.

80. Prieto-Diaz, Rubén. "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, *15*(2) (April 1990).

81. Rafsanjani, G-H Bagherzadeh and S. J. Colwill. "For Object-Z to C++: A Structural Mapping." *Z User Workshop, London 1992* edited by J.P. Bowen and J.E. Nicholls, 166–179, Springer-Verlag, 1992.

82. Rose, Gordon and Roger Duke. "An Object-Z Specification of a Mobile Phone System." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Houghton, 110–129, Prentice-Hall, 1994.

83. Rumbaugh, James and others. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1991.

84. Shlaer, S. and S.J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, MJ: Prentice Hall, 1988.

85. Silvio Lemos Meira, Ana Lúcia C. Cavalcanti and Cassio Souza Santos. "The Unix Filing System: A MooZ Specification." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Houghton, 80–109, Prentice-Hall, 1994.

86. Smith, Douglas R. "KIDS - A Semi-automatic Program Development System," *IEEE Transactions of Software Engineering*, *16*(9):1024–1043 (September 1990).

87. Smith, Douglas R. "Transformational Approach to Transportation Scheduling." *Proceedings of the 8th Knowledge-Based Software Engineering Conference*. 60–68. IEEE, October 1993.

88. Srinivas, Yellamraju V. *Algebraic Specification: Syntax, Semantics, Structure*. Technical Report, Department of Information and Computer Science, University of California, Irvine: Department of Information and Computer Science, University of California, Irvine, June 1990. TR 90-15.

89. Srinivas, Yellamraju V. *Category Theory Definitions and Examples*. Technical Report, Department of Information and Computer Science, University of California, Irvine: Department of Information and Computer Science, University of California, Irvine, February 1990. TR 90-14.

90. Srinivas, Yellamraju V. "Augmenting Algebraic Specifications with Structured Sorts and Structural Subsorting." *IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi (PROCOMET '94, 6-10 June 1994, San Miniato, Italy)*. 531–550. Elsevier/North-Holland, June 1994.

91. Susan Stepney, Rosalind Barden and David Cooper. *Object Orientation in Z*. Springer-Verlag, 1992.

92. Tao, Yonglei and Chenho Kung. "Formal Definitions and Verification of Data Flow Diagrams," *Journal of Systems and Software*, *16*(1):29–36 (September 1991).

93. Tracz, Will, et al., "A Domain-Specific Software Architecture Engineering Process Outline." In Report: Collected Papers of the Domain-Specific Software Architectures (DSSA) Avionics Domain Application Generation Environment (ADAGE), May 1993.

94. Vazquez, Federico. "An Algebra Approach to the Deduction of Data Flow Diagrams and Object Oriented Diagrams from a Set of Specifications," *OOPS Messenger*, *6*(2):18–27 (April 1995).

95. Wagner, Eric G. "Categorical Semantics, or Extending Data Types to Include Memory." *Recent Trends in Data Type Specification: 3rd Workshop on Theory and Applications of Abstract Data Types*. 1–21. New York: Springer-Verlag, 1985.

96. Wartik, Steve, et al. *Domain Engineering Guidebook.* Technical Report, Defense Technical Information Center, December 1992.

97. Wartik, Steven and Rubén Prieto-Diaz. "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches," *International Journal of Software Engineering and Knowledge Engineering, 2*(3):403–431 (1992).

98. Wasserman, A.I. and others. "The Object-Oriented Structured Design Notation for Software Design Representation," *IEEE Computer*, 50–62 (March 1990).

99. Wegner, Peter. "The Object-Oriented Classification Paradigm." *Research Directions in Object-Oriented Programming* edited by Bruce Shriver and Peter Wegner, 479–560, MIT Press, 1994.

100. Wills, Alan. "Specification in Fresco." *Object Orientation in Z* edited by Rosalind Barden Susan Stepney and David Cooper, 127–135, Springer-Verlag, 1992.

101. Wills, Alan. "Refinement in Fresco." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Houghton, 184–201, Prentice-Hall, 1994.

102. Wing, J. "A Specifier's Introduction to Formal Methods," *IEEE Computer, 23*(9):8–24 (September 1990).

103. Wirfs-Brock, R.J. and others. *Designing Object-Oriented Software.* Englewood Cliffs, NJ: Prentice Hall, 1990.

Vita

Major Edward A. Ingham [redacted]n. He graduated from high school in Edmonds, Washington in June of 1980. He then attended the United States Air Force Academy where he obtained a Bachelor of Science in Engineering Sciences on 30 May 1984. He was commissioned a second lieutenant and attended Undergraduate Pilot Training at Williams AFB, Arizona. After training, he continued at Williams as a first assignment instructor pilot in the T-37. In May of 1988, he upgraded to the F-16C and was assigned to the 14th Fighter Squadron, Misawa Air Base, Japan. He entered the Graduate School of Engineering, Air Force Institute of Technology, in May of 1992. At the Air Force Institute of Technology, Major Ingham completed a Masters of Science in Systems Engineering. He has continued to the present in a Ph.D. program with the department of Electrical and Computer Engineering. Major Ingham and his wife Renee were married in 1985. They have two sons: Connan and Dillon.

[redacted]Dr
[redacted]

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE June 1996	3. REPORT TYPE AND DATES COVERED Doctoral Dissertation
4. TITLE AND SUBTITLE FORMAL TRANSFORMATIONS FROM GRAPHICALLY-BASED OBJECT-ORIENTED REPRESENTATIONS TO THEORY-BASED SPECIFICATIONS		5. FUNDING NUMBERS
6. AUTHOR(S) Scott A. DeLoach, Major, USAF		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology 2750 P Street WPAFB OH 45433-7765		8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/DS/ENG/96-05
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Mr. Glenn Durbin NSA/Y21 9800 Savage Road, Suite 6718 Fort Meade, MD 20755-6718		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited		12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Formal software specification has long been touted as a way to increase the quality and reliability of software; however, it remains an intricate, manually intensive activity. An alternative to using formal specifications is to use graphically-based, semi-formal specifications such as those used in many object-oriented specification methodologies. While semi-formal specifications are generally easier to develop and understand, they lack the rigor and precision of formal specification techniques. The basic premise of this investigation is that formal software specifications can be constructed using correctness preserving transformations from graphically-based object-oriented representations. In this investigation, object-oriented specifications defined using Rumbaugh's Object Modeling Technique (OMT) were translated into algebraic specifications. To ensure the correct translation of graphically-based OMT specifications into their algebraic counterparts, a formal semantics for interpreting OMT specifications was derived and an algebraic model of object-orientation was developed. This model defines how object-oriented concepts are represented algebraically using an object-oriented algebraic specification language O-SLANG. O-SLANG combines basic algebraic specification constructs with category theory operations to capture internal object class structure as well as relationships between classes. Next, formal transformations from OMT specifications to O-SLANG specifications were defined and the feasibility of automating these transformations was demonstrated by the development of a proof-of-concept system.

14. SUBJECT TERMS formal specification languages, algebraic specification languages, specification acquisition, object-oriented models, object-orientation, software synthesis, software engineering			15. NUMBER OF PAGES 390
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL