Transforming Algebraically-Based Object Models

Into a Canonical Form for Design Refinement

THESIS

Charles G. Beem
Captain, USAF

AFIT/GCS/ENG/95D-01

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC QUALITY INSPECTED 1

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/95D-01

Transforming Algebraically-Based Object Models Into a Canonical Form for Design Refinement

THESIS

Charles G. Beem
Captain, USAF

AFIT/GCS/ENG/95D-01

19960207 033

Approved for public release; distribution unlimited

AFIT/GCS/ENG/95D-01

Transforming Algebraically-Based Object Models Into a Canonical Form for Design Refinement

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Charles G. Beem, B.S.C.S.

Captain, USAF

December 6, 1995

Approved for public release; distribution unlimited

Acknowledgements

I would like to thank my advisor, Lieutenant Colonel Paul Bailor, for his guidance and assistance during this research effort. I also wish to thank my committee members, Dr. Thomas Hartrum, Dr. Eugene Santos, and Dr. Bob Shock, for their help and advice.

I would also like to thank some of my fellow classmates for helping me navigate through the AFIT minefield. Special thanks to Vince Hibdon for helping me keep my football/class-work priorities straight; go Big Eight! I also want to thank Shawn Hannan for his sage advice on my presentation; I look forward to working together over the next three years.

I also wish to thank Major Scott DeLoach for his help and advice with O-SLANG and theory-based object models. His contributions were vital to the successful results achieved during this effort.

One of the most important parts of this effort was my family. I would like to thank my wife, Gina, and my children Chuck, Andrew, Kristen, and Sarah, for their support, love, and prayers. Without their help and sacrifices, this would have been an unsurmountable task. I love you guys!

Finally, and most importantly, I wish to thank my Lord and Savior, Jesus Christ, for giving me the strength, courage, and wisdom to get through the last eighteen months. I know that "I can do everything through him who gives me strength."

Charles G. Beem

Table of Contents

		Page
Acknowledgements		ii
List of Figures		viii
List of Tables		xi
Abstract		xii
I.	Introduction	1-1
	1.1 Background	1-1
	1.2 Problem Statement	1-3
	1.3 Scope	1-5
	1.4 Sequence of Presentation	1-5
II.	Literature Review	2-1
	2.1 Introduction	2-1
	2.2 Theory-based Object Models	2-1
	2.2.1 Algebraic Specifications and Theories	2-2
	2.2.2 Rumbaugh's Object Modeling Technique	2-4
	2.2.3 Theory-based Representation of Object Model	2-6
	2.2.4 Theory-based Representation of Dynamic Model	2-10
	2.2.5 Theory-based Representation of Functional Model	2-12
	2.2.6 O-SLANG	2-13
	2.3 Term Rewriting	2-14
	2.3.1 Term Rewriting Systems	2-14
	2.3.2 Basic Definitions in Term Rewriting	2-16
	2.3.3 Types of Rewrite Systems	2-18

Page

2.3.4 Properties of Rewrite Systems 2-20

2.3.5 Example Rewrite System 2-22

2.3.6 Graph Rewriting 2-24

2.4 Summary 2-25

III. Designing a Formal Object Transformation Process 3-1

3.1 Introduction 3-1

3.2 Creating the Unified Framework 3-2

3.3 The Canonical Algebraic Framework Phase 3-5

3.4 Validation Criteria and Domains 3-7

3.4.1 Validation Process 3-7

3.4.2 Bank Domain 3-9

3.4.3 Pump Domain 3-11

3.5 Summary 3-15

IV. Extensions to the Unified Model 4-1

4.1 Introduction 4-1

4.2 Changes to ULARCH 4-1

4.2.1 Changing the Dynamic Theory 4-1

4.2.2 Addition of Link and Association Theories 4-2

4.2.3 Addition of Boolean Attribute 4-3

4.3 Changes to OMT-to-ULARCH Mappings 4-4

4.3.1 Addition of Link and Association Theories 4-4

4.3.2 Representing Aggregation and Attributes 4-7

4.3.3 Representing Single and Multiple Inheritance 4-7

4.4 Changes to State Transition Table Model 4-9

4.5 Summary 4-10

Page

V. Definition of Unified Model to Canonical Model Transformations 5-1

5.1 Introduction 5-1

5.2 LARCH to O-SLANG Transformations 5-2

5.2.1 ObjectTheory Mapping 5-2

5.2.2 StateTheory Mapping 5-2

5.2.3 EventTheory Mapping 5-4

5.2.4 FunctionalTheory Mapping 5-5

5.2.5 LinkTheory Mapping 5-6

5.2.6 AssociationTheory Mapping 5-7

5.2.7 Tuple Mapping 5-9

5.3 State Transition Table to O-SLANG Transformations 5-11

5.3.1 Single and Multiple Event Specifications 5-11

5.3.2 Receive Event and Transition Event Axioms 5-12

5.4 Additional Transformations 5-14

5.4.1 Object Class Specifications 5-15

5.4.2 New Events and Create Methods 5-16

5.4.3 attr-equal Operation 5-17

5.4.4 Additional Axioms 5-18

5.4.5 Aggregate Specification Nodes and Arcs 5-18

5.5 Summary 5-20

VI. Design and Implementation of ULARCH to O-SLANG Transformations . . 6-1

6.1 Introduction 6-1

6.2 Overview of Implementation 6-2

6.2.1 REFINE Language Constructs 6-2

6.2.2 Control Structure 6-3

6.2.3 ULARCH to O-SLANG 6-5

6.2.4 State Transition Table to O-SLANG 6-8

Page

6.2.5 Extra Data Structures 6-8

6.2.6 Post Processing 6-9

6.3 Analysis of Implementation 6-10

6.3.1 Results of the Validation Process 6-10

6.3.2 Rewrite System Properties 6-11

6.4 Summary 6-14

VII. Conclusion and Recommendations 7-1

7.1 Summary of Accomplishments 7-1

7.2 Conclusions 7-2

7.3 Recommendations for Future Research 7-4

7.4 Final Comments 7-7

Appendix A. O-SLANG Domain Model A-1

Appendix B. Tree Manipulations for Rewrite Example B-1

Appendix C. ULARCH for Bank Domain Example C-1

Appendix D. O-SLANG for Bank Domain Example D-1

Appendix E. ULARCH for Pump Domain Example E-1

Appendix F. O-SLANG for Pump Domain Example F-1

Appendix G. User Manual for Formal Object Transformation System . . . G-1

G.1 Introduction G-1

G.2 Refine Files G-1

G.3 User Files G-2

G.4 Sample Session G-3

Bibliography BIB-1

Page

Vita VITA-1

List of Figures

Figure		Page
1.1.	Target Transformation Process	1-4
2.1.	Object Model for Rocket	2-5
2.2.	Dynamic Model for FuelTank	2-6
2.3.	Functional Model for FuelTank	2-7
2.4.	Cust-Acct Association	2-9
2.5.	Communication Theory	2-11
2.6.	Communicating Bank Aggregate Class	2-12
2.7.	Bank Aggregate Morphisms for Withdrawal Event	2-12
2.8.	Confluence/Church-Rosser Properties	2-21
3.1.	Analysis Synthesis Model	3-1
3.2.	Bank Object Model	3-10
3.3.	Account Dynamic Model	3-11
3.4.	Console Dynamic Model	3-12
3.5.	Bank Functional Model	3-12
3.6.	Pump Object Model	3-13
3.7.	Pump Dynamic Model	3-14
3.8.	Pump Functional Model	3-15
4.1.	Traits in original ULARCH	4-2
4.2.	Traits in modified ULARCH	4-2
4.3.	Changes to ULARCH Domain Model	4-3
4.4.	Association and Link in modified ULARCH	4-6
5.1.	Conceptual View of Mappings	5-1
5.2.	ObjectTheory Mappings	5-3

Figure		Page

5.3. ObjectTheory Transformation for Sophisticated Pump 5-3
5.4. StateTheory Mappings 5-4
5.5. StateTheory Transformation for Overdrawn 5-4
5.6. EventTheory Mappings 5-5
5.7. EventTheory Transformation for StartPumpMotor 5-5
5.8. FunctionalTheory Mappings 5-6
5.9. FunctionalTheory Transformation for Credit-Acct 5-6
5.10. LinkTheory Mappings 5-7
5.11. LinkTheory Transformation for Own 5-7
5.12. AssociationTheory Mappings 5-8
5.13. AssociationTheory Transformation for Owns 5-8
5.14. Tuple Mappings 5-9
5.15. Tuple Transformation for Pump ObjectTheory 5-10
5.16. Single Event Mappings 5-12
5.17. Multiple Event Mappings 5-12
5.18. Send Event Transformation for OverHeat 5-13
5.19. ObjectTheory Mappings 5-15
5.20. EventTheory Mappings 5-16
5.21. Object Class Transformation for Pump-Class 5-16
5.22. User Defined "new" Event 5-17
5.23. Default "new" Event 5-17
5.24. Object Communication Mappings - Single Event 5-19
5.25. Object Communication Mappings - Multiple Event 5-19
5.26. Association Mappings 5-19
5.27. Inheritance Mappings 5-20
5.28. Aggregate Transformation for Bank 5-21

6.1. *StateTheory* Transformation 6-5

Figure | Page

6.2. *EventTheory* Transformation 6-6

6.3. *FunctionalTheory* Transformation 6-6

6.4. *LinkTheory* Transformation 6-6

6.5. *AssociationTheory* Transformation 6-6

6.6. *Tuple* Transformation 6-6

7.1. Projected Transformation Process 7-6

A.1. Top Level of O-SLANG Domain Model A-1

A.2. Second Level of O-SLANG Domain Model A-2

A.3. Domain Model for Axioms A-2

B.1. Tree Rewrite Example (Step 1) B-1

B.2. Tree Rewrite Example (Step 2) B-2

B.3. Tree Rewrite Example (Step 3) B-3

B.4. Tree Rewrite Example (Step 4) B-4

B.5. Tree Rewrite Example (Step 5) B-5

B.6. Tree Rewrite Example (Step 6) B-6

List of Tables

Table		Page
2.1.	Rumbaugh to Theory-based Object Model Translation	2-13
3.1.	OMT Mappings to Algebraic and Object-based Frameworks	3-4
3.2.	Rumbaugh to O-SLANG Translation	3-6
3.3.	OMT Coverage of Validation Domains	3-8
4.1.	Original Pump State Transition Table	4-9
4.2.	Modified State Transition Table	4-10
C.1.	Account State Transition Table	C-12
C.2.	Checking State Transition Table	C-12
C.3.	Savings State Transition Table	C-12
C.4.	Console State Transition Table	C-13
E.1.	Clutch State Transition Table	E-8
E.2.	Display State Transition Table	E-8
E.3.	Gun State Transition Table	E-9
E.4.	Holster State Transition Table	E-9
E.5.	Motor State Transition Table	E-9
E.6.	Pump State Transition Table	E-9

AFIT/GCS/ENG/95D-01

Abstract

The understandability of object-oriented design techniques and the rigor of formal methods have improved the state of software development; however, both ideas have limitations. Object-oriented techniques, which are semi-formal, can still result in incorrect designs, while formal methods are complex and require an extensive mathematical background. The two approaches can be coupled, however, to produce designs that are both understandable and verifiable, and to produce executable code. This research proposes an approach where object-oriented models are first represented algebraically in a formal specification language such as LARCH and then transformed into a canonical form suitable for design refinement.

In the canonical form presented in this work, object-oriented models are represented as domain theories consisting of multiple class specifications. Each class specification has sorts, operations (attributes, methods, events, states, state attributes, and operations), and axioms which describe its structure and behavior. The ability to reason about relationships between specifications is handled through the use of category theory operations. Although the canonical form is methodology independent, this work demonstrates the proposed approach on object-oriented models developed using Rumbaugh's Object Modeling Technique. The models are first mapped to LARCH and then translated into the canonical form by a set of rewrite rules. The rewrite rules are shown to produce unique normal forms. The final product is a transformation system which converts object-oriented designs into a canonical form that can be used with a design refinement tool.

Transforming Algebraically-Based Object Models
Into a Canonical Form for Design Refinement

I. Introduction

1.1 Background

As industry continues to push existing computer technology to the limit, the software systems required to support the evolving sophisticated applications become more and more complex. This increasing complexity places an even greater emphasis on the need to accurately specify the desired behavior for a system. Traditionally, a natural language such as English is used to describe the requirements, but this technique is imprecise, error-prone, and often results in the software developer building the wrong product. Object-oriented analysis and design methodologies have become the software development technique of choice in many places throughout industry. Because object-oriented analysis and design techniques model problems in terms of real-world concepts, they greatly improve understanding of the system requirements on the part of both the customer and the implementer ($\text{RBP}^{+}91$). Unfortunately, this method is also informal and subject to errors.

To introduce more rigor into the software development process, formal methods and formal specification languages are being explored as a way to ensure that software specifications are both unambiguous and correct. Because formal specification languages are based on mathematical constructs like predicate logic and set theory (NS91), they provide software engineers with a means for reasoning about designs and, if they are executable,

for testing them before they are implemented; this is the way of the traditional engineer. It is this formal methods approach, coupled with object-oriented analysis and design techniques, that forms the path that the Knowledge-Based Software Engineering (KBSE) group at the Air Force Institute of Technology (AFIT) is exploring.

AFIT's KBSE group is researching the development of a composition system that builds domain-specific applications from new and existing domain models. For a specific problem domain, object-oriented models are created using Rumbaugh's Object Modeling Technique (OMT), translated into a formal specification language (Z or Larch), and then transformed into a common representation model in the REFINE [1] object base. These object-based models can then be used to produce executable specifications. (Bai94)

In 1994, Lin and Wabiszewski developed a formalized object transformation process. Lin employed a theory-based approach using Larch (Lin94), while Wabiszewski pursued a model-based approach using Z (Wab94). First, they created and validated mappings from the OMT models to Larch and Z. Once these mappings were validated using existing problem domain OMT models, they developed parsers for their respective formal specification languages. An analysis of the abstract syntax trees produced by the parsers revealed some commonalities between the two languages' representations of OMT. Capitalizing on these commonalities, Lin and Wabiszewski then created a unified model which unified the languages at a high level. Although they concluded that theory-based and model-based specification languages have a common set of constructs that can be used to build a canonical framework for formalizing object models, they stopped short of creating a true canonical model which captures the essence of object-oriented models in a language

[1] REFINE is a wide-spectrum language that is part of the SOFTWARE REFINERY^{TM} environment.

independent form. There is a limit to the degree of unification which their unified framework provides. The common constructs are limited to a few shared object classes. Objects below the signature declaration, external reference, and axiom objects, are depicted as specialized object classes due to the differences in the syntax of the two languages. To eliminate the language specific portions, transformations can be used to manipulate the unified model into a canonical form, i.e. a more general model which represents the same information. Before these transformations can be built, however, the canonical model itself must be defined.

One of the benefits of having a unified model is that it creates a layer of abstraction between the front-end and back-end of the composition system. In order to incorporate new specification languages into the composition system, just create new front-end transformations with the unified model as the target representation. To use a different theorem prover, design refinement mechanism, or different application, simply create back-end transformations with the unified model as the source representation. The key is that the unified model must be independent of the source specification languages and the target applications.

1.2 Problem Statement

An abstract framework that unifies the theory-based and model-based approaches should be a common base language with few, if any, dialects. The framework provided by Lin and Wabiszewski does not have this characteristic, as the languages were not reduced to their most significant form. Also, the framework should be independent of object-oriented methodology. This would provide more flexibility in the selection of modeling methodology,

and would open up the composition system for use by a larger group of potential users. Lin and Wabiszewski's unified model does not provide this methodology independence, since it is heavily influenced by Rumbaugh's OMT methodology. Figure 1.1 shows the language specific portions of the unified model, *UZed* and *U*LARCH, being transformed to the chosen canonical model, O-SLANG. As will be discussed in Chapter II and Chapter III, O-SLANG is an algebraic specification language which captures the notions of object-oriented models in a methodology-independent way. This language will also be a product of DeLoach's translation system (DeL95a), which translates the semi-formal object-oriented Rumbaugh OMT diagrams of a system into formal O-SLANG system specifications.

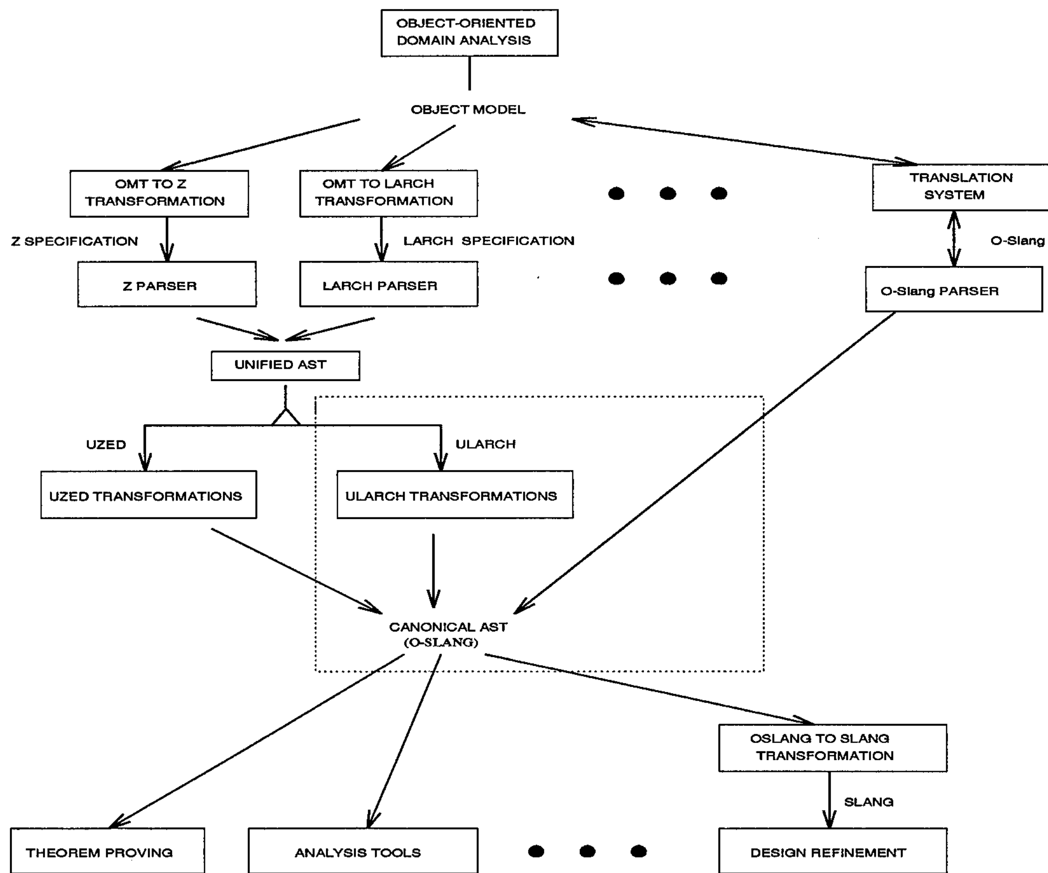

Figure 1.1 Target Transformation Process

Problem Statement:

Define a formal object transformation process by creating a canonical algebraic model to represent general object-oriented models and by using term rewriting techniques to develop transformations from LARCH *specifications to the canonical model.*

1.3 Scope

This work focused on extending the unified model for both ULARCH and UZED; however, time constraints prevented the implementation of transformations from both languages to the canonical model. As such, transformations are only defined and implemented from ULARCH. This corresponds to the portion of Figure 1.1 that is enclosed in the dashed box.

1.4 Sequence of Presentation

The remainder of this thesis is organized as follows:

Chapter II contains a review of theory-based object models followed by a discussion of Rumbaugh's OMT. Also, a review of current literature on term rewrite systems and term rewriting techniques is presented.

Chapter III outlines the specific design for a unified model to canonical model compiler which is based on Lin's ULARCH to REFINE compiler. Also included is a more detailed discussion of Lin and Wabiszewski's work on the formalized object-based composition system. Next, the canonical model (O-SLANG) is presented, thus completing the first phase in

extending the formal object transformation process. Finally, the design validation criteria and example problem domains are identified.

Chapter IV presents a discussion of the second phase of extending the formal object transformation process: making changes to Lin and Wabiszewski's unified model.

Chapter V defines the transformations from ULARCH to O-SLANG.

Chapter VI contains a design and implementation of the transformations from ULARCH to O-SLANG, as well as an analysis of the implementation.

Chapter VII presents general conclusions and recommendations for future research.

Several appendices are also included in order to provide additional information about the extended formal object transformation process. Appendix A contains the domain model for the object-oriented specification language, O-SLANG. Appendix B contains figures relating to a term rewriting example presented in Chapter II. Appendix C and Appendix E contain the ULARCH traits and state transition tables for the *Bank* and *Pump* examples, respectively, while Appendix D and Appendix F contain the O-SLANG produced by the compiler for the two examples. Finally, Appendix G contains a User's Manual for executing the unified model to canonical model compiler.

II. Literature Review

2.1 Introduction

In order to help reach the research goals outlined in Chapter I, this literature review explores research and information needed to extend Lin and Wabiszewski's formal object transformation process. First, Section 2.2 provides a discussion of theory-based object models, and introduces O-SLANG. A brief review of Rumbaugh's Object Modeling Technique(OMT) is also included since it is the object-oriented methodology on which Lin and Wabiszewski's unified model is based. Finally, Section 2.3 explores term rewriting and the feasibility of using term rewriting techniques to create a canonical model.

2.2 Theory-based Object Models

An area that is fast gaining attention in the software development world is the use of algebraic theories to represent software engineering knowledge. Most notably, the Kestrel Interactive Development System (KIDS) uses theory-based specifications as a foundation for software synthesis (Smi90). The strength of such an approach is that it provides a formal foundation for software development that is based on well-founded algorithm theories such as divide-and-conquer and global search. It also serves as a solid framework for reuse of specifications and designs and for the establishment of software engineering technology bases. However, their is one drawback to the theory-based approach. It represents a significant change in the way software is specified and the corresponding executable programs are produced; thus, there is a large learning curve to overcome (Bai95). To help overcome this drawback, transformations can be defined to create theory-based

algebraic specifications from object-oriented models. These transformations can be automated, allowing domain and system designers to build systems using the conceptually simpler object-oriented representation (DeL95a). This section provides background information on theories and Rumbaugh's OMT methodology, and then outlines the creation of theory-based object models from object-oriented models represented using OMT. A description of O-SLANG, an object-oriented extension of the algebraic specification language SLANG, is also presented.

2.2.1 Algebraic Specifications and Theories. An algebraic specification consists of *sorts*, *operations* over those sorts, and a set of *axioms* which describe the behavior of the operations (GH93). Sorts are collections of values. Along with the associated operations, they make up the *signature*, which defines the structure of the algebraic specification; the semantics are defined by the axioms. An algebraic specification is a *theory presentation* (Bai95), and a *theory* is the set of all assertions that can be derived from the axioms of the specification (GH93). It is important to note that a specification is merely a description of many possible valid implementations. An implementation which ensures that all of the axioms are satisfied is called a *model* (GH93).

The idea of creating theory-based algebraic specifications has two goals: modeling system behavior using signatures and axioms, and composition of larger specifications from smaller specifications (DBH95). As described above, the signature of a specification and its axioms describe its structure and semantics, i.e. they describe the *internal* behavior. In order to create larger specifications from smaller ones, it is necessary to be able to reason about relationships *between* the specifications; *category theory* is the mathematical

theory that can be used to describe these relationships. In category theory, a category is made up of a collection of *C-objects* and *C-arrows* between objects. Each object has a C-arrow to itself (i.e. reflexive). Also, arrows are composable and the composition operation is associative. In a category of algebraic specifications, the C-arrows are *specification morphisms.* Basically, a specification morphism consists of two functions which map sorts and operations in one specification to sorts and operations in another. These functions must ensure that all of the axioms in the first specification are theorems in the second. Another operation that is important to creating large specifications from small ones is the *colimit* operation. From an existing set of specifications, the colimit operation creates a new specification consisting of the shared union of all of the sorts and operations in the original specifications. This new specification, called the *colimit specification*, is defined by specification morphisms from each original specification to the colimit specification (DBH95).

Together, specification morphisms and the colimit operation make up a basic toolset for building specifications. Using these tools, there are several ways to build specifications (DBH95):

1. Create a specification by defining a signature and a set of axioms
2. Create a colimit specification using the colimit operation
3. Use a specification morphism to translate a specification
4. Parameterize a specification
5. Build a specification from features in other specifications

2.2.2 Rumbaugh's Object Modeling Technique. OMT describes an application domain by using three different models: the object model, dynamic model, and functional model. Usually, a complete description of an application domain requires the creation of all three models. While the object model is the foundation, the models are "orthogonal parts of the description of a complete system and are cross-linked." (RBP^{+}91)

The object model captures the structure of an application domain by depicting the objects in the domain, attributes and operations which characterize the objects, and the relationships between the objects. The model consists of object diagrams which are graphs whose nodes represent classes of objects and whose arcs represent the relationships among the classes (RBP^{+}91). Three important relationships between classes are association, inheritance, and aggregation. Associations are templates that define what classes of objects may be connected. Essentially, they are a group of links that have a common structure and meaning. A link is a physical or conceptual connection between instances of objects. An association may also define association attributes, which are attributes that do not belong to any of the objects involved in a link, but exist only because of the link between objects. Inheritance represents the "is a" relationship between a class of objects and some subclass. This is a generalization-specialization relationship in which a subclass inherits all of the attributes and operations of some parent class. The subclass may add additional attributes and operations of its own; this is specialization. Aggregation represents the "is composed of" relationship between a class of objects and its components. Aggregation is essential for modeling systems that are formed by combining subsystems (DBH95). Figure 2.1 shows an example of a simple object model for a rocket. Aggregation is represented by the diamond shape on the arcs. In the example, a rocket is composed of an airframe, two

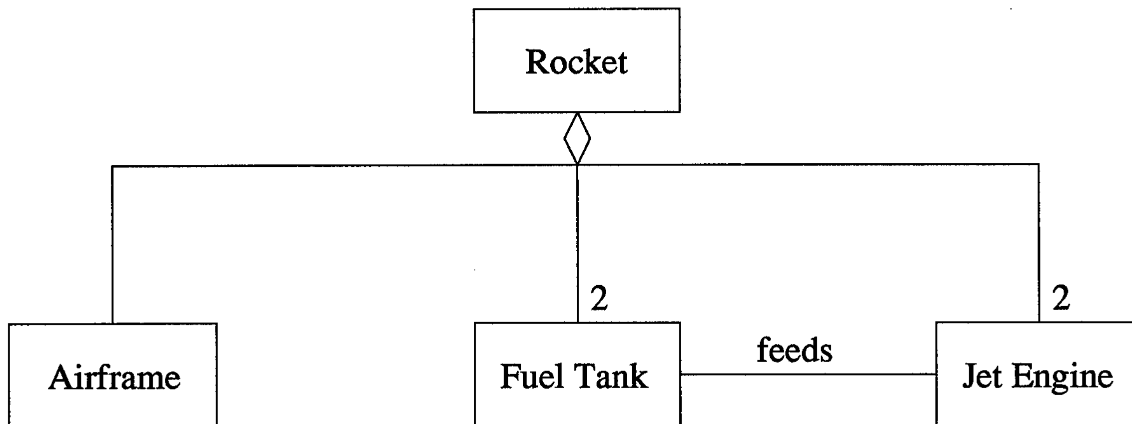

Figure 2.1 Object Model for Rocket

fuel tanks, and two jet engines. Feeds is a simple association between fuel tank objects and jet engine objects; a fuel tank feeds a jet engine.

The dynamic model describes the reactive, or event driven, aspects of the application domain. It consists of state diagrams, which are graphs whose nodes represent states and whose arcs represent transitions between states. The transitions are caused by events, which are represented as labels on the arcs. An object's state is an abstraction of the current values of its attributes. An event is an external stimulus that causes an object to react in a certain way. Figure 2.2 shows an example of a dynamic model for a fuel tank. States are shown in boldface and events are in italics.

The functional model captures the data transformations in the application domain. These data transformations may be operations defined in the object model or actions defined in the dynamic model. The functional model describes how an object's output values are derived from its input values. The model consists of data flow diagrams, which are graphs whose nodes represent processes and whose arcs represent data flow (RBP+91). Figure 2.3 shows a partial example of a functional model for a fuel tank object.

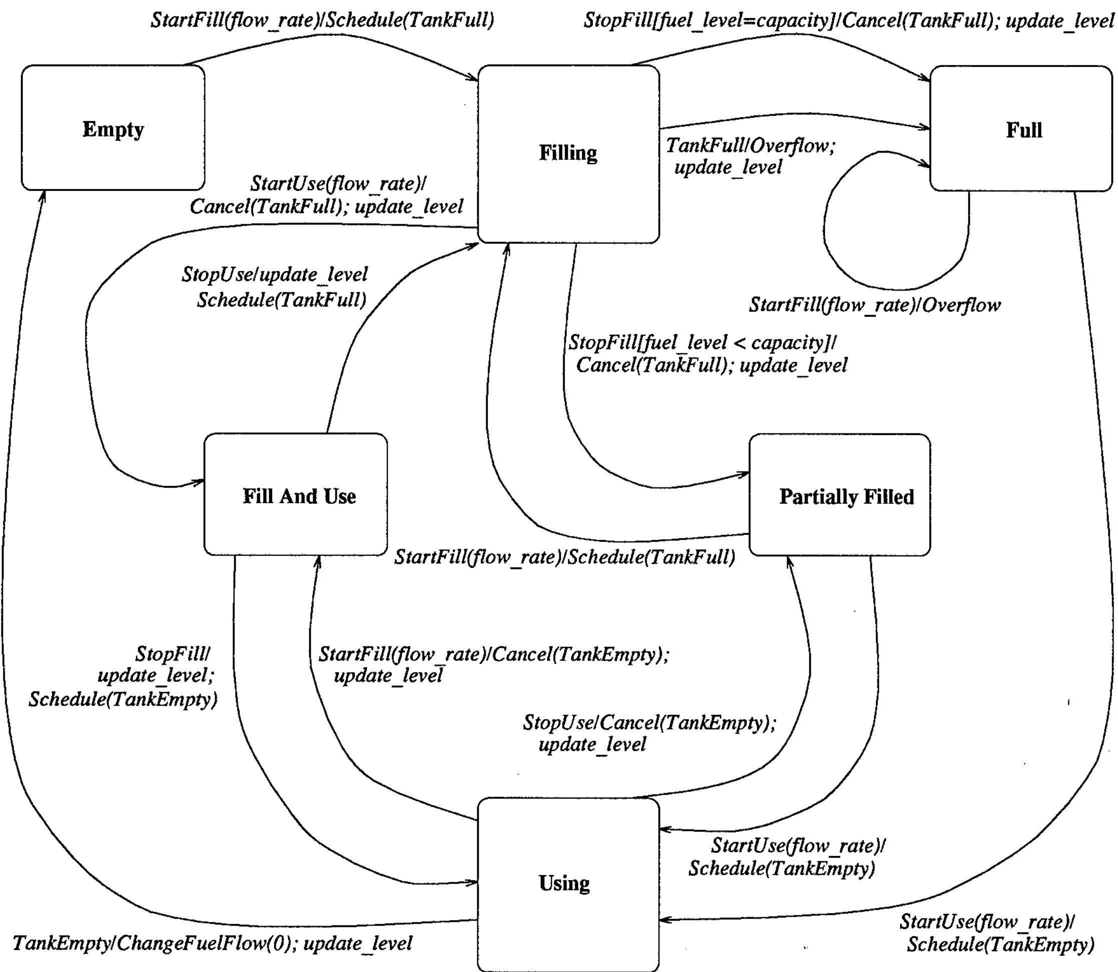

Figure 2.2 Dynamic Model for FuelTank

2.2.3 Theory-based Representation of Object Model. The first step in describing a theory-based representation of an object model is to define an object class. An object class is a theory presentation which represents five parts : class-sort, other sorts referenced in the theory, attributes, methods, and events. More formally (DeL95b):

Definition 2.2.1 Object Class - *A class, C, is a signature, $\Sigma = < S, \Omega >$ and a set of axioms, Φ, over Σ (i.e., a theory presentation, or specification) where*

$$S = \text{a set of sorts including the class sort}$$
$$\Omega = \text{a set of operations over } S \text{ representing attributes, methods, and events}$$
$$\Phi = \text{a set of axioms over } \Sigma$$

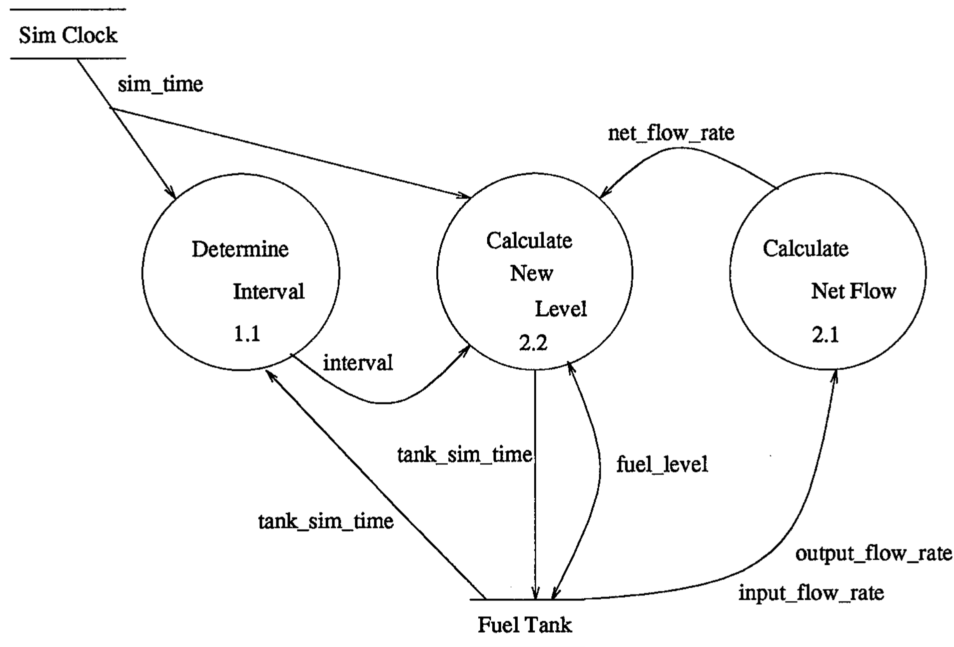

Figure 2.3 Functional Model for FuelTank

A *class-sort* is a distinguished sort which is a set that contains all of the possible names for objects in the object class. An object class can have many instances, each of which has a unique name from the class-sort. Another distinguished sort is the *state sort*, which is the set of all possible class states. *Attributes* are operations that take in an object in the class and return the value of a state component. Attributes can only return information, they can never modify an object. A distinguished set of attributes are the *state attributes*, that are used to obtain the current value in a state sort. To modify attributes, *methods* are used. Methods are operations which modify none, some, or all of an object's attribute values. State attributes can only be modified by *events* (See Section 2.2.4). Axioms are used to define the semantics of an object class. The axioms are usually defined by describing the effect of applying the object's methods on the object's attributes (DeL95b).

The next step in describing a theory-based representation of an object model is to define an object instance.

Definition 2.2.2 Object Instance - *An object, o, is a tuple, $o = < i, CT, \pi >$ where i is a unique name from the class-sort, CT is the class type, and π is a set of variables indexed on attributes in CT $\{a_1, a_2, ...a_n\}$ representing the state of the object.*

The unique name, *i*, is assigned when the object instance is created and does not change over the life of the object. The only part of the object instance that can be modified is its state, represented by the set of variables π (DeL95b).

The final step in describing a theory-based representation of an object model is to define association, inheritance, and aggregation. As mentioned in Section 2.2.2, association is an important relationship between object classes. An association is formally defined as (DBH95):

Definition 2.2.3 Association - *An association is defined as a tuple $A = < \alpha, \lambda >$, where α is an object class whose class sort is a set of the class sort of λ, and λ is a class with two or more object-valued attributes* [1].

Figure 2.4 (DBH95) shows the object classes which represent the *Cust-Acct* association. The class *CA-Link* represents λ in the association. It has two object-valued attributes, *customer* and *account*, as well as a method for creating new instances of the association. The class *Cust-Acct* defines a set of *CA-Link* objects, and the sorts *Accts* and *Custs* are sets of *Acct* and *Cust* objects. The axioms in *Cust-Acct* define the multiplicity relationship between customers and accounts. Each customer can have one or more accounts, while each account belongs to only one customer. If an association involves more than two classes, the relationship can be captured by adding additional object-valued attributes (DBH95).

[1] *Object-valued attributes* are attributes that return references to other objects.

```
link CA-Link is
  class-sort CA-Link
  sorts Cust, Acct
  operations
    attr-equal: CA-Link, CA-Link -$>$ Boolean
  attributes
    customer: CA-Link -$>$ Cust
    account: CA-Link -$>$ Acct
  methods
    create-ca-link: Cust, Acct -$>$ CA-Link
  events
    new-ca-link: Cust, Acct -$>$ CA-Link
  axioms
    attr-equal(c1, c2) =>
      customer(c1) = customer(c2) &
      account(c1) = account(c2);
    customer(create-ca-link(c, a)) = c;
    account(create-ca-link(c, a)) = a;
    attr-equal(new-ca-link(c, a), create-ca-link(c, a))
end-link

association Cust-Acct is
  link-class CA-Link
  import Custs, Accts
  class-sort Cust-Acct
  methods
    create-cust-acct: -$>$ Cust-Acct
    image: Cust-Acct, Cust -$>$ Accts
    image: Cust-Acct, Acct -$>$ Custs
  events
    new-cust-acct: -$>$ Cust-Acct
  axioms
    new-cust-acct() = create-cust-acct();
    create-cust-acct() = empty-set;
    size(image(ca, c)) >= 1;
    size(image(ca, a)) = 1;
... (definition of image operations) ...
end-link
```

Figure 2.4 Cust-Acct Association

Another important relationship between object classes is inheritance. Stated formally (DeL95b):

Definition 2.2.4 Inheritance - *A class D is said to inherit from a class C if there exists a specification morphism from C to D such that the class-sort of D is a subsort of the class-sort of C.*

In other words, all of the sorts and operations from class C are embedded in class D, and the class-sort of D is defined as a subsort of the class-sort of C. The *subsort* relationship among sorts is analogous to the subset relationship between sets. The subsort operator $<$ defines a subset relationship so that for any two sorts A and B, $A < B \Rightarrow A \subseteq B$ (DBH95).

The final important relationship between object classes mentioned in Section 2.2.2 is aggregation. Aggregation is defined formally as (DeL95b):

Definition 2.2.5 Aggregation - *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 referencing each class in $(D_1..D_n)$.*

The colimit operation provides the capability to unify sorts and operations that are defined in different classes and associations. Taking the colimit of a number of class specifications creates an aggregate class that specifies system or subsystem level functionality (DBH95).

2.2.4 Theory-based Representation of Dynamic Model. In defining a theory-based representation of a dynamic model, it is necessary to express the concepts of events and state transitions algebraically. This amounts to describing how objects communicate with each other. Each object is only aware of certain events that it must send. In essence, the events are broadcast to the entire system. For each send event, an operation signature must be defined that maps to a method in some anonymous object class. The anonymous

class-sort and its associated operations are defined in a separate specification called a *communication theory* (DBH95, DeL95b).

Definition 2.2.6 Communication Theory - *A communication theory consists of a class-sort, parameter sorts, and an event signature which are mapped via signature morphisms to sorts and events in the generating and receiving classes.*

The class-sort represents the class-sort of the objects being communicated with. The parameter sorts must be mapped to compatible sorts in the sending and receiving classes. The event signature maps to an event in the receiving class which has the same number of parameters as defined in the communication theory. Once the sorts and events have been mapped under signature morphisms, it is necessary to unify them so that invoking an event in the sending class causes a corresponding invocation in the receiving class. This unification is accomplished via the colimit of the sending and receiving classes, the communication theory, and the signature morphisms (DeL95b). Consider the example where a *Console* in a *Bank* sends a *Withdrawal* event (shown in Figure 2.5) to an *Account*. Figure 2.7 shows the morphisms required for the send event, and Figure 2.6 shows the *Bank* aggregate which defines the colimit operation for the bank which is made up of *Console* and *Account* objects. Recall from section 2.2.1 that a category is made up of *C-objects* and *C-arrows*. The nodes in Figure 2.6 represent C-objects, while the arcs are C-arrows.

```
event Withdrawal is
  class-sort Withdrawal
  sorts Account, Amnt
  events
    Withdrawal: Withdrawal, Account, Amnt -> Withdrawal
end-event
```

Figure 2.5 Communication Theory

```
aggregate Bank is
  nodes Integer, Set-1: Set, Set-2: Set, Account-Class,
        Console-Class, Withdrawal
  arcs  Set-1 -> Account-Class: {Set -> Account-Class, E -> Account},
        Set-2 -> Console-Class: {Set -> Console-Class, E -> Console},
        Integer -> Set-1: {},
        Integer -> Set-2: {},
        Withdrawal -> Console-Class: {},
        Withdrawal -> Account-Class: {}
end-aggregate
```

Figure 2.6 Communicating Bank Aggregate Class

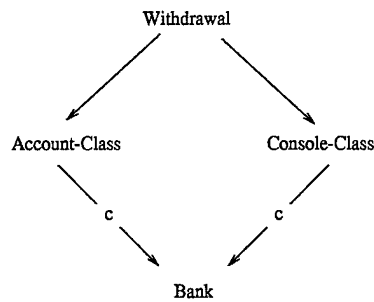

Figure 2.7 Bank Aggregate Morphisms for Withdrawal Event

2.2.5 Theory-based Representation of Functional Model. To define a theory-based representation of the functional model, three things need to be addressed: processes, data flow, and data stores. The processes, or functions, in the functional model correspond to the actions described in the dynamic model. These processes are defined as operations, i.e. methods, in an object class. The behavior of the processes is described axiomatically. Data flow in the functional model is described by the values returned by operations. Data stores are portrayed by separate object classes (DeL95b).

Table 2.1 provides a summary of the mappings from Rumbaugh's OMT to a theory-based model.

Table 2.1 Rumbaugh to Theory-based Object Model Translation

Rumbaugh Model	OO Concept	Theory-based Model
Object Model	classes	theory presentation
	attributes	operation on class sort
	operations	operation on class sort
	constraints	axioms
	object Instances	logical variables
	simple inheritance	morphism and subsort
	multiple inheritance	colimit and subsort
	aggregation	colimit and object-valued-attributes
	multiplicity	axioms
	associations	container of link objects
	link	theory presentation
	multiplicity	axioms
	qualifier	attribute and axioms
	link attributes	operations
	link operations	operations
	ordering	sequence of link objects
	constraints	aggregate axioms
Dynamic Model	transition Events	operations
	parameters	operation parameters
	actions	operations and axioms
	output Events	event theories
	state actions/activity	methods
	parameters	method parameters
	control flow	event theories
Functional Model	processes	operations
	operation definition	axioms
	data flow	operations return values
	data store	object classes

2.2.6 O-SLANG. O-SLANG is an object-oriented extension of the algebraic specification language SLANG. SLANG is used by Specware^{TM} to perform software refinement (BFG^{+}94). Based on category theory and first-order predicate logic, SLANG supports the specification morphism and colimit operations described in Section 2.2.1. An SLANG specification is made up of sorts, operations, and axioms (BGG^{+}94). O-SLANG takes these fundamental concepts of SLANG and uses them to capture object-oriented system specifications (DeL95c).

Sections 2.2.3 through 2.2.5 outlined a theory-based representation of an object-oriented model of a system. O-SLANG uses the same ideas to describe object class features and relationships between object classes. Appendix A shows an OMT domain model for O-SLANG.

2.3 Term Rewriting

This section presents a basic definition of a general term rewriting system as well as some definitions for the related concepts that are necessary to understand term rewriting. It wraps up with a discussion of some of the many types of term rewriting systems.

2.3.1 Term Rewriting Systems. Formally speaking, a term rewriting system is a pair $(\sum, R)$, where $\sum$ is an alphabet or signature and R is a set of rewrite rules. The syntax and vocabulary for a term rewriting system is (Klo92):

1. $\sum$ consists of a countably infinite set of variables $x_1, x_2, x_3, \ldots$ and a non-empty set $\sum_0$ of function symbols or operator symbols, each with an "arity", i.e. the number of arguments the function or operator is supposed to have.
2. The set of terms over $\sum$, $T(\sum)$ is defined inductively:

   (1) $x, y, z, \ldots \in T(\sum)$.

   (2) If $f \in \sum_0$ and $t_1, \ldots, t_n \in T(\sum)$ $(n \geq 0)$, then $f(t_1, \ldots, t_n) \in T(\sum)$.
3. Terms not containing a variable are ground terms.
4. A rewrite rule $\in R$ is a pair (l, r) of terms $\in T(\sum)$, written as $l \rightarrow r$. Rewrite rules can be named. (e.g. rewrite rule n is written as $r_n : l \rightarrow r$, and the application of r_n to some term α which produces some term β is written $\alpha \rightarrow_{r_n} \beta$).

Formalisms aside, term rewriting uses directed equations to iteratively replace subterms in a given expression with equal terms until the simplest form of the expression is reached. This is the same idea as simplifying expressions in algebra (Der93). The form could be some standard (canonical) form or some intermediate form needed to perform a manipulation of the symbols in the expression at a later time. For example, putting equations in disjunctive normal form for use with a mechanical theorem prover is term rewriting. Moving all quantification symbols to the left in a first order predicate logic equation is another example. Both of these manipulations produce equivalent terms since they are based on theorems in formal logic.

2.3.1.1 Simple Example. Klop (Klo92) provides the following simple example of a rewrite system:

- Let $\sum = \{A, M, S, 0\}$, with arities 2, 2, 1, and 0, respectively.
- Let R be defined by:

$$r_1 : A(x, 0) \rightarrow x$$
$$r_2 : A(x, S(y)) \rightarrow S(A(x, y))$$
$$r_3 : M(x, 0) \rightarrow 0$$
$$r_4 : M(x, S(y)) \rightarrow A(M(x, y), x)$$

Consider the expression: $M(S(S(0)), S(S(0)))$

Using the rewrite rules in R, the expression can be simplified to: $S(S(S(S(0))))$

One possible sequence of rewrite rule applications is as follows (underlined terms are rewritten in each step):

$$\underline{M(S(S(0)), S(S(0)))} \rightarrow_{r_4} A(M(S(S(0)), S(0)), S(S(0)))$$
$$\underline{A(M(S(S(0)), S(0)), S(S(0)))} \rightarrow_{r_2} S(A(M(S(S(0)), S(0)), S(0)))$$
$$S(\underline{A(M(S(S(0)), S(0)), S(0))}) \rightarrow_{r_2} S(S(A(M(S(S(0)), S(0)), 0)))$$
$$S(S(\underline{A(M(S(S(0)), S(0)), 0)})) \rightarrow_{r_1} S(S(M(S(S(0)), S(0))))$$
$$S(S(\underline{M(S(S(0)), S(0))})) \rightarrow_{r_4} S(S(A(M(S(S(0)), 0), S(S(0)))))$$

$S(S(A(\underline{M(S(S(0)),0)},S(S(0))))) \rightarrow_{r_3} S(S(A(0,S(S(0)))))$
$S(S(\underline{A(0,S(S(0)))})) \rightarrow_{r_2} S(S(S(A(0,S(0)))))$
$S(S(S(\underline{A(0,S(0))}))) \rightarrow_{r_2} S(S(S(S(A(0,0)))))$
$S(S(S(S(\underline{A(0,0)})))) \rightarrow_{r_1} S(S(S(S(0))))$

2.3.2 Basic Definitions in Term Rewriting. In addition to the basic notions of signature and rewrite rules, there are several concepts that are foundational to an understanding of term rewriting. These concepts can be lumped into two categories: those concerning the structure of terms, and those concerning rewriting operations. This section defines some of the more important concepts in these two categories.

2.3.2.1 Structural Term Definitions. Given a set F of function symbols, each function symbol $f \in F$ has a unique natural number associated with it called the *arity*, as described in Section 2.3.1. Any function with arity 0 is called a *constant* (Mit94). For a given set of variables, X, a term $t \in T(F, X)$ can be viewed as a finite ordered-tree where the leaves are variables in X or constants and the internal nodes are labeled with function symbols (Mit94). In the example in Section 2.3.1.1, $M(S(S(0)), S(S(0)))$ is a term. Terms are made up of *subterms*, which are substrings of symbols. The denotation of a subterm is $t\mid_p$, which represents the subterm of t which is rooted at position p in t (Der93). A position in a term can be represented by a sequence of positive integers that describes the path from the root symbol of the term to the head of the subterm that is rooted at that position. For example, if $t = push(0, pop(push(y, z)))$, then $t\mid_{2.1}$ is the first subterm of t's second subterm, i.e. $push(y, z)$ (Der93). A term is said to be *monadic* if it is made up only of unary functions, constants, and variables, and ends in either a constant or variable (Der93).

2.3.2.2 Rewriting Definitions. The rewriting of terms involves *replacement* of subterms with other terms. A term t with subterm $t\mid_p$ replaced by term s is denoted by $t[s]_p$ (Der93). When a term is replaced with a variable, it is referred to as *substitution.* A substitution is a function that uniquely maps variables to terms, and is written as $\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$. For a substitution σ, it is true that $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ (Mit94). A key operation in rewriting is determining when two terms *match.* A term t matches another term s if for some substitution σ, $s\sigma = t$. For two terms s and t and some substitution σ, if $s\sigma = t$, i.e. if t matches s, then s is said to *subsume* t (Der93). The selection of a rewrite rule may be *context* dependent. A context is some term u with a distinguished position p (Der93). Sometimes it is necessary to put together multiple substitutions. This process is called *composition.* The composition of two substitutions σ and θ is a composition of the two functions. For example, if $x\sigma = s$ for some variable x, then $x\sigma\theta = s\theta$ (Mit94).

The basic component of a rewrite system is a *rewrite relation*, which is defined as a binary relation $\rightarrow$ over a set of terms T that is closed with respect to replacement and substitution. If a rewrite relation is transitive and irreflexive, it is called a *rewrite ordering.* Finite or infinite sequences of applications of rewrite rules are called *derivations.* Derivations are denoted by $t_0 \rightarrow_R t_1 \rightarrow_R \cdots t_i \rightarrow_R \cdots$. A sequence of derivations from some term t_0 to another term t_n can be more compactly denoted by $t_0 \rightarrow^* t_n$. For a given rewrite system R, a term $s \in T$ is said to *rewrite* to a term $t \in T$ if $s\mid_p = l\sigma$ and $t = s[r\sigma]_p$ for some rule $l \rightarrow r$ in R, position p in s, and substitution σ. The rewrite is denoted by $s \rightarrow_R t$. The subterm $s\mid_p$ at which a rewrite can take place is called a *redex* (Der93).

When there is no term t such that $s \rightarrow t$, then s is said to be in *normal form*, denoted by $s \downarrow$ (Mit94).

2.3.3 Types of Rewrite Systems. The idea of a term rewriting system is very important to the study of computational procedures. One of the most well-known term rewriting systems, the λ-calculus, played a vital role in mathematical logic by helping formalize the concept of computability (Klo92). In the area of programming languages, the λ-calculus led to an important breakthrough in denotational semantics. Another term rewriting system, combinatory logic, has proven very helpful in implementing functional languages (Klo92). What makes term rewriting systems so desirable, at least those that involve terms in a first-order language, is their simple syntax and semantics (Klo92). This section briefly outlines some different types of term rewriting systems, while Section 2.3.6 describes another important type called graph rewriting systems.

One of the most basic types of term rewriting systems is called a *string-rewriting system*, or *semi-Thue system.* A string-rewriting system has monadic words that end in the same variable as left-hand and right-hand side terms (BO93). Consider the following example of a string-rewriting system (Klo92):

- Let $T = \{(aba, bab)\}$ be a string rewriting system with only one rule.
- T has unary function symbols a and b and a constant 0.
- T has one rule: $a(b(a(x))) \rightarrow b(a(b(x)))$.
- For the string $bbabaaa$, a reduction step might be $bb\underline{aba}aa \rightarrow bbbabaa$.

Another type of term rewriting system is known as *applicative term rewriting systems.* With these systems, there is a very special binary operator called *application*, or *Ap.*

Applicative term rewriting systems are very useful for Combinatory Logic. Consider the following example (vBSB93):

- Combinatory Logic can be represented as follows:
  $Sxyz = xz(yz)$
  $Kxy = x$
  $Ix = x$

- Combinatory Logic can be expressed as an applicative term rewriting system as follows:
  $Ap(Ap(Ap(S,x),y),z) \rightarrow Ap(Ap(x,z),Ap(y,z))$
  $Ap(Ap(K,x),y) \rightarrow x$
  $Ap(I,x) \rightarrow x$

A special case of applicative term rewriting systems is where all of the rewrite rules are left-linear. Left-linear means that no variable occurs more than once on the left-hand side of any rewrite rule. Using a tree representation for terms and rewrite rules, the concept of type assignment can be defined by assigning types to nodes and edges in a consistent manner. Van Bakel, et. al., developed a necessary and sufficient condition for preservation of types in left-linear applicative systems (vBSB93).

Term rewriting systems can be extended by allowing rewrite rules to have conditions attached to them. This is known as *conditional rewriting.* These conditions are really *enabling conditions*, i.e. the conjunction of all of the conditions must be true before the rewrite rule can be applied (for generalized systems (Klo92)). Consider the following set of rewrite rules for a stack (Der93):

$$top(push(x,y)) \rightarrow x$$

$$pop(push(x,y)) \rightarrow y$$

$$empty?(\Lambda) \rightarrow yes$$

$$empty?(push(x,y)) \rightarrow no$$

$empty?(x) = no \quad | \quad push(top(x), pop(x)) \rightarrow x$

The last rule is a conditional, while the others are not. If the stack is not empty (i.e. $empty?(x) = no$), then the rule $push(top(x), pop(x)) \rightarrow x$ can be applied.

In *priority rewriting systems*, the choice of which rewrite rule is to be applied is constrained to meet, *a priori*, some given priorities on the rules. In other words, priorities are merely a partial ordering of the rewrite rules. For example, the original Markov algorithms were a priority rewrite system in which the order in which the rules were written down determined their priority. Generally, priority rewrite systems can't be expressed as term rewriting systems (Der93).

There are different types of term rewriting systems, only a few of which have been presented in this section. Other examples are *graph rewriting systems*, *class rewriting systems*, and *ordered rewriting systems*. The latter two are extensions of general term rewriting systems that are designed to deal with problems of non-termination, such as commutativity (Der93). See Section 2.3.6 for a discussion of *graph rewriting*.

2.3.4 Properties of Rewrite Systems. Term rewriting systems can have many properties. This section describes some of them, particularly those properties that make a term rewriting system "nice". Included are discussions on confluence, termination, unique normalization, and convergence.

2.3.4.1 Confluence. Figure 2.8 provides a graphical view of confluence, also referred to as the *Church-Rosser* property. The basic idea is that no matter what order the rewrites are applied, the result is the same. There are two forms of confluence: local

confluence and confluence. Locally confluent systems are said to be *weakly Church-Rosser*, while confluent systems are said to be *Church-Rosser*.

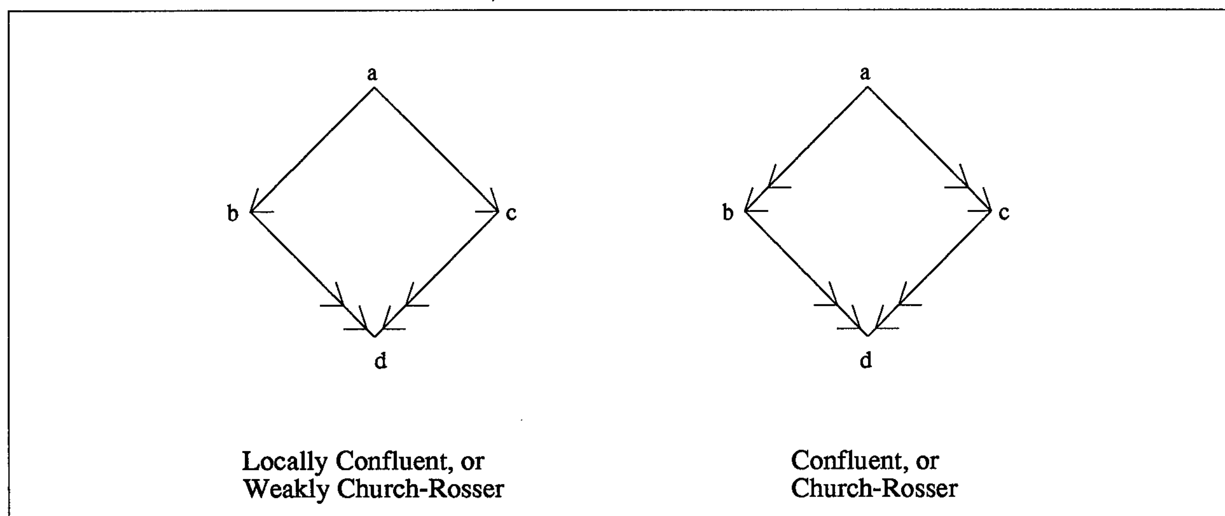

Figure 2.8 Confluence/Church-Rosser Properties

The formal definitions of these properties are as follows (Klo92):

- The binary relation $\rightarrow$ is locally confluent (weakly Church-Rosser) if

  $\forall a, b, c \in T \;\; \exists d \in T \; (a \rightarrow b \text{ and } a \rightarrow c \Rightarrow b \rightarrow^* d \text{ and } c \rightarrow^* d).$

- The binary relation $\rightarrow$ is confluent (Church-Rosser) if

  $\forall a, b, c \in T \;\; \exists d \in T \; (a \rightarrow^* b \text{ and } a \rightarrow^* c \Rightarrow b \rightarrow^* d \text{ and } c \rightarrow^* d).$

2.3.4.2 Termination. Another important property for term rewriting systems is *termination*, also called *strong normalization.* Simply put, a system is terminating (strongly normalizing) if there are no infinite derivations $t_1 \rightarrow_R t_2 \rightarrow_R \cdots$ of terms in T (Der93). Termination is a very useful property. If a system is known to be locally confluent, proving that it is also terminating will show that the system is confluent, since by Newman's lemma *locally confluent and terminating* $\Rightarrow$ *confluent* (Klo92). The catch

is that for term rewriting systems in general, the question of termination is undecidable (Klo92). Fortunately, there are many cases in which termination can be proved. Klop demonstrates a very powerful proof technique based on recursive path orderings (Klo92), and Dershowitz provides a survey of termination techniques in (Der87). Termination can also be guaranteed by creating a well-founded ordering in which the rewritten form of a term is always smaller than its original form (Der94).

2.3.4.3 Unique Normalization and Convergence. According to Dershowitz, one of the most essential properties for a rewriting system is *unique normalization.* If a term rewriting system has this property, then every term $t \in T$ has exactly one normal form. If all possible sequences of rewrites lead to a unique normal form, then the system is said to be *convergent.* Dershowitz states that if a rewrite system can be shown to be terminating and confluent, then that system is convergent and defines unique normal forms (Der93). Both Klop and Dershowitz also refer to this property as *canonical*; however, Klop prefers to call terminating, confluent rewrite systems *complete* (Klo92).

2.3.5 Example Rewrite System. Up to this point, the basic notions of term rewriting systems have been presented. This section will present a practical example of a term rewriting system which converts first-order predicate logic equations in typical infix notation to equivalent equations in a prefix notation that is similar to the syntax of SLANG, an algebraic specification language. For example, let $(\sum, R)$ be defined by the following rewrite rules:

1. $op1 \wedge op2 \rightarrow$ (And op1 op2)
2. $(op1 \wedge op2) \rightarrow$ (And op1 op2)
3. $op1 \vee op2 \rightarrow$ (Or op1 op2)

4. $(op1 \vee op2) \rightarrow$ (Or op1 op2)
5. $op1 \Rightarrow op2 \rightarrow$ (Implies op1 op2)
6. $(op1 \Rightarrow op2) \rightarrow$ (Implies op1 op2)
7. $op1 \Leftrightarrow op2 \rightarrow$ (Iff op1 op2)
8. $(op1 \Leftrightarrow op2) \rightarrow$ (Iff op1 op2)
9. $!op \rightarrow$ (Not op)
10. $!(op) \rightarrow$ (Not (op))
11. $\circ op \rightarrow$ ($\circ$ op)
12. $\circ(op) \rightarrow$ ($\circ$ (op))
13. $(op1 \Box op2) \rightarrow$ ($\Box$ op1 op2)
14. $\triangle$ (ops) $\rightarrow$ ($\triangle$ ops)
15. $op1 = op2 \rightarrow$ (Equal op1 op2)
16. $(op1 = op2) \rightarrow$ (Equal op1 op2)
17. $op1 + op2 \rightarrow$ (Iplus op1 op2)
18. $(op1 + op2) \rightarrow$ (Iplus op1 op2)
19. $op1 - op2 \rightarrow$ (Minus op1 op2)
20. $(op1 - op2) \rightarrow$ (Minus op1 op2)
21. $op1 * op2 \rightarrow$ (Times op1 op2)
22. $(op1 * op2) \rightarrow$ (Times op1 op2)
23. $(op1 \cup op2) \rightarrow$ (Union op1 op2)
24. $(op1 \cap op2) \rightarrow$ (Intersect op1 op2)
24. $(op1 \in op2) \rightarrow$ (In op1 op2)

In these rules, the operators have the standard first-order predicate logic precedence. The symbols $\circ$, $\Box$, and $\triangle$ represent user-defined unary relations, user-defined binary relations, and user-defined functions, respectively. For this example, consider the following first-order predicate logic equation:

$$(\mathrm{x} \in U \cup V) \Leftrightarrow ((\mathrm{x} \in \mathrm{U}) \vee (\mathrm{x} \in \mathrm{V}))$$

This equation can be rewritten using the following sequence of rewrites:

(x $\in$ U $\cup$ V) $\Leftrightarrow$ ((x $\in$ U) $\vee$ (x $\in$ V)) $\rightarrow_{14}$
    (x $\in$ (Union U V)) $\Leftrightarrow$ ((x $\in$ U) $\vee$ (x $\in$ V))
(x $\in$ (Union U V)) $\Leftrightarrow$ ((x $\in$ U) $\vee$ (x $\in$ V)) $\rightarrow_{24}$
    (In x (Union U V)) $\Leftrightarrow$ ((x $\in$ U) $\vee$ (x $\in$ V))
(In x (Union U V)) $\Leftrightarrow$ ((x $\in$ U) $\vee$ (x $\in$ V)) $\rightarrow_{7}$
    (Iff (In x (Union U V)) ((x $\in$ U) $\vee$ (x $\in$ V)))
(Iff (In x (Union U V)) ((x $\in$ U) $\vee$ (x $\in$ V))) $\rightarrow_{24}$
    (Iff (In x (Union U V)) ((In x U) $\vee$ (x $\in$ V)))
(Iff (In x (Union U V)) ((In x U) $\vee$ (x $\in$ V))) $\rightarrow_{24}$
    (Iff (In x (Union U V)) ((In x U) $\vee$ (In x V)))

(Iff (In x (Union U V)) ((In x U) $\vee$ (In x V))) $\rightarrow_4$
(Iff (In x (Union U V)) (Or (In x U) (In x V)))

The order of application of the rules was chosen based on knowledge of the precedence of the operations in the system. This idea of precedence can be captured in an LALR(1) grammar. If the example expression had been parsed into tree form, it would have looked like figure B.1. Figures B.1 through B.6 each portray a rewrite of the example expression. In step 1, there are five different rewrite choices, each enclosed in a box. Going from step 1 to step 2, the subtree corresponding to choice 2 is rewritten. Looking at the steps, it is easily seen that the choice of which rewrite to apply at each step does not affect the final form. This is confluence. Also, in looking at the rewrite rules for the system, it can be seen that each rewrite produces a term that is "smaller". In other words, there is less to rewrite since no terms that match the left-hand side of any rule are produced. This is termination. Termination was achieved because the rules are a well-founded ordering, as mentioned in Section 2.3.4. Since the example term rewriting system is confluent and terminating it is also convergent and defines unique normal forms (Der93). Another way to describe the system is to say it is *canonical.*

2.3.6 Graph Rewriting. The example in Section 2.3.5 demonstrated rewriting on a tree representation of a first-order predicate logic expression. Rewriting can be generalized to apply to graphs as well as simple trees. In graph rewriting, subgraphs are replaced according to rewrite rules which contain variables. The variables themselves refer to subgraphs (Der93). Because graphs do not have simple structures, like trees, graph rewriting has a more global flavor. Graph rewriting systems are more powerful, but as is usually the

case, are more complicated. Where LALR grammars can be used in conjunction with tree rewriting, graph grammars are required for graph rewriting.

2.4 Summary

This literature review provided the knowledge base needed to extend AFIT's object-based composition system towards the capability to perform design refinement. First, before defining a canonical model which represents object-oriented models algebraically, it is necessary to understand how object-oriented constructs such as inheritance, aggregation, and associations can be described in terms of theories. It is also necessary to understand how these constructs are represented in Rumbaugh's OMT, which is the starting point for the object-based composition system. Knowing the source and target of the composition system provides the background necessary to design the extension to the formal object transformation system. Finally, understanding the concepts of term rewriting provides some insight into showing that the transformations from the unified model to the target canonical model produce unique normal forms.

III. Designing a Formal Object Transformation Process

3.1 Introduction

In Chapter I, the notion of producing domain-specific applications from object-oriented domain models was presented. The foundation for such a system is Lin and Wabiszewski's formalized object transformation process. By following the general compiler model, they developed compilers which take in a formal specification language representation of an object-oriented domain model (ULARCH or UZed) and produce portions of an executable REFINE program. Looking at the portion of Figure 3.1 above the top dotted line, it can be seen that their ULARCH and UZed parsers correspond to the analysis block, while their transformations to REFINE correspond to the synthesis block. Both compilers have the unified AST as an intermediate representation and REFINE as a target language. In this research effort, Lin's compiler was modified by changing the target language to the

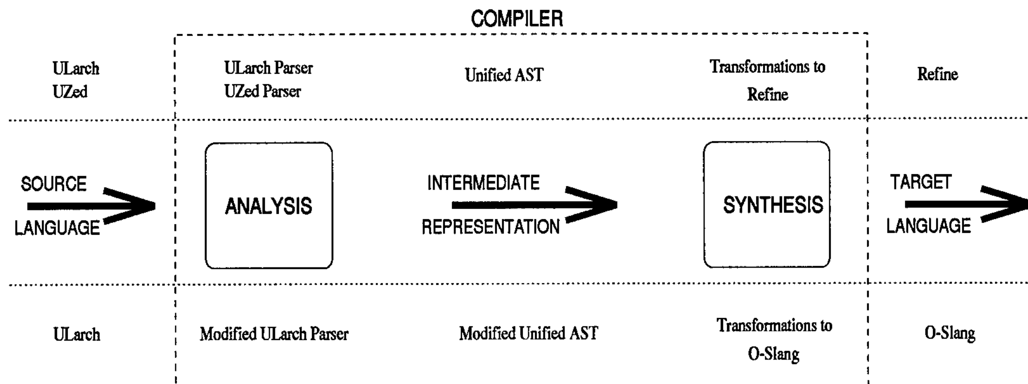

Figure 3.1 Analysis Synthesis Model

canonical model and by developing new transformations to replace the existing synthesis block. Also, the ULARCH parser was modified to account for changes made to the unified model. This is shown in the portion of Figure 3.1 that appears below the dotted line.

This compiler model identifies the required functional components of the extended formal object transformation process:

- A source language, ULARCH, which is a Larch representation of an OMT domain model
- The analysis portion of the compiler, which is a modified version of Lin's ULARCH parser
- An intermediate representation, which is a modified version of the unified AST
- The synthesis portion of the compiler, which is the transformations from the modified version of the unified AST to the canonical AST
- A target language, which is the canonical AST

To implement these components, an evolutionary approach was adopted which consisted of three phases with the following associated products:

1. A canonical algebraic framework
2. A modified version of Lin's unified model
3. A set of transformations from the unified model to the canonical model

Chapter IV provides a description of the changes made to the unified model, while Chapters V and VI present the design and implementation of the transformations from the modified version of the unified model to the canonical model. This chapter outlines the definition of the canonical model and the design validation criteria in Sections 3.3 and 3.4 after an overview of Lin and Wabiszewski's object transformation process.

3.2 Creating the Unified Framework

As stated in Chapter I, Lin and Wabiszewski developed a formalized object transformation process. Lin employed a theory-based approach using Larch, while Wabiszewski pursued a model-based approach using Z. This formalized transformation was developed in three main steps. Initially, they created and validated mappings from the OMT models

to Larch and Z. Once these mappings were validated using existing problem domain OMT models, they developed parsers for Larch and Z that created abstract syntax tree (AST) representations. These representations served as a basis for the next step, in which they analyzed the similarities and differences between the two resulting representations and created a unified model (Lin94).

The first step in developing the formalized transformation process was to map Rumbaugh's OMT models to the algebraic and model-based constructs of Larch and Z, respectively. In OMT, the object model captures the important objects and associated attributes and relationships between objects in the system. Also, object-oriented concepts such as inheritance, aggregation, and attribute invariance are present. The dynamic model embodies the reactive behavior of an object in terms of states and events. The functional model relates an object's data transformations, encapsulating a collection of data and operations on the data in much the same way as an abstract data type (RBP+91).[1] Creating verifiable frameworks for the object-oriented models required the preservation of the constructs present in the each of these models. Lin and Wabiszewski satisfied this requirement by addressing each model individually. Table 3.1 briefly summarizes the mappings from the OMT models to the algebraic and object-based frameworks (Lin94, HB94).

The second step in developing the formalized transformation process was to create parsers to translate algebraic specifications written in Larch and model-based specifications written in Z into AST representations. To accomplish this task, an object model was built for each language. These models were then used to build a formal language syntax in an extended Backus Naur Form (BNF) notation. The parsing toolset in SOFTWARE

[1] See Section 2.2.2 for a more in-depth description of Rumbaugh's OMT.

Table 3.1 OMT Mappings to Algebraic and Object-based Frameworks

Model	OMT Component	Algebraic Framework	Model-based Framework
Object Model	Object Class	Trait	Schema
	Attributes	Operators, Sorts	Schema Attributes
	Relations	Axioms	Schemas, Axioms
	Inheritance	Includes, Renames	Schema Inclusion
	Aggregation	Includes, Renames	Schema Attributes
Dynamic Model	State	Trait	Schema
	Event	Trait	Schema
	State Transition Table (STT)	STT	STT
Functional Model	Data Transform	Traits, Operators	Schema
	Behavior	Axioms	Axioms

REFINERY^{TM} was then used to parse Larch and Z programs based on these BNF notations (Lin94).

Once the AST representations for each language were developed, Lin and Wabiszewski used them to evaluate the structures of Larch and Z as used to describe OMT models. Their evaluation revealed that the two languages have strong similarities in the way they represent specifications. Both languages require signatures, axioms, and external references to describe a problem domain. These requirements are fulfilled in each language using different syntax, but the semantics, i.e. mathematical foundations, are similar. These similarities make up a set of common core objects in the unified model. Evaluation of the ASTs also showed the differences between the two languages. For example, Z has the capability to explicitly declare input and output variables, while Larch does not. These types of differences make up a set of language specific objects in the unified model.

Lin and Wabiszewski concluded that Larch and Z have a common set of constructs that can be used to build a canonical framework for formalizing object models. These constructs can be put together to create one unified model that serves as a front-end

for formal system composition, and which supports theorem-proving, code generation, and design refinement. They noted, however, that because of the significant differences between the syntax of Larch and the syntax of Z, their unified framework contains language specific extensions, and so is not completely unified (Wab94, Lin94).

3.3 The Canonical Algebraic Framework Phase

Section 3.1 outlined the phases required to extend Lin and Wabiszewski's formal object transformation process. The starting point was identified as the selection of the canonical framework to serve as the target language for the compiler. The object-oriented specification language O-SLANG, introduced in Section 2.2.6, was chosen as the target canonical framework. This selection was based on two criteria: completeness of coverage of OMT, and compatibility with the design refinement process. This section provides a discussion of these criteria.

The goal of each transformation step in a transformation system is to produce an output that is equivalent to the input, i.e. no loss of information. If the target representation, or framework, is not capable of capturing all of the information portrayed in the source, then some information will be lost in the transformation. This is incomplete coverage. The first criterion for creating the canonical algebraic framework, then, was to analyze the unified model and to identify the parts which were necessary for preservation of the original object-oriented model. Since the model was based on Rumbaugh's OMT, it was possible to analyze it in light of the three distinct views that OMT provides of a system: the object model, the dynamic model, and the functional model. O-SLANG cap-

tures all of the object-oriented concepts in OMT, and as will be discussed in Chapter IV, captures more than the original unified model created by Lin and Wabiszewski.

Table 3.2 Rumbaugh to O-SLANG Translation

OMT Model	OO Concept	O-SLANG Feature
Object Model	classes	class specs
	object Instances	class specs
	attributes	attributes
	operations	operations
	- constraints	- axioms
	simple inheritance	subsort, import
	multiple inheritance	subsort, import
	aggregation	aggregate spec
	- multiplicity	- axioms
	links	link spec
	associations	association specs
	- multiplicity	- axioms
	- qualifier	- axioms
Dynamic Model	transition Events	event specs
	- parameters	- parameters
	- actions	- methods
	output Events	event specs
	state actions/activity	- methods
	- parameters	- parameters
	control flow	axioms
Functional Model	processes	methods
	operation definition	axioms
	data flow	operations return values
	data store	class specs

Table 3.2 provides a summary of the object-oriented concepts which need to be captured in the canonical framework along with the features of O-SLANG which cover them. Chapter V provides a more detailed discussion of the mappings from the unified model to O-SLANG.

The second reason for selecting O-SLANG as the algebraic framework was its capability to represent the operations needed for design refinement, such as specification morphisms and colimits. As discussed in Section 2.2.6, O-SLANG is an extension of SLANG which is the language used by SpecWare^{TM} in performing design refinement. It is based on

category theory concepts such as morphisms, colimits, and diagrams. For example, with inheritance in O-SLANG, the class-sort of the inheriting object is a subsort of the class-sort of the object it is inheriting from. This corresponds to a morphism in SLANG. Aggregates in O-SLANG correspond to a colimit in SLANG.

3.4 Validation Criteria and Domains

In order to validate the canonical framework and the transformations from the unified model to the framework, two criteria were chosen, coverage and consistency. These criteria can be defined as follows:

1. Coverage: The entire OMT model is captured in the canonical model
2. Consistency: The behavioral constraints of the system do not contain any contradictions, that is any instances where $true = false$

To demonstrate that these criteria were met, two examples were developed, a gasoline pump and a bank. Their main purpose was to provide example coverage of OMT so that transformations from the unified model to the canonical model could be analyzed and validated with respect to the coverage criterion. The emphasis in building the examples was on exercising each facet of OMT rather than on developing complete domain models. Table 3.3 shows the examples' coverage of OMT. The following sections outline the process used to apply the chosen validation criteria and describe each example in detail.

3.4.1 Validation Process. In order to determine compliance with the validation criteria introduced in Section 3.4, the compiler had to be checked at three points: before the input to the parser, after the run of the parser and before the execution of the trans-

Table 3.3 OMT Coverage of Validation Domains

OMT	Bank	Pump
classes	•	•
object Instances	•	•
attributes	•	•
operations	•	•
- constraints	•	•
simple inheritance	•	•
multiple inheritance	•	
aggregation	•	•
- multiplicity	•	•
links	•	•
associations	•	•
- multiplicity	•	•
- qualifier	•	•
transition Events	•	•
- parameters	•	
- actions	•	
output Events	•	•
state actions/activity	•	•
- parameters	•	•
control flow	•	•
processes	•	•
operation definition	•	•
data flow	•	•
data store	•	

formations, and finally after the execution of the transformations. At each of these three points, compliance with the coverage and consistency criteria was evaluated.

The first step in the validation process was to evaluate the traits created for the bank and pump examples against the original domain models. This evaluation ensured that the traits were consistent with the models' structure, operations, and invariants, demonstrated that complete coverage of the models was achieved, and provided validated input for the compiler.

Once the input to the compiler was validated, the ULARCH parser could be run and the abstract syntax tree produced could be checked to see if all objects in the source

program were created as expected. To check the abstract syntax tree, a graphical tool called *Inspector* was used to visually inspect the objects, attributes, and overall structure of the tree [2]. This ensured that the synthesis portion of the compiler had validated input.

The final step in the validation process involved checking the O-SLANG produced by the transformations. Each O-SLANG file created was checked to ensure that all of the aspects in the original OMT domain model were present in the O-SLANG domain theory. Also, all axioms in the domain theory were checked to ensure that no inconsistencies were present.

3.4.2 Bank Domain. The bank domain example is a fairly complex system which covers all of the OMT concepts being validated. This section describes the object, dynamic, and functional models for the bank domain. Appendix C contains the ULARCH representation of the bank domain model.

3.4.2.1 Bank Object Model. The object model contains 14 different object classes: bank, person, console, account, archive, customer, employee, checking account, savings account, customer-employee, executive, teller, combined checking and savings account, and date. Aggregation is covered by the bank object, which is the top-level system aggregate. Inheritance and multiple inheritance are both covered by classes inheriting from the account object class and person object class. Several associations are also modeled. For example, the owns association models the relationship where a bank customer can own zero or more accounts. Figure 3.2 depicts the bank domain object model.

[2] *Inspector* is provided by a package called *Intervista* which is part of the *Software Refinery*^{TM} environment (int91).

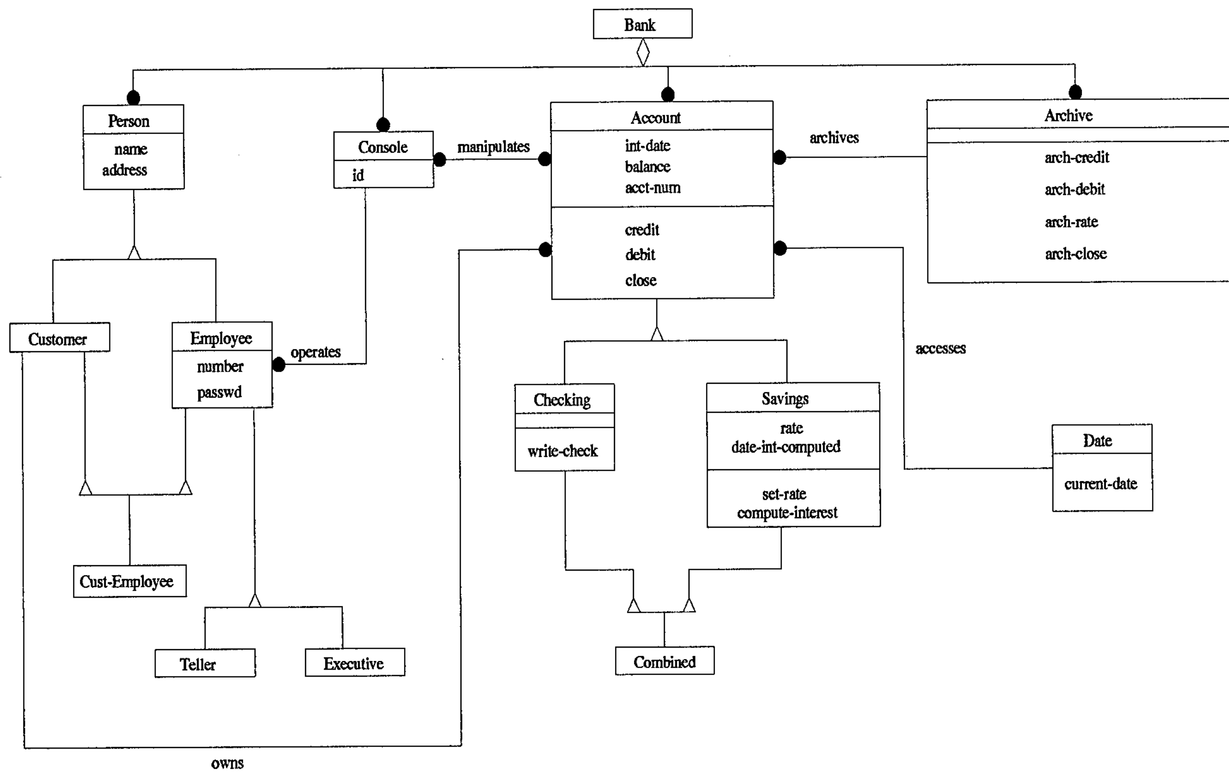

Figure 3.2 Bank Object Model

3.4.2.2 Bank Dynamic Model. The dynamic model used for the bank domain consists of state diagrams and state transition tables for the account and console object classes. The dynamic model covers simple state transitions, actions, and single receiver send events. The account dynamic model is portrayed in Figure 3.3, while the console dynamic model is shown in 3.4.

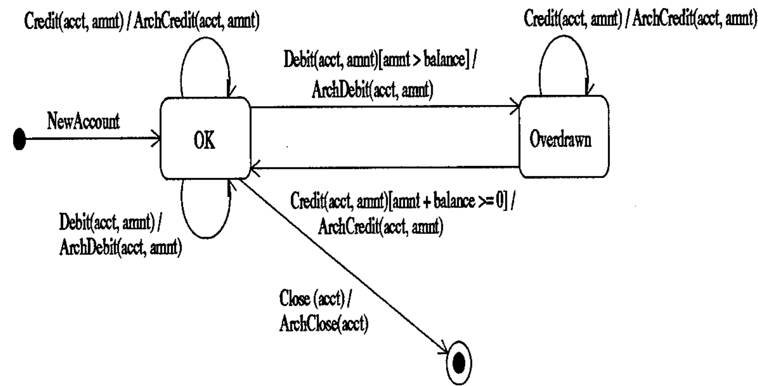

Figure 3.3 Account Dynamic Model

3.4.2.3 Bank Functional Model. The functional model for the bank domain consists of several data transformations; the account class has credit-acct, debit-acct, and close-acct, the archive class has arch-credit, arch-debit, arch-close, and arch-rate, and the checking account class has write-check, set-rate, compute-interest. The functional model is shown in Figure 3.5.

3.4.3 Pump Domain. The pump domain example is not as complex as the bank domain. It does not cover multiple inheritance, parameterized events, or data stores. It does, however, cover events sent to multiple receivers. This section describes the object, dynamic, and functional models for the pump domain. Appendix E contains the ULARCH representation of the pump domain model.

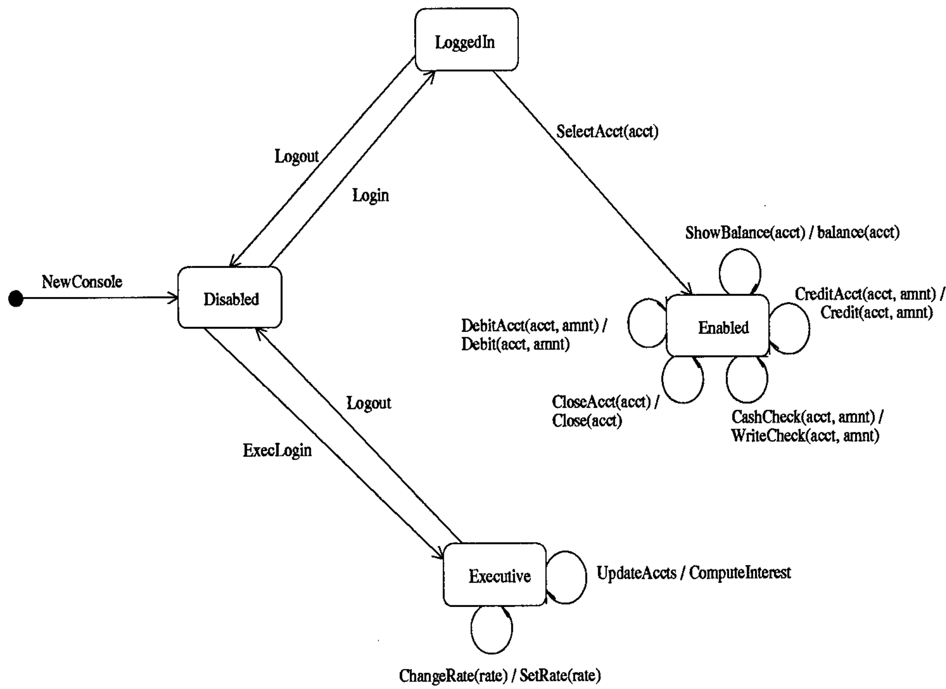

Figure 3.4 Console Dynamic Model

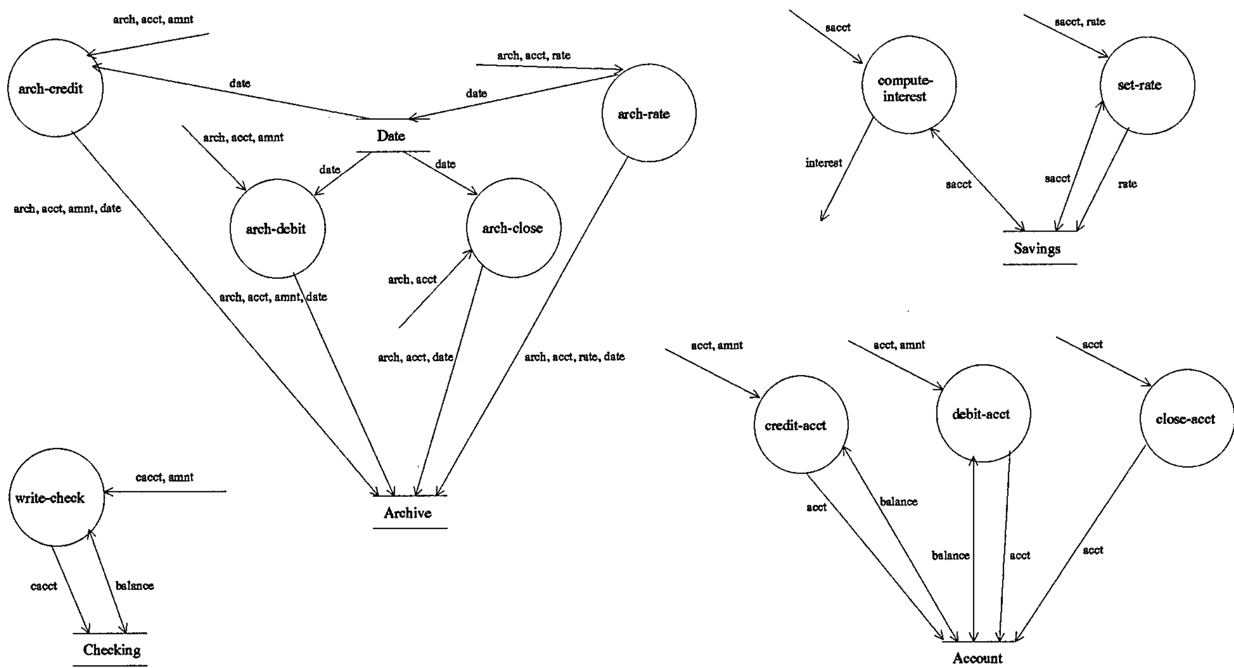

Figure 3.5 Bank Functional Model

3.4.3.1 Pump Object Model. The object model contains 10 different object classes: pump, display, pump-controller, gun, holster, gun-holster assembly, clutch, motor, clutch-motor assembly, and sophisticated pump. Aggregation is covered by the pump object, which is the top-level system aggregate. The gun-holster assembly object and clutch-motor assembly object also cover aggregation. Inheritance is covered by the sophisticated pump class which inherits from the pump object class. One association is modeled. The kept-in association models the relationship where a particular gun is kept in a particular holster. Figure 3.6 depicts the pump domain object model.

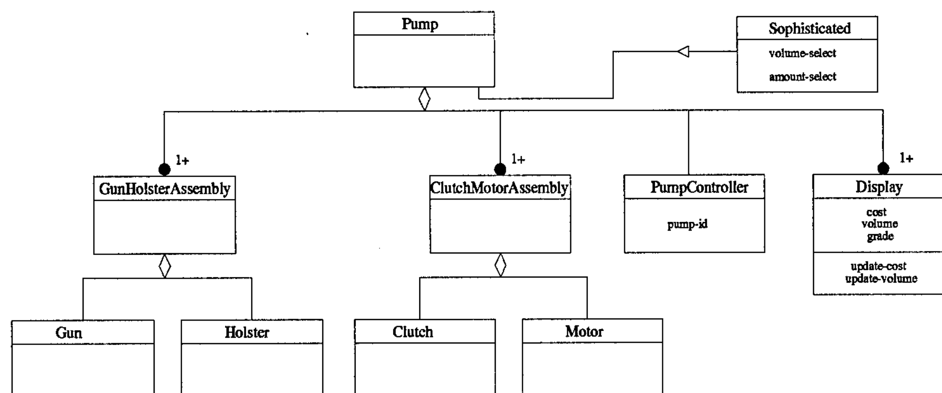

Figure 3.6 Pump Object Model

3.4.3.2 Pump Dynamic Model. The dynamic model used for the pump domain consists of state diagrams and state transition tables for the clutch, display, gun, holster, motor, and pump object classes. The dynamic model covers simple state transitions, actions, and single and multiple receiver send events. The pump dynamic model is portrayed in Figure 3.7.

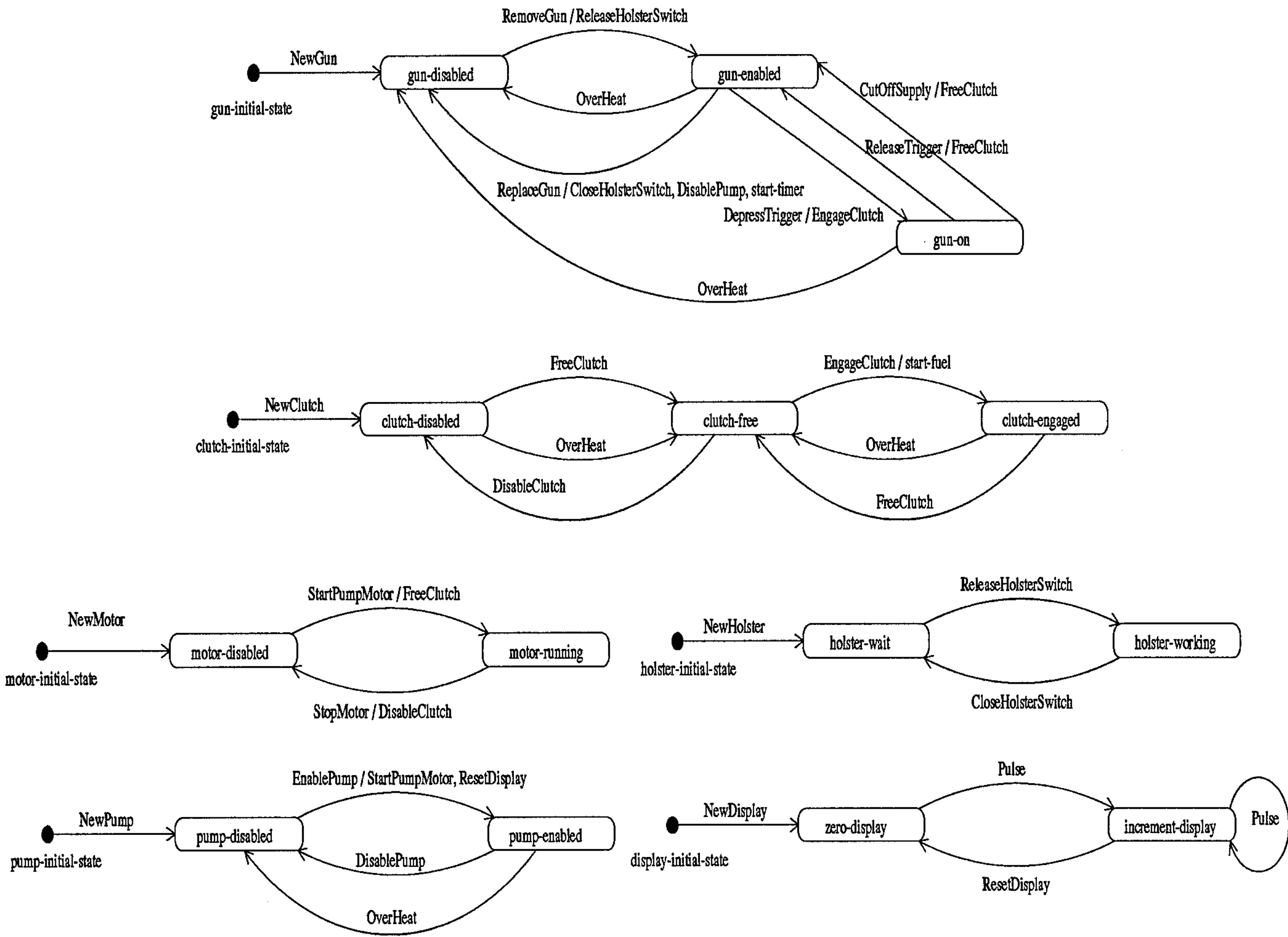

Figure 3.7 Pump Dynamic Model

3.4.3.3 Pump Functional Model. The functional model for the pump domain consists of two data transformations; the display class has update-cost and update-volume. The functional model is shown in Figure 3.8.

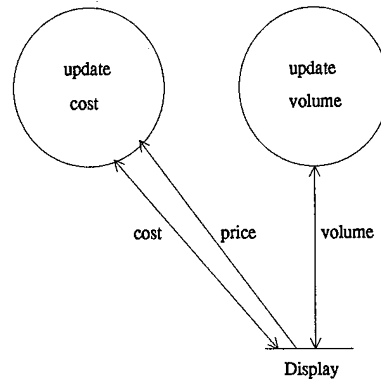

Figure 3.8 Pump Functional Model

3.5 Summary

This chapter outlined the compiler design model on which the extension of Lin and Wabiszewski's formal object transformation process was based. To implement the design, three phases were identified: defining a canonical algebraic framework, extending the unified model, and creating transformations from the unified model to the canonical model. The first of these phases was described in Section 3.3, where O-SLANG was identified as the chosen canonical algebraic framework [3]. The identification of the target canonical model, along with the analysis of Lin and Wabiszewski's unified framework, set the stage for the second phase which was analyzing the unified model to determine what changes were required. These changes are discussed in Chapter IV. The final phase, defining and implementing the transformations from the unified model to O-SLANG, is described in Chapters

[3] The canonical algebraic framework, or canonical model, will henceforth be referred to as O-SLANG.

V and VI. Also presented in this chapter were the validation criteria and domains used to analyze the extended formal object transformation process. This analysis is presented in Chapter VI.

IV. Extensions to the Unified Model

4.1 Introduction

Section 3.3 presented the first phase in extending the formal object transformation process: the selection of O-SLANG as a canonical framework for algebraic models. This selection established a target representation for transformations from the unified model. Before these transformations could be developed, however, it was necessary to evaluate the existing unified model to determine if it could capture all of the information needed to represent object-oriented domain models in O-SLANG. An analysis of the unified model showed that four changes were needed in ULARCH and one change was needed in the state transition tables. Also, changes had to be made in Lin's mappings from OMT to ULARCH. This chapter outlines the changes made to the original unified model.

4.2 Changes to ULARCH

Three of the four changes made to ULARCH were required in order to add information to the unified model, while the remaining change was made to facilitate the implementation of transformations to O-SLANG. This section describes the changes made to the ULARCH domain model and grammar.

4.2.1 Changing the Dynamic Theory. In Lin's domain model for ULARCH, the top level object is a *DomainTheory*, which consists of three other objects: an *ObjectTheory*, a *DynamicTheory*, and a *FunctionalTheory*. These three theory objects correspond to the Object, Dynamic, and Functional Models in OMT, respectively. In the ULARCH grammar, the objects are differentiated by the presence of a comment, as seen in Figure

4.1. Unfortunately, there is no way to distinguish between states and events from the

```
\begin{spec} %ObjectTheory
FuelTank: trait
  includes ...
  introduces ...
  asserts ...
\end{spec}\\

\begin{spec} %DynamicTheory
UsingState: trait
  includes ...
  introduces ...
  asserts ...
\end{spec}\\

\begin{spec} %DynamicTheory
StartFill: trait
  includes ...
  introduces ...
  asserts ...
\end{spec}\\

\begin{spec} %FunctionalTheory
CalculateFilledLevel: trait
  includes ...
  introduces ...
  asserts ...
\end{spec}\\
```

Figure 4.1 Traits in original ULARCH

dynamic model. For example, in Figure 4.1 both *UsingState* and *StartFill* are *DynamicTheory* objects. When building transformations, states and events must be distinguishable, as they really are different objects with different semantics. To capture this difference in semantics, the domain model for ULARCH was changed by breaking the *DynamicTheory* object into two separate objects, a *StateTheory* and an *EventTheory*. Also, the ULARCH grammar was changed by adding the syntax for the two new theories. After the changes, the traits in Figure 4.1 appear as in Figure 4.2.

```
\begin{spec} %ObjectTheory
FuelTank: trait
  includes ...
  introduces ...
  asserts ...
\end{spec}\\

\begin{spec} %StateTheory
UsingState: trait
  includes ...
  introduces ...
  asserts ...
\end{spec}\\

\begin{spec} %EventTheory
StartFill: trait
  includes ...
  introduces ...
  asserts ...
\end{spec}\\

\begin{spec} %FunctionalTheory
CalculateFilledLevel: trait
  includes ...
  introduces ...
  asserts ...
\end{spec}\\
```

Figure 4.2 Traits in modified ULARCH

4.2.2 Addition of Link and Association Theories. Lin and Wabiszewski's unified model does not capture the concept of associations between objects. Associations are a vital part of object-oriented models. Without them, the only relationships between objects that can be represented are aggregation and inheritance. In order to be able to represent associations in O-SLANG, the ULARCH domain model was modified by adding two new types of trait objects, *LinkTheory* and *AssociationTheory*. In OMT, a *link* represents a

relationship between two (or more) objects, while an *association* represents a group of links with common structure and meaning (RBP+91). A *LinkTheory*, then, is a trait which represents a link in OMT. An *AssociationTheory* is a trait which represents a set of individual links. This relationship between the *AssociationTheory* and *LinkTheory* is analogous to the relationship between classes and objects (DBH95). In addition, the ULARCH grammar was modified to include the syntax for the two new theories. This was done using comments, as with the other theory types. Figure 4.3 shows the top levels of the unified model before and after modification.

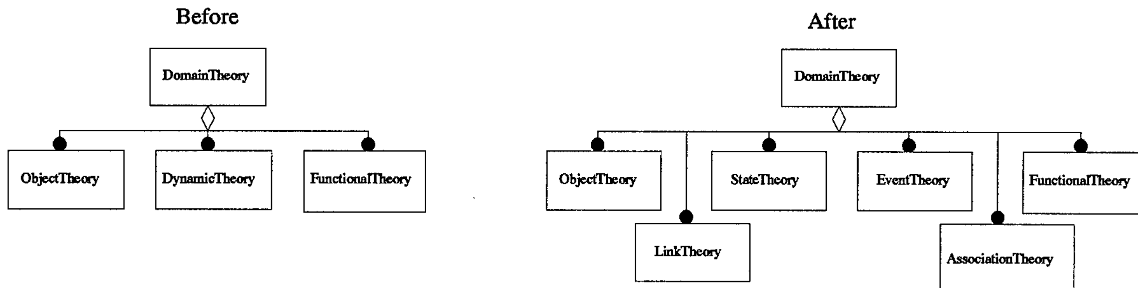

Figure 4.3 Changes to ULARCH Domain Model

4.2.3 Addition of Boolean Attribute. The remaining change made to the unified model was to add a boolean attribute, *done-Transform*, to certain objects in the ULARCH domain model. As will be discussed in Chapter VI, the ULARCH-to-O-SLANG transformations are a series of rules which are applied to the abstract syntax tree representation of a ULARCH domain theory. A top-down control structure is used for applying those rules, via a pre-order traversal of the tree. When a rule is successfully applied, the traversal starts over with the object that results from the application of the rule. This could lead to an infinite loop of rule applications. To prevent this, the rules must be defined in a way that

ensures that something changes to indicate that the rule has been applied and should not be reapplied. This can be achieved in several ways (Ref90):

- Have the rule transform the target object into some other object to which the rule cannot be applied.
- Have the rule annotate the object to indicate that the rule has been applied. The left-hand side of the rule should check for the annotation.
- Have the rule modify some global data structure to indicate that the rule has been applied. Again, the left-hand side of the rule should check the data structure.

Using a boolean attribute falls into the second category. The *done-Transform* attribute is initialized to *false* when an object is created in the abstract syntax tree, and when a rule is successfully applied to it, the attribute is set to *true*. Each rule has the precondition *not done-Transform*, which ensures that a rule can be applied to an object only once.

4.3 Changes to OMT-to-ULARCH Mappings

In addition to changes to the ULARCH domain model and grammar, several changes were also needed in Lin's mappings from OMT to ULARCH. Some of these changes were necessitated by additions to the ULARCH domain model, and some were needed to add information required for mapping ULARCH to O-SLANG. This section provides a description of the changes made to the mappings from OMT to ULARCH.

4.3.1 Addition of Link and Association Theories. Once the syntax and the domain model were defined for *LinkTheory* and *AssociationTheory* objects, mappings had

to be defined from OMT links and associations to the respective ULARCH representation. To derive ULARCH traits for associations, the following steps are needed:

1. Create a ULARCH *LinkTheory* trait for each association which includes the traits for each object participating in the association.
2. Define an LSL [1] operator for each participant in the relationship using the form: attr-name: link-sort → attr-sort.
3. Create an LSL sort for the LINKTHEORY trait, and for each link-attribute define an attribute.
4. Define an LSL "new" operator which constructs a link.
5. Use axiomatic equations to describe the behavior of the "new" operator.
6. Create a ULARCH *AssociationTheory* trait for each association which includes a set of *LinkTheory* objects.
7. Define an LSL "new" operator which constructs an association.
8. Define LSL "image" operators which return the members of the association.
9. Use axiomatic equations to describe the behavior of the "new" operator.
10. Use axiomatic equations to describe the behavior of the "image" operators.
11. Use axiomatic equations to describe the multiplicity of the association.

Figure 4.4 shows the ULARCH for the *Owns* association between a *Customer* class and an *Account* class in the Bank example. In the example, a customer can own zero or more accounts, while an account can only be owned by one customer.

[1] LSL stands for *Larch Shared Language*, which is used to write specifications in a form that is independent of any specific programming language (GH93).

```
\begin{spec} %LinkTheory
Own: trait
 includes Customer, Account
 introduces
  a-customer: Own-Link -> Cust
  an-account: Own-Link -> Acct
  new-Own-Link: Cust, Acct -> Own-Link
 asserts \forall c: Cust, a: Acct
  a-customer(new-Own-Link(c, a)) = c;
  an-account(new-Own-Link(c, a)) = a
\end{spec}\\
```

```
\begin{spec} %AssociationTheory
Owns: trait
 includes
  Set(Owns for C, Own for E)
 introduces
   new-Owns: O, Cust, Acct -> O
   image: O, Cust -> Accounts
   image: O, Acct -> Customers
   does-own: O, Cust, Acct -> Bool
 asserts \forall o: O, c: Cust, a: Acct, x: Own-Link
   size(image(o, c)) >= 0;
   size(image(o, a)) =1;
   (in(x, o) \and (a-customer(x) = c)) ==
       in(an-account(x), image(o, c));
   (in(x, o) \and (an-account(x) = a)) ==
       in(a-customer(x), image(o, a));
   new-Owns = empty-set;
   does-own(new-Owns, c, a) = false;
   does-own(0, c, a) == (in(c, image(o, a)) \and
        in(a, image(o, c)))
  \end{spec}\\
```

Figure 4.4 Association and Link in modified ULARCH

4.3.2 Representing Aggregation and Attributes. In Lin's mappings from OMT to ULARCH, the aggregation property is captured using the LSL *includes* clause; each component object's trait is included in the aggregate's trait (Lin94). Unfortunately, the inheritance property is also captured the same way. In order to properly model aggregation in O-SLANG, it was necessary to distinguish between aggregation and inheritance. To do this, the aggregation mappings were changed to the following steps:

- Create an LSL sort representing the trait using tuple notation.
- For each component, add a field in the tuple, and include the component object's trait. If the component object is a set, create an LSL sort for the set and include the Set trait, renaming the container sort, C, to the set sort and the element sort, E, to the component object's trait.

This change to the OMT mappings still could cause ambiguity, since Lin's mappings allowed the tuple notation to be used for attributes. To prevent this conflict, the attribute mappings were changed so that attributes are modeled in ULARCH using LSL operators only.

4.3.3 Representing Single and Multiple Inheritance. As stated in Section 4.3.2, both inheritance and aggregation require the use of the LSL *includes* clause. Even after the changes were made to the OMT mappings for aggregation, it was still necessary to distinguish between an includes clause for inheritance and for aggregation. This was accomplished by changing the mapping for inheritance to include the LSL *renames* notation. The presence of the renames notation indicates inheritance. To extend Lin's mappings to cover multiple inheritance, simply include, with the renames clause, the trait for each

object being inherited. Also, axioms must be built to describe the behavior of operators in the inheriting trait on attributes in the inherited trait, and visa versa. Similarly, if any event associated with the inheriting trait affects a state from the inherited trait, a state table entry must be defined to describe the new state and what actions or send events to send. The reverse also holds true: events from the inherited trait which affect the state of the inheriting trait must also be described with a state table entry. After the changes, the steps for transforming inheritance to ULARCH are:

- For each object that inherits from a parent(s), include the associated parent trait(s) in the LSL *includes* clause, using rename notation to indicate inheritance.
- For each LSL operator in the inheriting object, if the operator affects the value of an attribute in the parent trait, describe the behavior using an axiomatic equation.
- For each LSL operator in the parent trait, if the operator affects the value of an attribute in the inheriting object, describe the behavior using an axiomatic equation.
- For each *EventTheory* object associated with an inheriting object, if the event affects a state inherited from the parent trait(s), define a state table entry for the inheriting object which describes the effects.
- For each *EventTheory* object associated with a parent trait(s), if the event affects a state in the inheriting object, define a state table entry for the inheriting object which describes the effects.

4.4 Changes to State Transition Table Model

Evaluation of Lin and Wabiszewski's state transition table model revealed that two changes were required. First, the grammar for the state transition table did not allow for entries without receive events. In OMT, state transitions can occur when certain guard conditions are met. To capture this concept, the state transition table grammar was modified so that a receive event is optional in an entry in the table. The second change was made to distinguish between actions and send events in OMT. In O-SLANG, actions are methods, while send events are calls to events. As with states and events, actions and send events are different entities with different semantics. To capture this difference, the state action column was broken up into two separate columns, one for actions and one for send events. This required the addition of two new objects in the state transition table domain model and changes to the grammar to reflect the new column in the table. Tables 4.1 and 4.2 show the state transition table for the pump in the pump domain example before and after the changes to the state transition table model.

Table 4.1 Original Pump State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action
PumpInitialState	NewPump			PumpDisabled	
PumpDisabled	EnablePump			PumpEnabled	updatePump, StartPumpMotor, ResetDisplay
PumpDisabled	DisablePump			PumpDisabled	
PumpDisabled	OverHeat			PumpDisabled	
PumpEnabled	DisablePump			PumpDisabled	
PumpEnabled	EnablePump			PumpEnabled	
PumpEnabled	OverHeat			PumpDisabled	

Table 4.2 Modified State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
PumpInitialState	NewPump			PumpDisabled		
PumpDisabled	EnablePump			PumpEnabled	updatePump	StartPumpMotor; ResetDisplay
PumpDisabled	DisablePump			PumpDisabled		
PumpDisabled	OverHeat			PumpDisabled		
PumpEnabled	DisablePump			PumpDisabled		
PumpEnabled	EnablePump			PumpEnabled		
PumpEnabled	OverHeat			PumpDisabled		

4.5 Summary

Once the target language was identified, Lin and Wabiszewski's unified model had to be carefully evaluated to determine if it contained all of the information needed to build O-SLANG specifications. This evaluation highlighted several areas that required change. Modifications were made to add information to the unified model and to eliminate ambiguities. Adding the boolean annotated attribute was necessary to ensure that transformations do not loop infinitely. Once implemented, these changes set the stage for the next phase of extending the formal object transformation process: defining the transformations from ULARCH to O-SLANG.

V. Definition of Unified Model to Canonical Model Transformations

5.1 Introduction

In Chapter III, O-SLANG was established as the canonical model. Based on this choice of representation, Chapter IV described modifications to ULARCH. These changes were needed to ensure that the information required to capture the object-oriented semantics of OMT in O-SLANG were available in ULARCH. This concept is demonstrated in Figure 5.1. The modified version of ULARCH, which includes the state transition table and LARCH portions of the unified model, appears on the left-hand side of the figure. The canonical model, O-SLANG, appears on the right. The next step in extending the formal object transformation process was to define the actual transformations from ULARCH to O-SLANG so that the OMT semantics of ULARCH are properly transformed into equivalent O-SLANG representations. These transformations are the mappings shown in Figure 5.1.

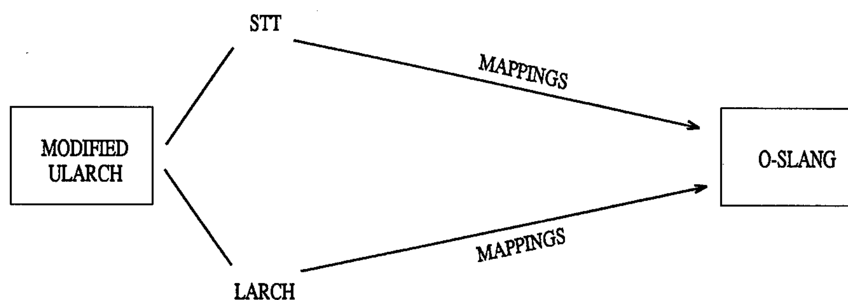

Figure 5.1 Conceptual View of Mappings

The mappings have the form $A \rightarrow B$, where A represents objects in ULARCH abstract syntax trees and B represents objects in an O-SLANG abstract syntax tree. This chapter presents these mappings by showing how each object in ULARCH is used to create the corresponding object(s) in O-SLANG. The mappings are covered by first looking at trans-

formations from the LARCH portion of ULARCH to O-SLANG, then at transformations from the state transition table portion of ULARCH to O-SLANG, and then finally at some additional transformations which require information from various parts of the unified model.

5.2 LARCH to O-SLANG Transformations

In ULARCH, the top level objects are *theory presentations*, which are described as LSL traits. Each theory presentation is composed of a *theory-id* and a *body*. Each body is composed of an *includes clause*, an *introduces clause*, and an *asserts clause*. This section outlines the mappings from ULARCH objects to O-SLANG objects in terms of LARCH traits, as well as in terms of tuple objects, which represent aggregation in the unified model.

5.2.1 ObjectTheory Mapping. In the unified model, the *ObjectTheory* trait represents an object instance. The O-SLANG counterpart is a *Class Specification*; however, there is more information contained in an O-SLANG *Class* than a ULARCH *ObjectTheory*. For example, a *Class* has state and object communication information while an *ObjectTheory* does not. There is a complete mapping from an *ObjectTheory* to a *Class*, but an *ObjectTheory* is not sufficient for building a complete *Class*. The mappings from an *ObjectTheory* and its subcomponents to a *Class* are shown in Figure 5.2, while Figure 5.3 shows the O-SLANG *Class Specification* created from the *ObjectTheory* for a *SophisticatedPump* object.

5.2.2 StateTheory Mapping. A ULARCH *StateTheory* trait defines a state of the object which appears in its includes clause. Axioms in the asserts clause describe the valid attribute ranges for the state (Lin94). The *StateTheory* and its subcomponents map di-

- theory-id → class-id
- theory-id → class-sort
- trait-ref and renames → inherited sorts
- trait-ref and no renames → import in imports block
- trait-parameter → sort-axiom
- operator → attribute in attributes block
- operator domain → attribute domain
- operator domain → sort in sorts block
- operator range → attribute range
- operator range → sort in sorts block
- axioms → axioms in axiom block

Figure 5.2 ObjectTheory Mappings

```
\begin{spec} %ObjectTheory
SophisticatedPump: trait
  includes Pump(P for P), Integer
  introduces
    volumeSelect: SP -> Int
    amountSelect: SP -> Int
\end{spec}
```

```
class SophisticatedPump is
  class-sort SophisticatedPump < Pump
  import Pump
  sort SP
  sort-axioms SophisticatedPump = SP
  attributes
    volumeSelect: SP -> Integer
    amountSelect: SP -> Integer
end-class
```

Figure 5.3 ObjectTheory Transformation for Sophisticated Pump

rectly to state and axiom subcomponents of the *Class Specification* which corresponds to its included trait. Mappings from the *StateTheory* components to their O-SLANG counterparts are provided in Figure 5.4. Figure 5.5 shows the mapping from the ULARCH trait *Overdrawn* to a state in the *Account Class Specification.*

- theory-id → operation-id of state operation in states block
- operator domain → state operation domain in states block
- operator range → state operation range in states block
- axioms → state invariant axioms in axioms block

Figure 5.4 StateTheory Mappings

```
\begin{spec} %StateTheory
Overdrawn: trait
  includes Account
  introduces
    OverdrawnState: Acct -> Bool
  asserts \forall a:Acct
    balance(a) < 0
\end{spec}\\
```

```
class Account is
        .
        .
  state-attributes
    AccountState: Account -> Account-State
        .
        .
  states
    Overdrawn: -> Account-State
        .
        .
  axioms
    AccountState(a) = Overdrawn => (balance(a) < 0)
        .
        .
end-class
```

Figure 5.5 StateTheory Transformation for Overdrawn

5.2.3 EventTheory Mapping. *EventTheory* traits define receive events for the object which appears in its includes clause. The trait name, contained in the theory-id, becomes the name of the event operation, while any parameters which appear in the domain of the *EventTheory* become part of the domain of the event operation and part of the sorts block of the corresponding *Class Specification.* Since events in O-SLANG operate on objects, the domain of the event operation includes the class-sort of the corresponding

Class Specification. The range of the event operation is also the class-sort of the *Class Specification*, indicating that events return the object which they operate on. The mappings are outlined in Figure 5.6, while Figure 5.7 demonstrates the transformations on the *StartPumpMotor* event.

- theory-id → operation-id of event operation in events block
- operator domain → domain of event operation in events block
- operator domain → sort in sorts block
- axioms → axioms in axioms block

Figure 5.6 EventTheory Mappings

```
\begin{spec} %EventTheory
StartPumpMotor: trait
  includes Motor
  introduces
    start-pump-motor : -> Bool
\end{spec}\\
```

```
class Pump is
  .
  .
  events
    StartPumpMotor: Motor -> Motor
  .
  .
end-class
```

Figure 5.7 EventTheory Transformation for StartPumpMotor

5.2.4 FunctionalTheory Mapping. In the unified model, the *FunctionalTheory* depicts data transformations in the OMT Functional Model. An operator models the transform process, while axioms in the asserts clause define the behavioral constraints of the operator. This maps directly to a method and axioms in O-SLANG. The method and axioms become part of the *Class Specification* which corresponds to the *FunctionalTheory*'s included trait. The operator name becomes the name of the method operation, while parameters appearing in the domain of the operator become part of the domain of the method operation and are declared in the sorts block of the corresponding *Class Specification.* As with event operations, methods operate on objects. Again, this means

that the class-sort of the corresponding *Class Specification* is in the domain of, and is the range of, the method operation. Figure 5.8 describes the *FunctionalTheory* mappings. The mappings are demonstrated in Figure 5.9 on the *Credit-Acct FunctionalTheory.*

- theory-id → operation-id of method operation in methods block
- operator domain → domain of method operation in methods block
- operator domain → sort in sorts block
- axioms → axioms in axioms block

Figure 5.8 FunctionalTheory Mappings

```
\begin{spec} %FunctionalTheory
Credit-Acct: trait
  includes Account
  introduces
    credit-acct: Acct, Amnt -> Acct
  asserts \forall ac: Acct, am: Amnt
    balance(credit-acct(ac, am)) =
       (balance(ac) + am)
\end{spec}\\
```

```
class Account is
     .
     .
 methods
   credit-acct: Acct, Amnt -> Acct
     .
     .
 axioms
   balance(credit-acct(ac, am)) =
     (balance(ac) + am);
     .
     .
 end-class
```

Figure 5.9 FunctionalTheory Transformation for Credit-Acct

5.2.5 LinkTheory Mapping. The *LinkTheory* object was modeled directly after the O-SLANG *Link Specification.* With a few exceptions, the mapping is essentially one-to-one. One exception is the sort names for the objects in the *LinkTheory*; the sort names are replaced with the corresponding O-SLANG class-sort names. Another exception is the "new" operator which becomes an event operation versus an attribute. Also, any references to the "new" operator in the trait axioms are replaced with the name of the "create" method. This is because events do not affect the values of attributes in O-SLANG, only

methods do. The mappings are presented in Figure 5.10, while Figure 5.11 demonstrates the effect of the mappings on the *Own* trait.

- theory-id concatenated with "-Link" → link-id
- theory-id concatenated with "-Link" → class-sort
- operator → attribute in attributes block
- operator domain → attribute domain
- operator domain → sort in sorts block
- operator range → attribute range
- operator range → sort in sorts block
- "new" operator → method in methods block
- axioms → axioms in axiom block
- "new" operator reference in axiom → "create" method reference in axiom

Figure 5.10 LinkTheory Mappings

```
\begin{spec} %LinkTheory
Own: trait
  includes Customer, Account
  introduces
    a-customer: Own-Link -> Cust
    an-account: Own-Link -> Acct
    new-Own-Link: Cust, Acct -> Own-Link
  asserts \forall c: Cust, a: Acct
    a-customer(new-Own-Link(c, a)) = c;
    an-account(new-Own-Link(c, a)) = a

\end{spec}\\
```

```
link Own-Link is
  class-sort Own-Link
  sort Account, Customer
  attributes
    a-customer: Own-Link -> Customer
    an-account: Own-Link -> Account
  events
    new-Own-Link: Customer, Account -> Own-Link
  axioms
    a-customer(create-Own-Link(a, c)) = c;
    an-account(create-Own-Link(a, c)) = a
end-link
```

Figure 5.11 LinkTheory Transformation for Own

5.2.6 AssociationTheory Mapping. As with the *LinkTheory* object, the *AssociationTheory* object was modeled directly after its O-SLANG counterpart, the *Association Specification.* Again the mappings are essentially one-to-one, as can be seen in Figures 5.12 and 5.13.

- theory-id → assoc-id
- theory-id → class-sort
- included set of links → link-class
- operator → method in methods block
- operator domain → method domain
- operator domain → sort in sorts block
- operator range → method range
- operator range → sort in sorts block
- "new" operator → method in methods block
- axioms → axioms in axiom block

Figure 5.12 AssociationTheory Mappings

```
\begin{spec} %AssociationTheory
Owns: trait
  includes
    Set(Owns for C, Own for E), Own
  introduces
    new-Owns: O, Cust, Acct -> O
    image: O, Cust -> Accounts
    image: O, Acct -> Customers
    does-own: O, Cust, Acct -> Bool
  asserts \forall o: O, c: Cust,
                  a: Acct, x: Own-Link
    Size(image(o, c)) >= 0;
    Size(image(o, a)) = 1;
    (in(x, o) \and (a-customer(x) = c)) ==
      in(an-account(x), image(o, c));
    (in(x, o) \and (an-account(x) = a)) ==
      in(a-customer(x), image(o, a));
    new-Owns = empty-set;
    does-own(new-Owns, c, a) = false;
    does-own(o, c, a) ==
      (in(c, image(o, a)) \and
       in(a, image(o, c)))
\end{spec}\\
```

```
association Owns is
  class-sort Owns
  link-class Own-Link
  sort Customer, Account, Bool,
    O, Customers, Accounts
  sort-axioms Owns = O
  methods
    does-own: O, Customer, Account -> Bool
    image: O, Account -> Customers
    image: O, Customer -> Accounts
  events
    new-Owns: Customer, Account -> Owns
  axioms
    Size(image(o, c)) >= 0;
    Size(image(o, a)) = 1;
    does-own(new-Owns(o, c, a)) = false;
    does-own(o, c, a) <=>
      (in(c, image(o, a)) &
       in(a, image(o, c)));
end-association
```

Figure 5.13 AssociationTheory Transformation for Owns

5.2.7 Tuple Mapping. As discussed in Section 4.3.2, tuples are used in the unified model to represent aggregation. This concept is captured in an *Aggregate Specification* in O-SLANG. An *Aggregate* allows multiple classes to be combined to specify system or subsystem functionality. This is done through the use of the colimit operation (DBH95). While the colimit operation does not exist in ULARCH, it is still possible to build portions of an *Aggregate* from a tuple and the *ObjectTheory* of which it is a part. The remaining parts of the *Aggregate* deal with object communication and inheritance, and they are created from state transition table information and inheritance information. There are also some additional nodes and arcs that must be generated in certain cases. For each set component, a node must be created for a new copy of the *Set Class Specification.* An arc must then be created which maps the class-sort *Set* to the class-sort of the set component, and the sort *E*, which is a sort in the *Set Class Specification*, to the class-sort of the members of the set component. A node must also be included for the *Integer Class Specification.* When multiple set components exist, arcs must be added to ensure that each *Set Specification* uses the same copy of *Integer.* The mappings in Figure 5.14 represent the portions of an *Aggregate Specification* which can be built from a ULARCH tuple. An example of these mappings appears in Figure 5.15.

- theory-id of parent theory concatenated with "-aggregate" → agg-id
- single object field in tuple → object node in *Aggregate*
- single object field in tuple → object-valued attribute in corresponding *Class Specification*
- set object field in tuple → Set node in *Aggregate*
- set object field in tuple → object node in *Aggregate*
- set object field in tuple → class object-valued attribute in corresponding *Class Specification*

Figure 5.14 Tuple Mappings

```
\begin{spec} %ObjectTheory
Pump(P): trait
  includes
    Set(DisplaySet for C, Display for E),
    Set(GHASet for C, GunHolsterAssembly for E),
    Set(CMASet for C, ClutchMotorAssembly for E),
    PumpController, Kept-In
  P tuple of gun-holster-assembly : GHASet,
             clutch-motor-assembly : CMASet,
             pump-controller  : PC,
             display   : DisplaySet,
             kept-in: Kpt-In
             .
             .
\end{spec}\\

aggregate Pump-aggregate is
  nodes
    GunHolsterAssembly-Class, ClutchMotorAssembly-Class,
    PumpController, Display-Class, Kept-In, Integer,
    SET-1: Set, SET-2: Set, SET-3: Set, SET-4: Set
  arcs
    SET-1 -> GunHolsterAssembly-Class:
      {Set -> GunHolsterAssembly-Class, E -> GunHolsterAssembly},
    SET-2 -> ClutchMotorAssembly-Class:
      {Set -> ClutchMotorAssembly-Class, E -> ClutchMotorAssembly} ,
    SET-3 -> Display-Class: {Set -> Display-Class, E -> Display},
    SET-4 -> Kept-In: {Set -> Kept-In, E -> KI-Link},
    Integer -> SET-1: {},
    Integer -> SET-2: {},
    Integer -> SET-3: {},
    Integer -> SET-4: {}
end-aggregate
```

Figure 5.15 Tuple Transformation for Pump ObjectTheory

5.3 State Transition Table to O-SLANG Transformations

The state transition table defines the state and communication behavior of an object. Each row in the table is a *StateEntry* object in the state transition table domain model which represents the behavior of an object in response to a particular receive event, set of guard conditions, or a combination of both. In O-SLANG, this information is captured in *Event Specifications*, axioms in *Class Specifications*, and nodes and arcs in *Aggregate Specifications.* This section describes the mappings from entries in a state transition table to various objects in O-SLANG. Since the same information in the state transition table is used to make multiple O-SLANG objects, the discussion of the mappings is organized by the O-SLANG objects produced in the transformations.

5.3.1 Single and Multiple Event Specifications. If an entry in a state transition table has a send event, then an O-SLANG *Event Specification* must be created. The *Event Specification*, along with the colimit operations specified by an *Aggregate Specification*, provides a line of communication between objects. For each send event, a separate *Event Specification* is built. Also, an object-valued attribute is added to the sender's *Class Specification* for each receiving object. The mappings from send events to single *Event Specifications* is shown in figure 5.16.

When a send event is received by multiple objects, then a multiple *Event Specification* must also be built. An event operation must be created in the *Event Specification* for the send event and for each object which will receive the send event. For example, if some event *Event1* is sent to two different objects, then there will be three event operations in the multiple *Event Specification*, one for the send event and two for the receiving objects.

- name of send event → event-id
- name of send event → event class-sort
- name of send event → operation-id of event operation
- parameters of send event → domain of event operation
- parameters of send event → sorts in sort block
- name of receiver concatenated with "-obj" → object-valued attribute

Figure 5.16 Single Event Mappings

The event operations for the receivers need different sorts from the class-sort of the *Event Specification.* These sorts are eventually unified with the class-sorts of the receivers to enable communication. Also, axioms must be created in the *Event Specification* to connect the send event to each of the event operations. The mappings from send events to multiple *Event Specifications* can be seen in Figure 5.17. Figure 5.18 demonstrates an example where the event *OverHeat* is sent to three different objects.

- name of send event → event-id
- name of send event → event class-sort
- name of send event → operation-id of event operations
- parameters of send event → domain of event operations
- parameters of send event → sorts in sort block

Figure 5.17 Multiple Event Mappings

5.3.2 Receive Event and Transition Event Axioms. An entry in a state transition table indicates that some action must be taken due to the receipt of an event, the satisfaction of guard conditions, or both. That action could be a change of state, invocation of a method or operation, or the sending of an event(s). In order to define the behavior depicted in the state transition table entry, axioms must be created in the *Class Specifica-*

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
⋮	⋮	⋮	⋮	⋮	⋮	⋮
MotorRunning			*temp* > 300	MotorDisabled		OverHeat
⋮	⋮	⋮	⋮	⋮	⋮	⋮

```
event OverHeat is
  class-sort OverHeat
  events
    OverHeat: OverHeat -> OverHeat
end-event
```

```
event OverHeat-mult is
  class-sort OverHeat
  sort OBJ-1, OBJ-2, OBJ-3
  attributes
    OBJ-1-obj: OverHeat -> OBJ-1
    OBJ-2-obj: OverHeat -> OBJ-2
    OBJ-3-obj: OverHeat -> OBJ-3
  events
    OverHeat: OverHeat -> OverHeat
    OverHeat: OBJ-1 -> OBJ-1
    OverHeat: OBJ-2 -> OBJ-2
    OverHeat: OBJ-3 -> OBJ-3
  axioms
    OBJ-1-obj(OverHeat(o)) = OverHeat(OBJ-1-obj(o));
    OBJ-2-obj(OverHeat(o)) = OverHeat(OBJ-2-obj(o));
    OBJ-3-obj(OverHeat(o)) = OverHeat(OBJ-3-obj(o))
end-event
```

Figure 5.18   Send Event Transformation for OverHeat

tion which corresponds to the object described by the state transition table. For an entry with no receive event, the axiom built has the form:

(object-state(x) = current-state & guard-conditions) ⇒
    (object-state(x) = next-state &
     receive-obj-1(x) = send-event-1(receive-obj-1(x), send-params-1) &
     ⋮
     receive-obj-n(x) = send-event-n(receive-obj-n(x), send-params-n) &
     action-1(action-params-1) &
     ⋮
     action-n(action-params-n))

For an entry with a receive event, the axiom built has the form:

(object-state(x) = current-state & guard-conditions & receive-event(x, params)) ⇒
    (object-state(x) = next-state &
     receive-obj-1(x) = send-event-1(receive-obj-1(x), send-params-1) &
     ⋮
     receive-obj-n(x) = send-event-n(receive-obj-n(x), send-params-n) &
     attr-equal(receive-event(params), action-1(action-params-1)) &

⋮

attr-equal(receive-event(params), action-n(action-params-n)))

In these axioms, *object-state* is a state attribute which contains the value of the current state of the object, while *receive-obj* is an object-valued attribute which points to the receiver of the send-event. The terms *current-state, guard-conditions, receive-event, params, next-state, send-event, send-params, action* and *action-params* all come from the state transition table entry.

5.4 Additional Transformations

The information in the unified model is organized differently than in the canonical model. Because of this difference, the transformations from ULARCH to O-SLANG are not entirely straight forward. In some instances, previous information captured in one portion of the unified model must be known in order to transform another portion. A good example of this is in building axioms to describe the effects of methods and operations on attributes, and events on state attributes. In a *Class*, the effect of each method and operation must be described over each attribute. The same holds true for events and state attributes. Additionally, if the *Class* is a subclass, then its operations, methods, and events must be described over its superclass's attributes and states, and visa versa. In order to build all of these axioms, all *ObjectTheory* traits must first be processed so that the entire system structure is known and inheritance relationships can be determined. In other instances, information to transform part of the unified model is obtained from part of the canonical model that has already been built. These types of situations require information to be stored in separate data structures, as will be discussed in Chapter VI.

This section describes those mappings from ULARCH to O-SLANG which require information from different portions of the unified model and the canonical model.

5.4.1 Object Class Specifications. *ObjectTheory* traits in ULARCH depict object instances. Section 5.2.1 showed how an *ObjectTheory* maps to a *Class Specification.* To capture the concept of a class of objects, a *Class Specification* for an object class must also be created. The object class is a set of the *Class Specification* objects created from the *ObjectTheory.* It has the same events as the instance *Class Specification*; however, the events are of the form:

Event: object-Class $\rightarrow$ object-Class

Each event takes in an object class and returns the object class. Axioms connect the object class events to the class events as follows:

in(x, y) $\Leftrightarrow$ in(event(x), event(y))

In this example, x is of an instance type, while y is of the corresponding object class type. When an event is sent to an object class, it has the same effect as sending the event to each object instance in the object class. The mappings for building an object class *Class Specification* are presented in Figure 5.20. An example of an O-SLANG object instance and its corresponding object class are shown in Figure 5.21.

- theory-id concatenated with "-Class" $\rightarrow$ class-id
- theory-id concatenated with "-Class" $\rightarrow$ class-sort
- theory-id $\rightarrow$ contained-class

Figure 5.19 ObjectTheory Mappings

- theory-id → operation-id of event operation in events block

Figure 5.20 EventTheory Mappings

```
class Pump is
  class-sort Pump
      .
      .
  events
      .
      .
    OverHeat: Pump -> Pump
    EnablePump: -> Pump
    DisablePump: -> Pump
      .
      .
end-class
```

```
class Pump-Class is
  class-sort Pump-Class
  contained-class Pump
  events
    new-Pump-Class: -> Pump-Class
    OverHeat: Pump-Class -> Pump-Class
    EnablePump: Pump-Class -> Pump-Class
    DisablePump: Pump-Class -> Pump-Class
  axioms
    create-Pump-Class = empty-set;
    new-Pump-Class = create-Pump-Class;
    in(p, pc) <=>
      in(OverHeat(p), OverHeat(pc));
    in(p, pc) <=>
      in(EnablePump(Pump-207), EnablePump(pc));
    in(p, pc) <=>
      in(DisablePump(p), DisablePump(pc))
end-class
```

Figure 5.21 Object Class Transformation for Pump-Class

5.4.2 New Events and Create Methods. For each *Class Specification*, there must be a way to create an instance of the *Class*. This is modeled using "new" events which invoke "create" methods. The "new" event can be constructed in two different ways. First, the user (domain modeler), can describe it in the state transition table for the domain model [1]. This allows the user to initialize components of an aggregate by passing them as parameters to the "new" event. Alternatively, the user can omit the definition of a "new" event, and one will be created automatically. The default event will have no domain. For each "new" event, a "create" method is automatically created with a signature that is identical to the event. Since object class *Class Specifications* are created automatically, their "new" event and "create" method are also generated automatically. The mappings to "new" events and "create" methods are shown in Figures 5.22 and 5.23.

[1] If an object in the unified model has states, the "new" event behavior **must** be described in a state transition table entry.

- "new-" concatenated with class-id → operation-id "new" event operation
- parameters of "new" event → domain of "new" event operation
- "create-" concatenated with class-id → operation-id of "create" method operation
- parameters of "new" event → domain of "create" method operation

Figure 5.22 User Defined "new" Event

- "new-" concatenated with class-id → operation-id "new" event operation
- "create-" concatenated with class-id → operation-id of "create" method operation

Figure 5.23 Default "new" Event

5.4.3 attr-equal Operation. For a receive event to invoke a method, its behavior must be linked to the behavior of the method. This is done by stating the equivalence of the receiving object's attributes before the arrival of the receive event with the attributes after the invocation of the method. Unfortunately, this cannot be done directly since by definition events can only affect state attributes, not regular attributes. To get around this problem, an operation called "attr-equal" is defined with the signature and behavior described as follows:

$$\text{attr-equal: object-sort, object-sort} \rightarrow \text{object-sort}$$

$$\begin{aligned} \text{attr-equal}(x, y) \Rightarrow (&\text{attr-1}(x) = \text{attr-1}(y)\ \& \\ &\vdots \\ &\text{attr-n}(x) = \text{attr-n}(y)) \end{aligned}$$

This operation must be created for any *Class Specification* with attributes. Because some attributes are added due to object communication, the operation must be built after all state transition tables have been processed.

5.4.4 Additional Axioms. There are some axioms which can only be built after the entire structure of the domain model has been transformed into O-SLANG. These axioms fall into two categories: axioms describing the behavior of the "attr-equal" operation, and axioms describing an object's functional inheritance. The first case was described in Section 5.4.3. For the second case, axioms describing the effect of each of the subclass's methods on the superclass's attributes can be included in the *asserts* clause of the subclass's *FunctionalTheory* trait. Axioms describing the effect of the superclass's methods on the subclass's attributes can be included in the *asserts* clause of the subclass's *ObjectTheory* trait. For any method and any attribute, if there is no effect to be described then the axioms can be omitted from the ULARCH and they will automatically be generated. These axioms can only be built after the entire structure of the domain model is known. The default axioms will have the following form:

attr-n(method-m(x)) = attr-n(x)

5.4.5 Aggregate Specification Nodes and Arcs. As discussed in Section 5.2.7, *Aggregate Specifications* are created from tuple objects, but the *Aggregate* is not complete at that point. It also must capture information regarding 1) communication between objects, 2) information regarding associations, and 3) information regarding inheritance. This information is depicted in nodes and arcs. They define morphisms between different sorts and different operations which will be unified in the colimit operation. In each of these three cases, nodes and arcs must be added after all state transition table entries and ULARCH traits are processed. The mappings for object communication, associations,

and inheritance are shown in Figures 5.24 through 5.27. Figure 5.28 shows an aggregate specification for the bank object in the bank domain example.

- name of send event → node
- name of sender → node
- name of receiver → node
- name of sender and send event → arc
- name of receiver and send event → arc

Figure 5.24 Object Communication Mappings - Single Event

- name of send event → nodes
- name of send event concatenated with "-mult" → node
- name of sender → node
- name of receivers → nodes
- name of sender and send event → arc
- name of receivers and send event → arcs
- name of send event concatenated with "-mult" and name of send event→ arcs
- sorts in multiple event *Event Specification* → sort-axioms

Figure 5.25 Object Communication Mappings - Multiple Event

- association name → node
- sorts in association → nodes
- association name → arc
- sorts in association → arcs

Figure 5.26 Association Mappings

- superclasses → nodes
- subclasses → nodes
- superclasses, subclasses → arcs

Figure 5.27 Inheritance Mappings

5.5 Summary

This chapter outlined the second phase of extending Lin's formal object transformation process. Mappings were defined from the unified model objects to the O-SLANG objects in the canonical model representation, thus demonstrating that the canonical model can capture all of the OMT concepts embodied in the unified model. Because the information in the unified model is organized differently than in the canonical model, some transformations require input from different portions of the unified model in order to create canonical model objects. This had a definite impact on the design and implementation of the transformations, as will be seen in Chapter VI.

```
aggregate Bank-aggregate is
  nodes
    Person-Class, Customer-Class, Employee-Class,
    Cust-Employee-Class, Teller-Class, Executive-Class,
    Console-Class, Account-Class, Checking-Class, Savings-Class,
    Combined-Class, Archive-Class, Owns, Integer, SET-1: Set,
    SET-2: Set, SET-3: Set, SET-4: Set, SET-5: Set, SET-6: Set,
    SET-7: Set, SET-8: Set, SET-9: Set, SET-10: Set,
    SET-11: Set, SET-12: Set, SET-13: Set, Credit, Debit, Close,
    WriteCheck, SetRate, ComputeInterest, ArchCredit, ArchDebit, ArchClose
  arcs
    SET-1 -> Person-Class: {Set -> Person-Class, E -> Person},
    SET-2 -> Customer-Class: {Set -> Customer-Class, E -> Customer},
    SET-3 -> Employee-Class: {Set -> Employee-Class, E -> Employee},
    SET-4 -> Cust-Employee-Class: {Set -> Cust-Employee-Class, E -> Cust-Employee},
    SET-5 -> Teller-Class: {Set -> Teller-Class, E -> Teller},
    SET-6 -> Executive-Class: {Set -> Executive-Class, E -> Executive},
    SET-7 -> Console-Class: {Set -> Console-Class, E -> Console},
    SET-8 -> Account-Class: {Set -> Account-Class, E -> Account},
    SET-9 -> Checking-Class: {Set -> Checking-Class, E -> Checking},
    SET-10 -> Savings-Class: {Set -> Savings-Class, E -> Savings},
    SET-11 -> Combined-Class: {Set -> Combined-Class, E -> Combined},
    SET-12 -> Archive-Class: {Set -> Archive-Class, E -> Archive},
    Integer -> SET-1: {}, Integer -> SET-2: {},
    Integer -> SET-3: {}, Integer -> SET-4: {},
    Integer -> SET-5: {}, Integer -> SET-6: {},
    Integer -> SET-7: {}, Integer -> SET-8: {},
    Integer -> SET-9: {}, Integer -> SET-10: {},
    Integer -> SET-11: {}, Integer -> SET-12: {},
    Integer -> SET-13: {},
    SET-13 -> Owns: {Set -> Owns, E -> Own-Link},
    SET-2 -> Owns: {Set -> Customers, E -> Customer},
    SET-8 -> Owns: {Set -> Accounts, E -> Account},
    Credit -> Console-Class: {},
    Credit -> Account-Class: {Credit -> Account-Class},
    Debit -> Console-Class: {},
    Debit -> Account-Class: {Debit -> Account-Class},
    Close -> Console-Class: {},
    Close -> Account-Class: {Close -> Account-Class},
    WriteCheck -> Console-Class: {},
    WriteCheck -> Checking-Class: {WriteCheck -> Checking-Class},
    SetRate -> Console-Class: {},
    SetRate -> Savings-Class: {SetRate -> Savings-Class},
    ComputeInterest -> Console-Class: {},
    ComputeInterest -> Savings-Class: {ComputeInterest -> Savings-Class},
    ArchCredit -> Account-Class: {},
    ArchCredit -> Archive-Class: {ArchCredit -> Archive-Class},
    ArchDebit -> Account-Class: {},
    ArchDebit -> Archive-Class: {ArchDebit -> Archive-Class},
    ArchClose -> Account-Class: {},
    ArchClose -> Archive-Class: {ArchClose -> Archive-Class},
    Acct -> Acct-Class: {}, Acct -> Checking: {}, Acct -> Savings: {},
    Checking -> Checking-Class: {}, Checking -> Combined: {},
    Savings -> Savings-Class: {}, Savings -> Combined: {},
    Combined -> Combined-Class: {},
    Person -> Person-Class: {}, Person -> Customer: {}, Person -> Employee: {},
    Employee -> Employee-Class: {}, Employee -> Exec: {},
    Employee -> Teller: {}, Employee -> Cust-Employee: {},
    Exec -> Exec-Class: {},
    Customer -> Customer-Class: {}, Customer -> Cust-Employee: {},
    Cust-Employee -> Cust-Employee-Class: {},
    Teller -> Teller-Class: {}
end-aggregate
```

Figure 5.28 Aggregate Transformation for Bank

VI. Design and Implementation of ULARCH to O-SLANG Transformations

6.1 Introduction

Chapter V defined a set of mappings from objects in the modified version of ULARCH to objects in O-SLANG. These mappings provided a description of how to build specific O-SLANG objects in a way which preserves the object-oriented semantics of the original OMT domain model. The next step in extending the formal object transformation process was to implement each of these rules in REFINE so that O-SLANG domain theories can be automatically generated from ULARCH representations of domain models. Before this implementation could begin, however, two tasks had to be accomplished. First, the mappings from Chapter V had to be correlated with specific language constructs in REFINE which match the *precondition* $\rightarrow$ *postcondition* semantics of transformations. Second, a mechanism for implementing the control structure of the transformation process had to be identified. Once the transformations were implemented, it was necessary to answer the question, *"Do the transformations produce unique normal forms?"* This concept of *unique normalization* [1] is key in rewrite systems. It guarantees that every term output by a rewrite system has exactly one normal form.

This chapter first outlines a detailed design and implementation of the ULARCH to O-SLANG mappings presented in Chapter IV, and then presents an analysis of the implemented transformations. The analysis includes a discussion of the results of applying the validation process defined in Section 3.4.1 along with a presentation of how rewrite system properties can be shown to hold for the transformations. By first showing that the individ-

[1] See Section 2.3.4.3 for a definition of unique normalization

ual transformations are semantically correct and then applying the rewrite properties of *termination* and *confluence*, the transformations as a whole are shown to be semantically correct.

6.2 Overview of Implementation

Recall from Chapter V that because of the organization of the information in the unified model, the ULARCH to O-SLANG mappings were grouped into three categories:

1. ULARCH traits to O-SLANG transformations
2. State transition table to O-SLANG transformations
3. Additional transformations

This grouping heavily influenced the design of the transformations: each category became a step in the overall transformation process. This section describes the REFINE language constructs and control structure used to implement the mappings provided in Chapter V, along with detailed presentations of the implementation of each mapping.

6.2.1 REFINE Language Constructs. REFINE provides two different constructs for implementing the semantics of transformations: *transforms* and *rules*. Transforms have the form P **-->** Q, where P and Q are predicates and **-->** is a special "transform arrow." In short, some initial state described by P is transformed into a final state described by Q. Consider the example where some variable v is assigned the value 100 whenever $v < 100$. Using a transform, this can be implemented as $v < 100$ **-->** $v = 100$. Rules provide a way of encapsulating transforms in the same way as a function body. Essentially, rules are named transforms which can be parameterized (Ref90). Consider the following example rule:

```
rule RuleA(a-parameter: parameter-type)
  P(a-parameter) --> Q(a-parameter)
```

In this example, P is some property which $a - parameter$ might possess, and Q is a function which performs processing based on $a - parameter$. When *RuleA* is applied, if P is true, then the function Q is called. This is precisely the type of construct needed for implementing the ULARCH to O-SLANG mappings.

In the implementation of the mappings from Chapter V, each of the ULARCH mappings in Sections 5.2.1 through 5.2.7 became a rule, as well as the state transition table entry mapping. For example, the mapping for *ObjectTheory* traits defined in Section 5.2.1 was implemented by the following rule:

```
rule Trans-ULarch-ObjectTheory(Input-Object:user-object)
   ObjectTheory(Input-Object) & ~StateTheory(Input-Object) &
   ~LinkTheory(Input-Object) & ~Association(Input-Object) &
   ~EventTheory(Input-Object) & ~FunctionalTheory(Input-Object) &
   ~done-Transform(Input-Object) --> Make-Class-(Input-Object)
```

In *Trans-Ularch-ObjectTheory*, if *Input-Object* is an *ObjectTheory* trait, then the function *Make-Class-* is called to build an O-SLANG *Class-* object. Each of the rules created has a similar format.

6.2.2 Control Structure. The remaining issue that needed to be resolved for the implementation of the transforms was how to traverse the tree of ULARCH objects being transformed. REFINE provides a couple of ways to do this. In the first way, REFINE traversal functions can be used to apply rules to objects in an abstract syntax tree in either a bottom-up [2] or top-down [3] fashion. As each object in the tree is visited, the rules passed

[2] POSTORDER-TRANSFORM function
[3] PREORDER-TRANSFORM function

to the traversal function are applied to the object one at a time. The main difference in these functions is the order in which the objects are visited (Ref90). The second way to traverse abstract syntax trees is through the use of the *enumerate* ... *over* ... construct and the *DESCENDANTS-OF-CLASS* function. Consider the following transformation function:

```
function Update-Aggregates() =
  (enumerate Temp-Class over DESCENDANTS-OF-CLASS(Domain-Theory, 'Class-) do
      Update-AggCommunication(Temp-Class);
      Update-AggAssociation(Temp-Class))
```

In *Update-Aggregates*, the *enumerate* construct builds a set containing all objects in the tree rooted at *Domain-Theory* which are of type *Class-*. The variable *Temp-Class* then takes on the value of each member of the created set, one at a time, and is passed to the functions *Update-AggCommunication* and *AggAssociation* which update any associated *Aggregate-* objects to account for object communication and associations as described in Section 5.4.5.

Each of the above control structures was used in the implementation of the ULARCH to O-SLANG mappings. The *PREORDER-TRANSFORM* function was used to transform ULARCH objects and state transition table objects. This control structure is depicted in the following two rules:

```
rule Trans-ULarch(Input-Object:user-object)
   DomainTheory(Input-Object) -->
      Input-Object = preorder-transform(Input-Object,
                                ['Trans-ULarch-ObjectTheory,
                                 'Trans-ULarch-StateTheory,
                                 'Trans-ULarch-EventTheory,
                                 'Trans-ULarch-FunctionalTheory,
                                 'Trans-ULarch-LinkTheory,
                                 'Trans-ULarch-AssociationTheory,
                                 'Trans-Ularch-TupleObj])
```

```
rule Trans-STT(Input-Object:user-object)
   StateTable(Input-Object) -->
      Input-Object = preorder-transform(Input-Object,
                                  ['Trans-STT-StateEntry])
```

The *enumerates* construct was used to perform the additional processing transformations described in Section 5.4.

6.2.3 ULARCH to O-SLANG. As stated in Section 6.2.1, each of the high-level mappings in Sections 5.2.1 through 5.2.7 became a rule. The remaining mappings provided guidance on how to build O-SLANG objects. Each rule calls a function whose purpose is to build the equivalent high-level object in O-SLANG. For example, *Trans-ULarch-StateTheory* calls the function *Make-State*, which builds a state operation in the appropriate O-SLANG *Class-*. Those objects which become specifications in O-SLANG, i.e. *ObjectTheory*, *LinkTheory*, *AssociationTheory*, and *tuples*, follow the same basic sequence of events. First, the specification name and class-sort are created, followed by a call to a function which builds the specification body. This function then builds the body subcomponents based on the mappings in Chapter V. The remaining rules are described below in Figures 6.1 through 6.6.

```
rule Trans-ULarch-StateTheory(Input-Object:user-object)
   StateTheory(Input-Object) & ~ObjectTheory(Input-Object) &
   ~LinkTheory(Input-Object) & ~Association(Input-Object) &
   ~EventTheory(Input-Object) & ~FunctionalTheory(Input-Object) &
   ~done-Transform(Input-Object) --> Make-State(Input-Object)
```

Figure 6.1 *StateTheory* Transformation

```
rule Trans-ULarch-EventTheory(Input-Object:user-object)
   EventTheory(Input-Object) & ~ObjectTheory(Input-Object) &
   ~LinkTheory(Input-Object) & ~Association(Input-Object) &
   ~StateTheory(Input-Object) & ~FunctionalTheory(Input-Object) &
   ~done-Transform(Input-Object) --> Make-RecvEvent(Input-Object)
```

Figure 6.2 *EventTheory* Transformation

```
rule Trans-ULarch-FunctionalTheory(Input-Object:user-object)
   FunctionalTheory(Input-Object) & ~ObjectTheory(Input-Object) &
   ~LinkTheory(Input-Object) & ~Association(Input-Object) &
   ~StateTheory(Input-Object) & ~EventTheory(Input-Object) &
   ~done-Transform(Input-Object) --> Make-Funct(Input-Object)
```

Figure 6.3 *FunctionalTheory* Transformation

```
rule Trans-ULarch-LinkTheory(Input-Object:user-object)
   LinkTheory(Input-Object) & ~ObjectTheory(Input-Object) &
   ~Association(Input-Object) & ~StateTheory(Input-Object) &
   ~EventTheory(Input-Object) & ~FunctionalTheory(Input-Object) &
   ~done-Transform(Input-Object) --> Make-Link(Input-Object)
```

Figure 6.4 *LinkTheory* Transformation

```
rule Trans-ULarch-AssociationTheory(Input-Object:user-object)
   AssociationTheory(Input-Object) & ~ObjectTheory(Input-Object) &
   ~Link(Input-Object) & ~StateTheory(Input-Object) &
   ~EventTheory(Input-Object) & ~FunctionalTheory(Input-Object) &
   ~done-Transform(Input-Object) --> Make-Association(Input-Object)
```

Figure 6.5 *AssociationTheory* Transformation

```
rule Trans-ULarch-TupleObj(Input-Object:user-object)
   Tuple-Obj(Input-Object) &
   ~done-Transform(Input-Object) --> Make-Aggregate-(Input-Object)
```

Figure 6.6 *Tuple* Transformation

6.2.3.1 Axioms Transformation. O-SLANG axioms were created in one of two possible ways. First, some axioms were automatically built. An example of this is the axioms for describing state transitions. Strings were built according to the formats presented in Section 5.3, and then those strings were parsed using the O-SLANG parser and the *PARSE-FROM-STRING* function. Another example of automatically generated axioms are those which describe the *attr-equal* operation.

The second way axioms are created is from axioms in the ULARCH traits, such as those describing state invariants, guard conditions, or the behavior of LSL operators. To build these axioms, advantage was taken of the similarity of the syntax between LSL axioms and O-SLANG axioms. By restricting the user to writing axioms in the O-SLANG syntax, the ULARCH axioms could be pretty printed to a string, and then that string could be parsed by the O-SLANG parser with *PARSE-FROM-STRING.* To build axioms in this manner, the following two functions were used:

```
function Make-OslangAxiom(Temp-AxString: string): Axiom-Def =
   let(Temp-AxDef: Axiom-Def = nil,
       Temp-AxiomsBlock: AxiomsBlock = nil)
   Temp-AxiomsBlock <- parse-from-string(Temp-AxString, 'oslang);
  (enumerate Temp-Axiom over axiom-or-def(Temp-AxiomsBlock) do
       Temp-AxDef <- Temp-Axiom);
   Temp-AxDef

function Make-String-From-Object(Input-Object: user-object): string =
   let(Temp-String: string = "")
   Temp-String <- format(false, "~\\pp\\", Input-Object);
   Temp-String
```

Make-OslangAxiom returns an *Axiom-Def* object that is created by parsing the string *Temp-AxString* using the O-SLANG grammar. *Make-String-From-Object* creates a string from the output of the printing of an AST rooted at Input-Object, and returns it.

6.2.4 State Transition Table to O-SLANG. As described in Section 5.3, each entry in a state transition table is represented by a *StateEntry* object. It is used to create O-SLANG *Event* specifications and to build axioms describing state transition behavior. The state transition table rule is:

```
rule Trans-STT-StateEntry(Input-Object:user-object)
   StateEntry(Input-Object) & ~StateTable(Input-Object)&
   ~Ident-(Input-Object) & ~SendEvent(Input-Object) & ~
   done-Transform(Input-Object) --> Make-StateEntry(Input-Object)
```

If the *StateEntry* object dictates that an event must be sent, then the function *Make-SendEvent* is called to create an O-SLANG *Event.* If there are multiple objects which will receive the send event, then *Make-MultSendEvent* is called to create another O-SLANG *Event*, this time for the multiple event. The behavior defined by the *StateEntry* object is captured by calling either *Make-RecvEventAxiom* or *Make-TransAxiom* to build an axiom, depending on whether there is a receive event or not.

6.2.5 Extra Data Structures. To facilitate the building of O-SLANG objects, some data structures were needed above and beyond the ULARCH, state transition table, and O-SLANG ASTs. These data structures were tables, represented as sets or sequences, and maps. The extra data structures were as follows:

1. *AggTable* : Sequence containing tuples consisting of a class name and an *Aggregate*-specification where the object named by the class name is a component of the *Aggregate*-
2. *ClassSorts* : Set containing all of the declared class-sorts
3. *ClassSortMap* : Map from a class-sort to a sequence of equivalent sorts
4. *MultSendEventMap* : Map from a multiple send event name to a sequence of state table entrys
5. *AddedAttributes* : Set containing the names of all classes which attributes have been added to.

6. *DescribedAttrs* : A sequence of tuples which contain a Class- Spec, a method, and a sequence of attributes where the effects of the method on the attributes have already been described in an axiom.
7. *DescribedInheritedAttrs* : A sequence of tuples which contain a Class- Spec, a method, and a sequence of inherited attributes where the effects of the method on the attributes have already been described in an axiom.
8. *InitialStateMap* : Map from a class name to the name of the initial state
9. *NewEventMap* : Map from a class name to the state table entry which describes the receiving of a "new" event
10. *EventMap* : Map from an event name to a set of receiver names
11. *InheritsMap* : Map from a subclass name to a set of superclass names
12. *ObjValAttrTable* : Set containing tuples consisting of an object name and a class name where the object has been added to the *Class-* specification named by the class name as an object-valued attribute

6.2.6 Post Processing. Section 5.4 described how some transformations were required to be performed after all of the ULARCH traits and state transition tables were parsed and transformed. These additional transforms are performed by several functions. The top-level function is called *PostProcess.* It makes five different calls to functions to complete the ULARCH to O-SLANG transformations. *Update-Aggregates* is called to add nodes and arcs to *Aggregate-* specifications due to object communication, associations, or inheritance. *Add-ObjValAttributes* is invoked to add new object-valued attributes to *Class-* specifications as dictated by the data structure *ObjValAttrTable.* Next, *Add-NewEvents-and-CreateMethods* is called to build default "new" events and "create" methods for any *Class-* specifications in which they are not already declared. The remaining axioms needed to describe the behavior of the domain theory are built by the function *FinishAxioms.* Finally, the function *ReplaceInt* is called to replace any occurrences of the sort *Int* with *Integer*, since the latter is the class-sort of the *Integer* class specification which is built into O-SLANG.

6.3 Analysis of Implementation

Once the transformations were implemented, two tasks remained. First, the validation process described in Section 3.4.1 had to be applied to show that the individual transformations from ULARCH to O-SLANG were each semantically correct. Second, the issue of unique normalization, or completeness, needed to be addressed in order to show that the transformations as a whole produce semantically correct O-SLANG representations of object-oriented domain models. This section provides a summary of the results of the validation process followed by a discussion of how the term rewriting techniques introduced in Section 2.3 apply to the transformations.

6.3.1 Results of the Validation Process. In Section 3.4 two criteria were identified for validation: coverage and consistency. In order to determine if these criteria were met or not, checks needed to be applied at three different points in the transformation process. The first point was prior to input to the ULARCH parser. For both example domains, the traits and state transition tables were visually examined to make sure that the manual transformations from OMT were done correctly. Particular attention was paid to the portions relating to the modifications described in Section 4.3. This check validated that the input to the modified ULARCH parser was correct.

The next check point in the validation process was to check the output from the ULARCH parser. After changing the ULARCH grammar, it was compiled using *Dialect.* The compilation reported one reduce/reduce error which, upon inspection of the parse table, was determined to be the same reduce/reduce error reported by Lin (Lin94). A visual inspection of the ASTs produced by the modified parser, done using the graphical tool *In-*

spector, revealed that the changed ULARCH and state transition table parsers performed as desired. The parser was not adversely affected by any of the changes or the reduce/reduce error. Since the modified ULARCH parser is unambiguous, parsing a ULARCH file produces an abstract syntax tree that is unique to that file. This set the stage for the final checkpoint of the design, validating the output from the transformations.

To finish validating the OMT to O-SLANG transformation process, the O-SLANG domain theories produced for the bank and pump examples had to be examined. Each of the mappings defined in Chapter V was checked to ensure that the proper O-SLANG objects were created from ULARCH objects. Also, each object in the OMT domain model, along with its associated attributes, relationships, and operations, was checked to see that it was represented in the O-SLANG domain theory. The behavioral aspects of the OMT model, i.e. state and function, were also checked. All aspects of the original OMT domain model were covered, and no inconsistencies were found.

6.3.2 Rewrite System Properties. The final step in the analysis of the ULARCH to O-SLANG transformations was to determine if the transformations as a whole were semantically correct. To do this, term rewriting system properties were explored. In particular, if a term rewriting system is *uniquely normalizing*, then it is guaranteed to produce a unique output for each input. This is analogous to showing that a compiler is unambiguous. Recall from Section 2.3 the formal definition for a rewrite system:

... a pair $(\sum, R)$, where $\sum$ is an alphabet or signature and R is a set of rewrite rules. The syntax and vocabulary for a term rewriting system is (Klo92):

1. $\sum$ consists of a countably infinite set of variables $x_1, x_2, x_3, ...$ and a non-empty set $\sum_0$ of function symbols or operator symbols, each with an "arity", i.e. the number of arguments the function or operator is supposed to have.

2. The set of terms over $\sum$, $T(\sum)$ is defined inductively:
   (1) $x, y, z, ... \in T(\sum)$.
   (2) If $f \in \sum_0$ and $t_1, ..., t_n \in T(\sum)$ $(n \geq 0)$, then $f(t_1, ..., t_n) \in T(\sum)$.
3. Terms not containing a variable are ground terms.
4. A rewrite rule $\in R$ is a pair (l, r) of terms $\in T(\sum)$, written as $l \to r$. Rewrite rules can be named. (e.g. rewrite rule n is written as $r_n : l \to r$, and the application of r_n to some term α which produces some term β is written $\alpha \to_{r_n} \beta$).

In the context of this research, the transformations from ULARCH and state transition table ASTs to O-SLANG ASTs can be viewed as a rewrite system where:

1. $\sum$ is a set containing the objects in the ULARCH, state transition table, and O-SLANG grammars.
2. R is a set containing the REFINE rules defined in Section 6.2.3.
3. $\sum_0$ is a set containing all of the functions in the REFINE implementation of the transformations.
4. A term $t \in T$ is a ULARCH, state transition table, or O-SLANG object.

In order to determine if the ULARCH to O-SLANG transformations produce unique normal forms, it must be shown that the transformations have two properties, *termination* and *confluence*, as described in Section 2.3.4.3.

Termination involves showing that no infinite derivations of terms exist in the rewrite system. For the unified model to canonical model transformations, this required showing that termination was guaranteed for each of the three categories of mappings outlined in Chapter V. For ULARCH and state transition table transformations this amounted to demonstrating two things: objects are transformed only once, and no infinite loops of rule applications can occur. For the additional transformations which are performed as post processing, it must be shown that the functions are guaranteed to terminate.

The rules described in 6.2.3 do not produce ULARCH or state transition table objects. Since the preconditions for each of those rules only check for ULARCH or state transition

table objects and there is a finite number of objects in the ULARCH and state transition table ASTs, each object in the source AST will only be transformed once. When a REFINE rule is successfully applied to an object, the traversal restarts with the object to which the rule was successfully applied. As described in Section 4.2.3, this could result in an infinite loop of rule applications. By setting the boolean attribute of the ULARCH or state transition table object to false and checking the value of the attribute in the rule preconditions, the possibility of an infinite loop is eliminated. These two conditions ensure that no infinite derivations exist for O-SLANG objects.

As described in Section 6.2.2, the *enumerate* construct and the *DESCENDANTS-OF-CLASS* function in REFINE were used to traverse the ULARCH, state transition table, and O-SLANG ASTs to perform the additional processing transformations presented in Section 5.4. The *enumerates* construct builds sets of objects from an AST subtree. Since the ASTs are finite, the sets must be finite, and so the post processing functions must terminate.

Confluence says that for any two sequences of rewrites on a term, no matter how they diverge initially, their paths are guaranteed to rejoin at some common descendent term. This implies the impossibility of the existence of more than one normal form (DJ90). For the transformations from ULARCH to O-SLANG there is no possibility of divergence since each transformation rule uniquely maps ULARCH objects to O-SLANG objects. Since there is only one rewrite sequence for each ULARCH term, then there is only one possible O-SLANG form. Thus, the transformations can be said to be confluent.

6.4 Summary

Implementing the mappings presented in Chapter V proved to be a fairly straight forward task. Each mapping in Sections 5.2.1 through 5.2.7, as well as the mapping for state transition table entries, became a REFINE rule. The remaining mappings were used to guide the construction of O-SLANG objects. The overall control structure of the transformations was provided by the REFINE function *PREORDER-TRAVERSAL* which applies rules to subtrees of an object in a top-down fashion, moving from the root towards the leaves (Ref90). Because of the way the rules were defined and applied and the way objects were built, it was possible to show that the transformations were both terminating and confluent. This was important because it guarantees that the objects produced have exactly one normal form, and since the individual transformations were shown to be semantically correct, the overall transformation process will therefore produce semantically correct specifications.

VII. Conclusion and Recommendations

This chapter summarizes the accomplishments of this thesis effort, along with conclusions which can be drawn from the work. Finally, recommendations for the direction of future research are outlined.

7.1 Summary of Accomplishments

Recall from Chapter I that the overall objective of this research effort was to design a formalized object transformation process which produces canonical form algebraic models from object-oriented designs for use in design refinement. Specifically, the stated objective was:

> *Define a formal object transformation process by creating a canonical algebraic model to represent general object-oriented models and by using term rewriting techniques to develop transformations from* LARCH *specifications to the canonical model.*

To accomplish the objective, a literature review of theory-based object models and term rewriting systems was done, resulting in the knowledge base needed to identify the canonical model and to extend the formal object transformation process. This extension amounted to modifying Lin's ULARCH to Refine compiler so that it produces O-SLANG, the selected canonical model. The modification process was broken up into three phases:

1. Identifying a canonical algebraic framework
2. Defining a modified version of Lin's unified model
3. Implementing a set of transformations from the unified model to the canonical model

The transformations in phase three were implemented within the predefined constraints of coverage and consistency. Furthermore, they were shown to produce unique normal forms.

A solid foundation for transforming object-oriented domain models into a canonical model for use with design refinement was demonstrated. This supports the feasibility of a next generation application composition system.

7.2 Conclusions

This following conclusions can be drawn from this research:

1. The object-oriented algebraic specification language O-SLANG is a generalization of Rumbaugh's OMT. As can be seen in Figure 4.3, the structure of the unified model is heavily influenced by OMT. Each of the models in OMT has a corresponding theory object in the unified model. O-SLANG, on the other hand, is not dependent on any particular object-oriented methodology. As can be seen in Appendix A, O-SLANG seems to capture the essence of object-oriented designs.

2. O-SLANG can represent object-oriented constructs. This effort showed that object-oriented concepts as represented by Rumbaugh's OMT are completely captured in O-SLANG, but it did not demonstrate that the language captures object-oriented concepts in general. This issue is addressed in Section 7.3.

3. The evolutionary approach taken to extend the formal object transformation process facilitated the implementation of the transformations. Since this effort was broken up into three phases, each phase could build upon the validated product of the previous phase. This ensured that the updated version of the unified model could represent all of the information needed in the canonical model, and that the canonical model could represent all of the object-oriented concepts present in OMT.

4. The Software Refinery^{TM} environment is ideal for developing transformation systems. The DIALECT tool, along with OBJECT ISPECTOR, provides the means to create a formal language parser and to view the abstract syntax trees produced. Using Refine language constructs such as predefined abstract syntax tree traversal functions, the *enumerates* clause, and rules, allows the parser output to be transformed into the abstract syntax tree representation of an alternate language.

5. The ULARCH to O-SLANG compiler completed in this effort successfully parses ULARCH traits and state transition tables and produces an O-SLANG domain theory which represents the initial OMT domain model. Appendix G contains the user's manual for the compiler.

6. Performing design refinement on algebraic specifications produced from object-oriented models is feasible. This effort showed that O-SLANG can be produced from OMT domain models. SpecWare, which is Kestrel's design refinement tool, uses the specification langauage SLANG. The O-SLANG domain theories produced by the ULARCH to O-SLANG compiler can be captured in SLANG, since O-SLANG was designed as an extension to SLANG. As will be discussed in Section 7.3, the only thing needed is to implement the transformations from O-SLANG to SLANG.

7. Term rewriting techniques can be applied to transformation systems. In this effort, the Refine function *PREORDER-TRAVERSAL* was used to traverse the tree, and the mappings from ULARCH to O-SLANG were applied as Refine rules to each object in the tree. Defining the mappings as rules and functions, along with using the tree representation, allowed for the transformations to be viewed as a rewrite system and

made it possible for the properties of *termination* and *confluence* to be applied to the transformations. Showing that these two properties hold guarantees that the transformations produce unique forms.

7.3 Recommendations for Future Research

This section outlines some issues that should be addressed in future research efforts. Those issues are:

1. *Extend the UZed Portion of the Formal Object Transformation Process* - This research focused on extending the Larch portion of Lin and Wabiszewski's formal object transformation process. Because of time constraints, the transformations from OMT to UZed to O-SLANG were not addressed. Implementation of this portion of the transformation process should be completed to show that O-SLANG does unify the LARCH-based and Z-based approaches to writing formal specifications.

2. *Define Transformations from O-SLANG to SLANG* - One of the main goals of a next generation composition system is to produce executable code from object-oriented domain models; this can be done through design refinement. To make code production a reality, the transformations from O-SLANG to SLANG should be defined and implemented. This would allow OMT models to be transformed into a form that is used by SpecWare.

3. *Combine ULARCH and State Transition Table Parsers* - Currently, the ULARCH domain model and grammar is separate from those of the state transition table. To simplify the transformation process, the domain models and grammars should be

merged to produce a single grammar and domain model, and thus a single parser. Complete domain models could then be captured in single ULARCH files. Using one parser would mean that transformations would be applied to a single AST versus multiple ASTs.

4. *Eliminate the Unified Model* - In future versions of the formal object transformation process, it may be possible to eliminate the unified model. Using the surface syntax of Larch, Z, and the state transition tables, semantic processing routines can be triggered to build equivalent O-SLANG domain theories. In essence, Larch traits, Z schemas, and the associated state transition tables could be parsed directly into O-SLANG ASTs. This is shown in Figure 7.1, where the unified AST and the UZed and ULARCH transformations to O-SLANG have been removed.

5. *Incorporate Alternate Object-oriented Methodologies* - The current formal object transformation process is based solely on Rumbaugh's OMT. If O-SLANG is indeed a canonical formal model for capturing object-oriented designs, then it should be possible to develop transformations which produce O-SLANG domain theories from other object-oriented methodologies.

6. *Develop Theorem Prover Interface* - Lin and Wabiszewski pointed out that during specification refinement it is necessary to be able to demonstrate consistency and completeness (Lin94, Wab94). The use of an automated theorem prover facilitates this task. Rather than develop theorem prover interfaces for multiple specification languages, a single interface should be developed for use with O-SLANG.

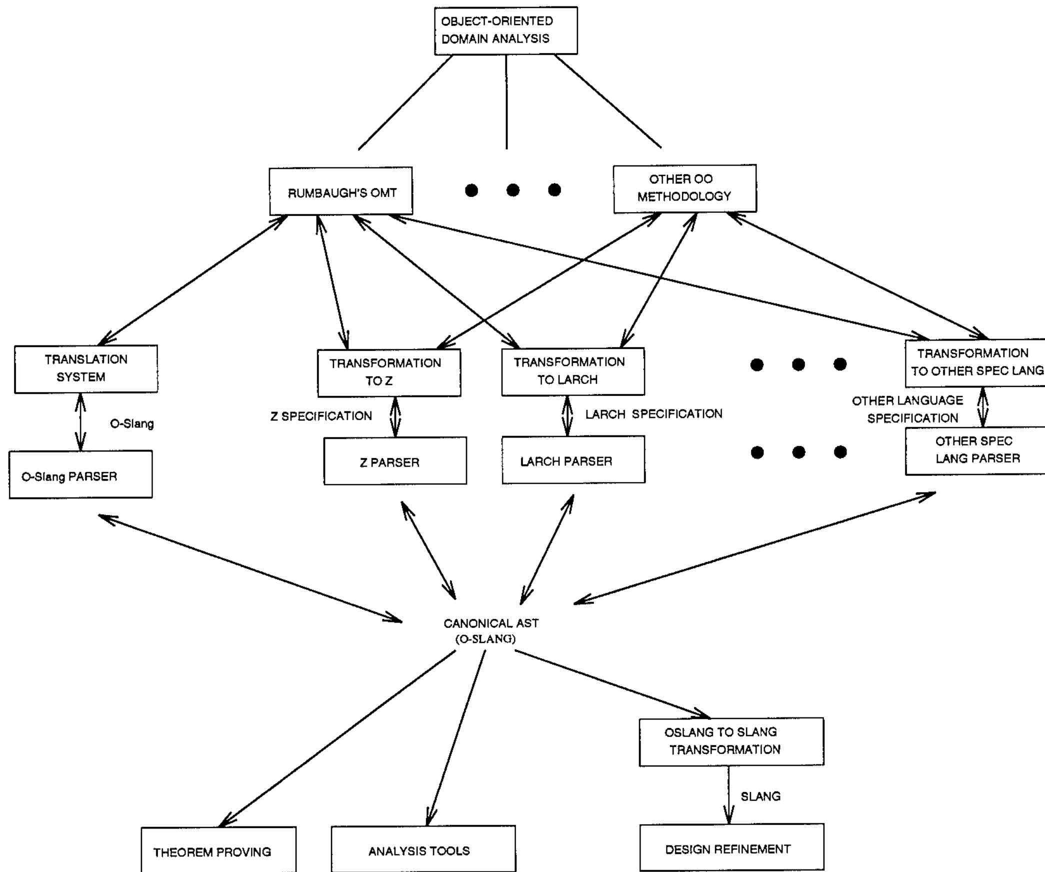

Figure 7.1 Projected Transformation Process

Implementation of the recommendations would result in a "flatter" system which does not contain the unified AST to canonical AST transformation step. The system would also be able to handle the transformation of domain models captured using other object-oriented methodologies and other formal specification languages. Finally, the system would be fully bidirectional; object-oriented domain models could be built from O-SLANG domain theories. This projected transformation process is reflected in Figure 7.1.

7.4 Final Comments

Perhaps the most critical point in the software development process is the transition from a user-provided system description to a software system specification. Object-oriented techniques facilitate the transition by providing a more naturally understandable representation of the system through the use of diagrams. Formal methods and formal specification languages enhance the transition by providing a means to reason about the object-oriented model in terms of completeness and consistency, thus ensuring that the system specification is correct before the development process continues. The extended formal object transformation process developed during this effort provides the basis for being able to check the correctness of specifications. Furthermore, this research demonstrates the feasibility of transforming object-oriented domain models into a form suitable for design refinement. This is a significant step towards the capability to produce executable code from object-oriented models.

Appendix A. O-SLANG Domain Model

This appendix contains the object model diagram for the language O-SLANG. The creation of O-SLANG abstract syntax trees is based on the syntax of the language as well as this domain model.

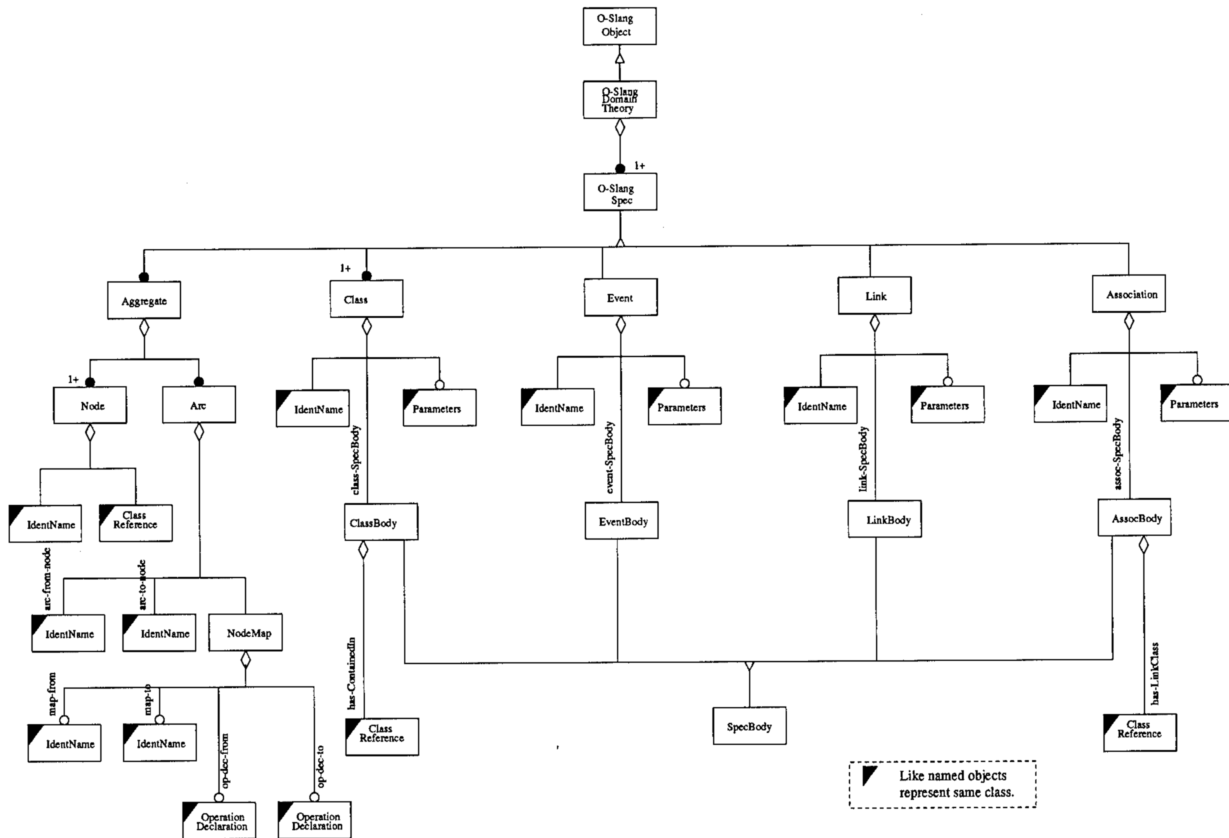

Figure A.1 Top Level of O-SLANG Domain Model

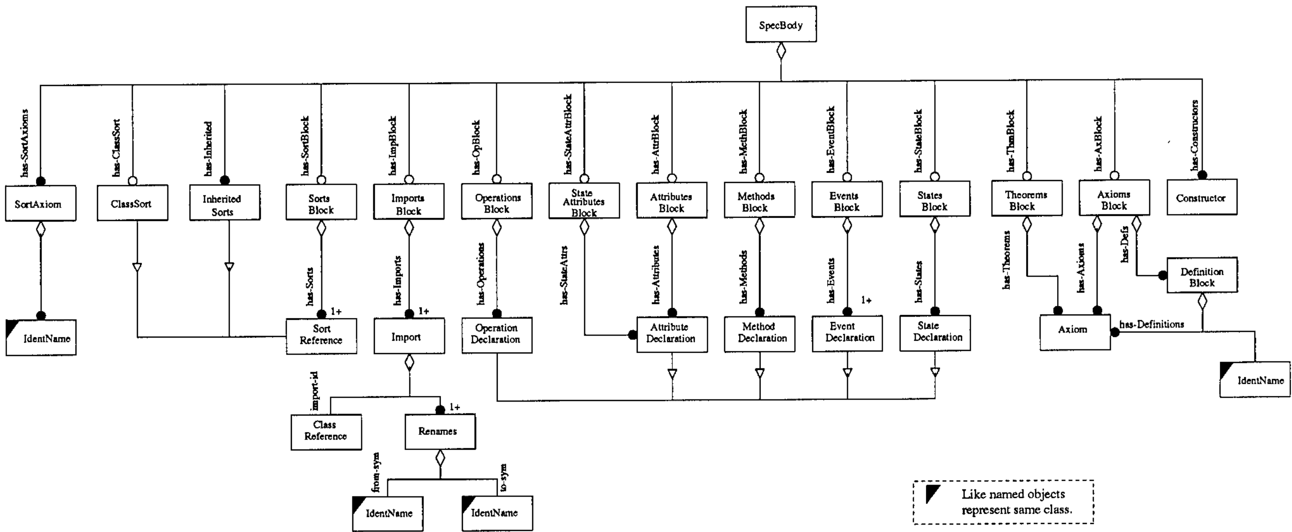

Figure A.2 Second Level of O-SLANG Domain Model

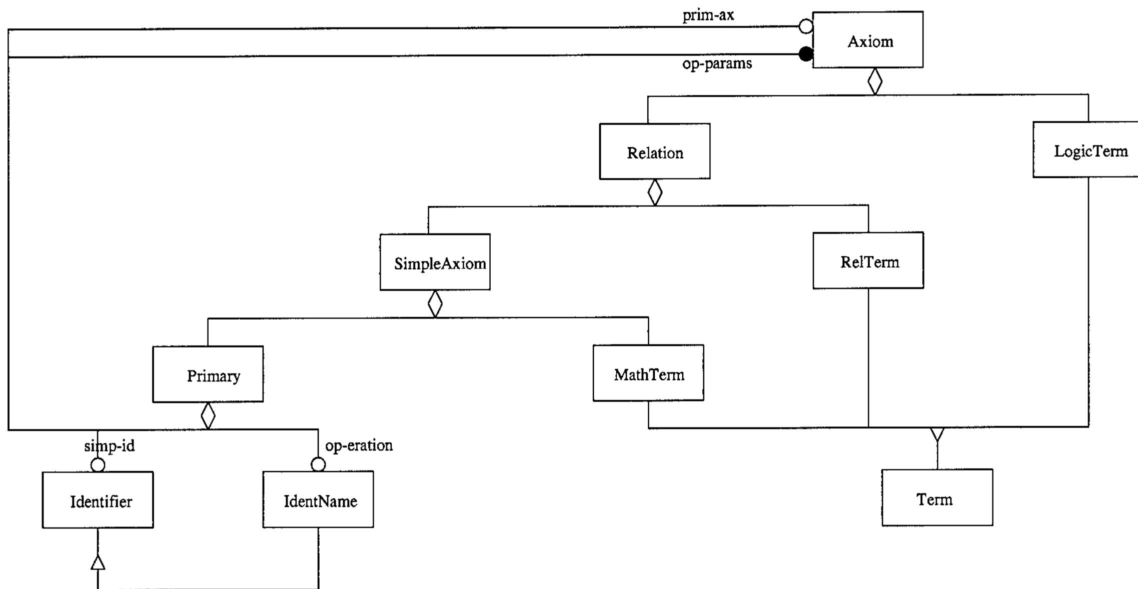

Figure A.3 Domain Model for Axioms

This appendix contains the steps described in the rewrite example presented in Section 2.3.5. Each tree represents a step in the rewriting process. In Figure B.1 there are five possible terms that can be rewritten. Figure B.2 shows the tree that results from applying choice 2. Figure B.6 shows the final tree that results from applying all rewrites.

e.g. (X IN UNION(U, V)) <=> ((X IN U) | (X IN V))

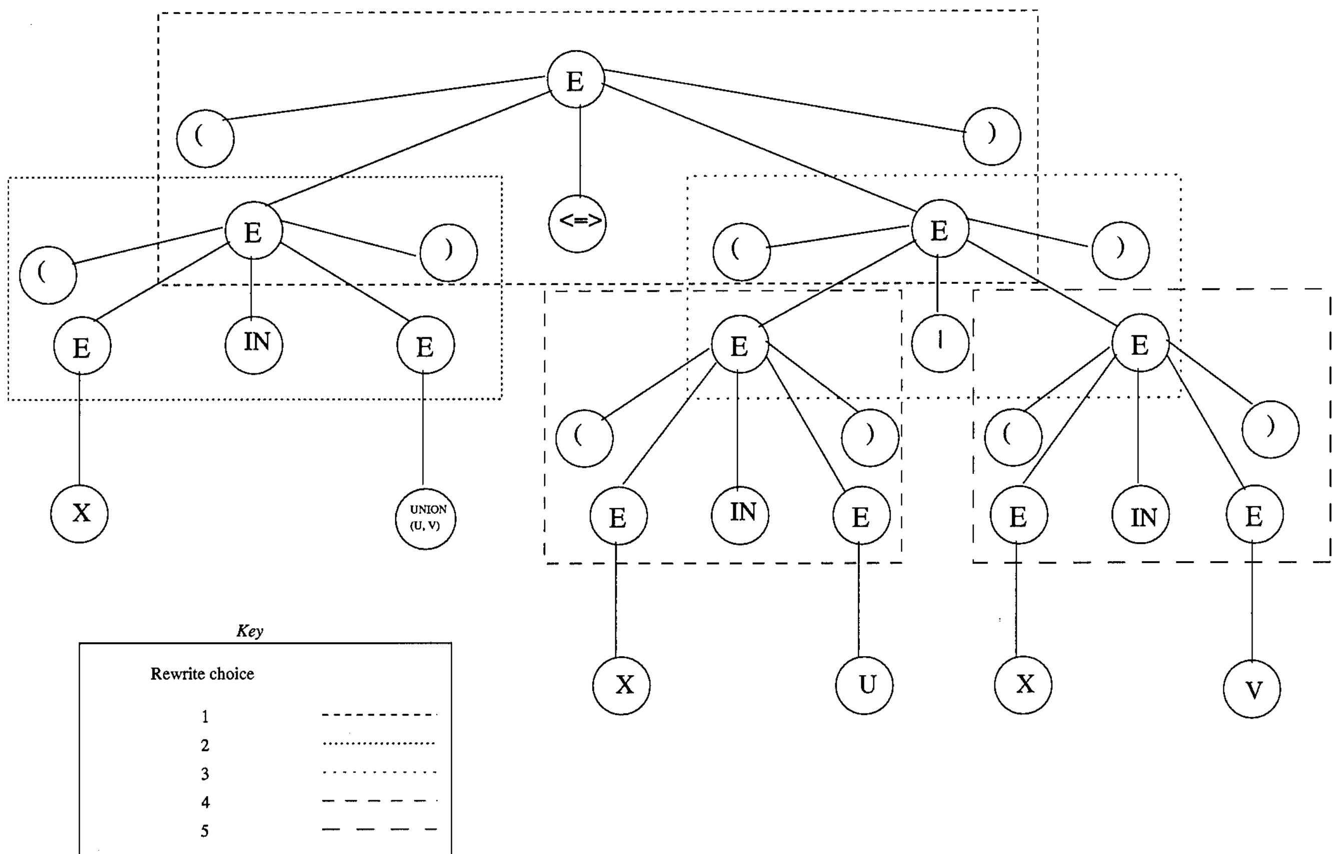

Figure B.1 Tree Rewrite Example (Step 1)

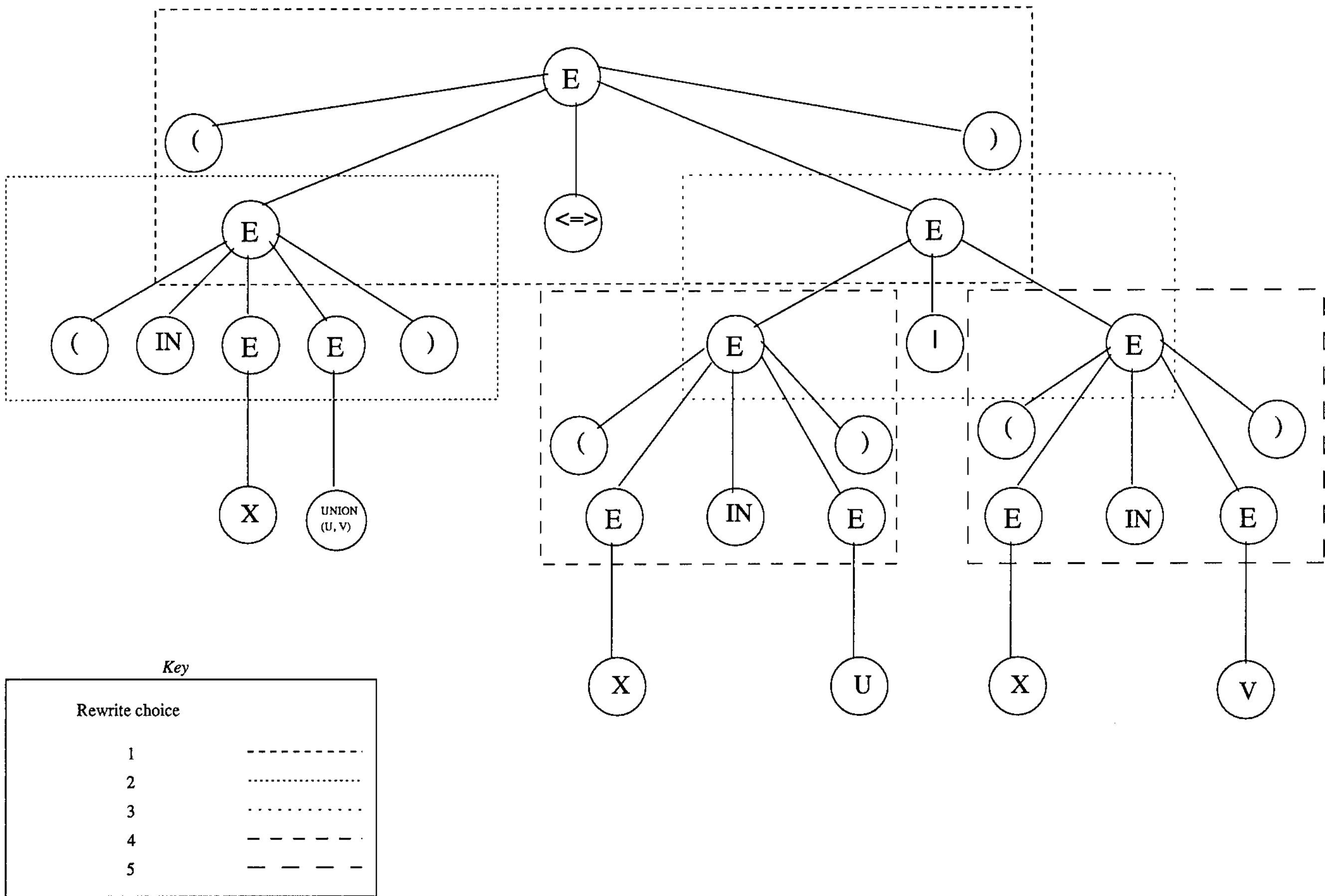

Figure B.2 Tree Rewrite Example (Step 2)

e.g. (X IN UNION(U, V)) <=> ((X IN U) | (X IN V))

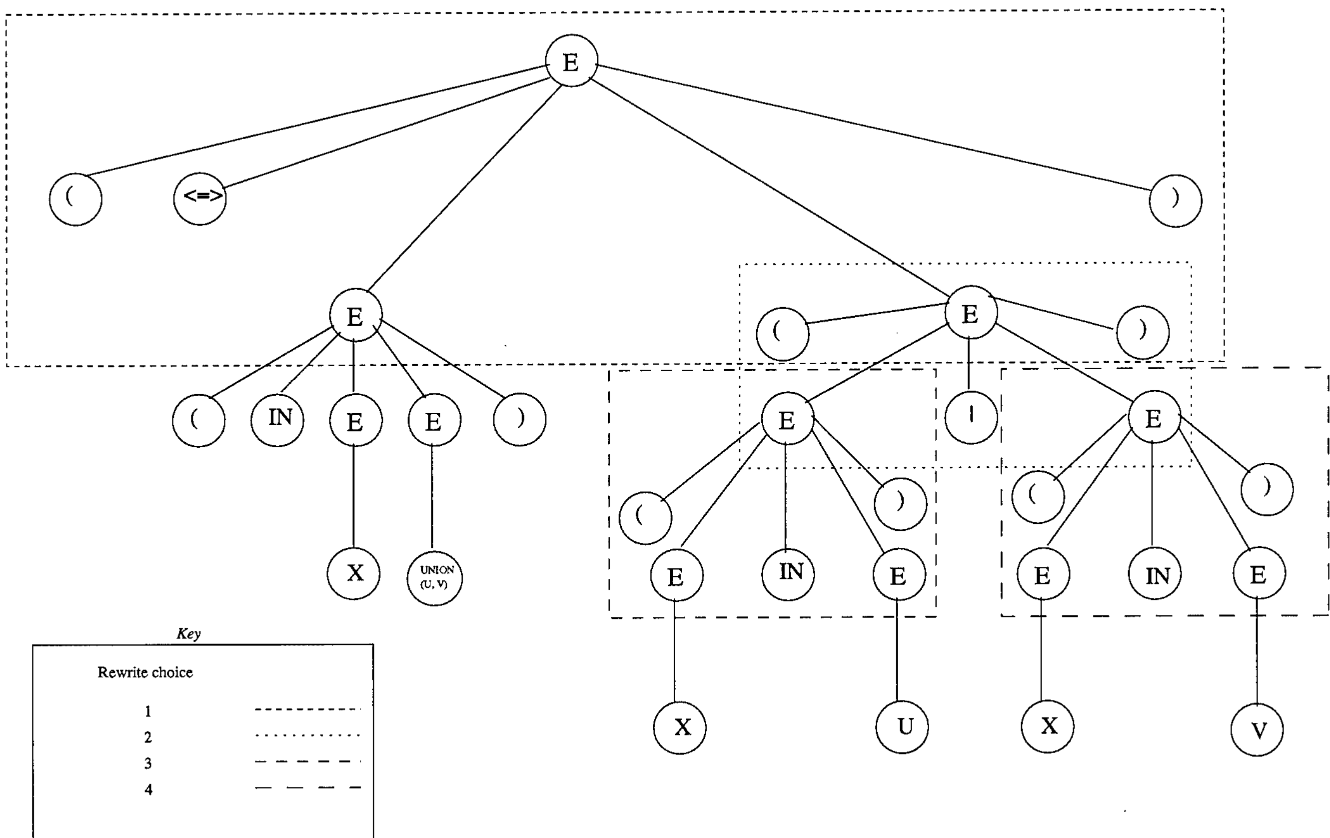

Figure B.3 Tree Rewrite Example (Step 3)

e.g. (X IN UNION(U, V)) <=> ((X IN U) | (X IN V))

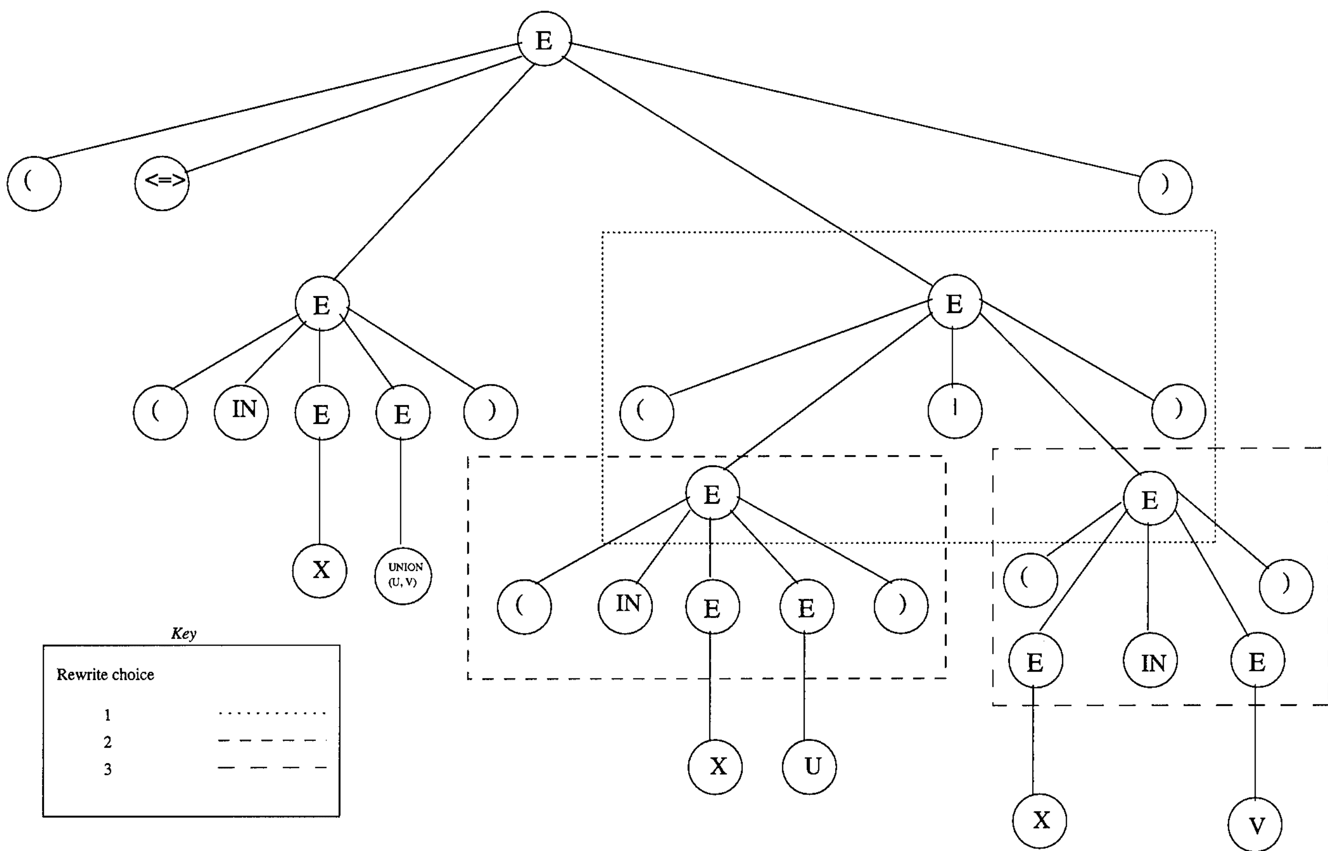

Figure B.4 Tree Rewrite Example (Step 4)

e.g. (X IN UNION(U, V)) <=> ((X IN U) | (X IN V))

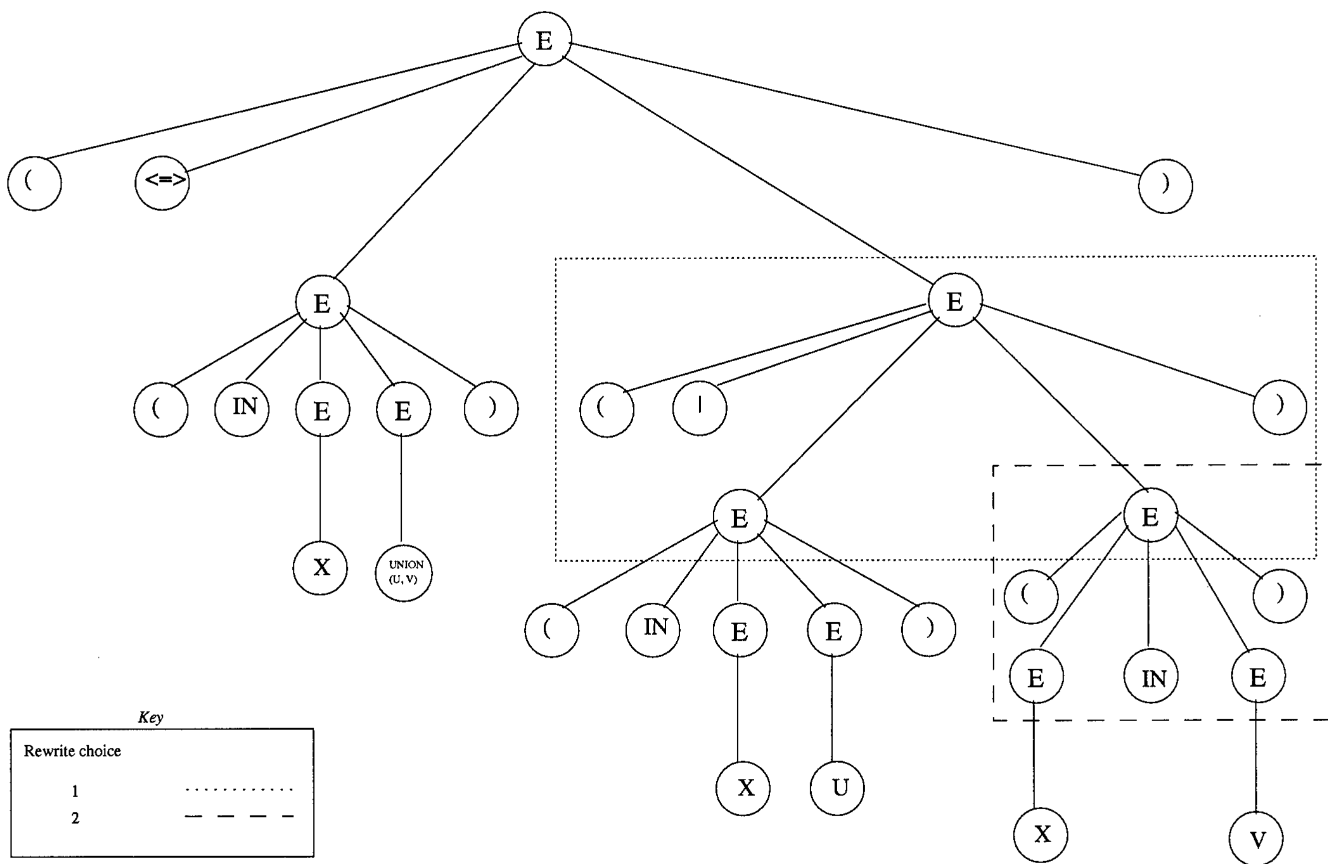

Figure B.5 Tree Rewrite Example (Step 5)

e.g. (X IN UNION(U, V)) <=> ((X IN U) | (X IN V))

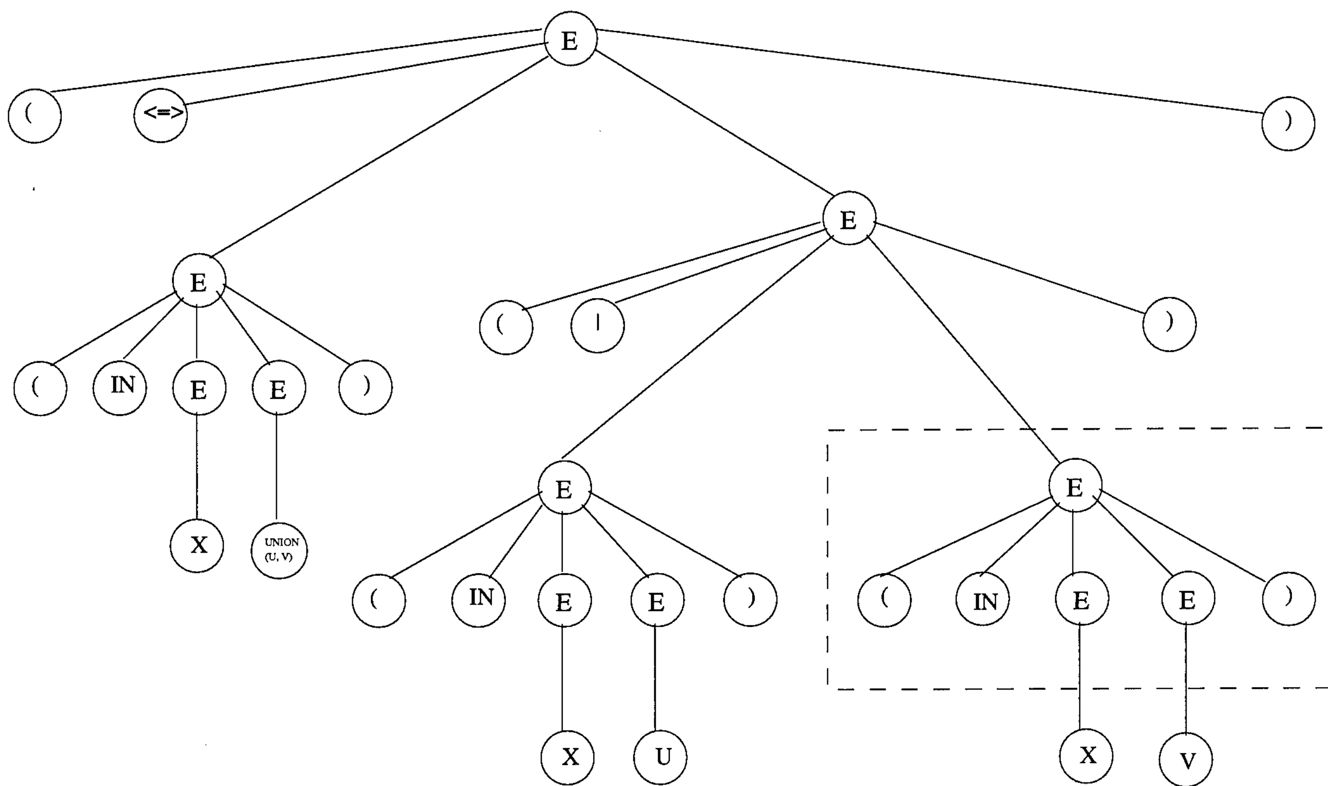

Figure B.6 Tree Rewrite Example (Step 6)

Appendix C. ULARCH for Bank Domain Example

This appendix contains the ULARCH traits and state transition tables for the *Bank* example. The traits are based on the *Bank* object model described in Section 3.4.2.

```
\documentstyle[fullpage,larch]{article}
\begin{document}

\begin{spec} %ObjectTheory
Date(D): trait
  includes String
\end{spec}\\

\begin{spec} %FunctionalTheory
Current-Date: trait
  includes Date
  introduces current-date: D -> String
\end{spec}\\

\begin{spec} %ObjectTheory
Account(Acct): trait
  includes Date, Integer
  introduces
    int-date: Acct -> D
    balance: Acct -> Amnt
    acct-num: Acct -> Int
\end{spec}\\

\begin{spec} %StateTheory
OK: trait
  includes Account
  introduces OKState: Acct -> Bool
  asserts \forall a:Acct
    balance(a) >= 0
\end{spec}\\

\begin{spec} %StateTheory
Overdrawn: trait
  includes Account
  introduces OverdrawnState: Acct -> Bool
  asserts \forall a:Acct
    balance(a) < 0
\end{spec}\\

\begin{spec} %EventTheory
NewAccount: trait
  includes Account
```

```
  introduces new-account: -> Bool
  asserts \forall a:Acct
    balance(new-account) = 0;
    acct-num(new-account) = 0
\end{spec}\\

\begin{spec} %EventTheory
Credit: trait
  includes Account
  introduces credit: Amnt -> Bool
\end{spec}\\

\begin{spec} %EventTheory
Debit: trait
  includes Account
  introduces debit: Amnt -> Bool
\end{spec}\\

\begin{spec} %EventTheory
Close: trait
  includes Account
  introduces close: -> Bool
\end{spec}\\

\begin{spec} %FunctionalTheory
Credit-Acct: trait
  includes Account
  introduces credit-acct: Acct, Amnt -> Acct
  asserts \forall ac: Acct, am: Amnt
    balance(credit-acct(ac, am)) = (balance(ac) + am)
\end{spec}\\

\begin{spec} %FunctionalTheory
Debit-Acct: trait
  includes Account
  introduces debit-acct: Acct, Amnt -> Acct
  asserts \forall ac: Acct, am: Amnt
    balance(debit-acct(ac, am)) = (balance(ac) - am)
\end{spec}\\

\begin{spec} %FunctionalTheory
Close-Acct: trait
  includes Account
  introduces close-acct: Acct -> Acct
\end{spec}\\

\begin{spec} %ObjectTheory
Archive(Arch): trait
  includes Date
\end{spec}\\
```

```
\begin{spec} %EventTheory
ArchCredit: trait
  includes Archive
  introduces archcredit: Acct, Amnt, D -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ArchDebit: trait
  includes Archive
  introduces archdebit: Acct, Amnt, D -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ArchRate: trait
  includes Archive
  introduces archdebit: Acct, Rate, D -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ArchClose: trait
  includes Archive
  introduces archclose: Acct, D -> Bool
\end{spec}\\

\begin{spec} %FunctionalTheory
Arch-Credit: trait
  includes Archive, Account, D
  introduces arch-credit: Arch, Acct, Amnt, D -> Arch
\end{spec}\\

\begin{spec} %FunctionalTheory
Arch-Debit: trait
  includes Archive, Account, D
  introduces arch-debit: Arch, Acct, Amnt, D -> Arch
\end{spec}\\

\begin{spec} %FunctionalTheory
Arch-Rate: trait
  includes Archive, Account, D
  introduces arch-rate: Arch, Acct, Rate, D -> Arch
\end{spec}\\

\begin{spec} %FunctionalTheory
Arch-Close: trait
  includes Archive, Account, D
  introduces arch-close: Arch, Acct, D -> Arch
\end{spec}\\

\begin{spec} %ObjectTheory
Person(P): trait
  includes String
```

```
  introduces
    name: P -> String
    address: P -> String
\end{spec}\\

\begin{spec} %ObjectTheory
Customer(Cust): trait
  includes Person(P for P)
\end{spec}\\

\begin{spec} %ObjectTheory
Employee(Empl): trait
  includes Person(P for P), Integer, String
  introduces number: Empl -> int
             passwd: Empl -> String
\end{spec}\\

\begin{spec} %ObjectTheory
Cust-Employee(Cust-Empl): trait
  includes Customer(Cust for Cust), Employee(Empl for Empl)
\end{spec}\\

\begin{spec} %ObjectTheory
Teller(Tell): trait
  includes Employee(Empl for Empl)
\end{spec}\\

\begin{spec} %ObjectTheory
Executive(Exec): trait
  includes Employee(Empl for Empl)
\end{spec}\\

\begin{spec} %ObjectTheory
Checking(CAcct): trait
  includes Account(CAcct for Acct)
\end{spec}\\

\begin{spec} %EventTheory
WriteCheck: trait
  includes Checking
  introduces writecheck: Amnt -> Bool
\end{spec}\\

\begin{spec} %FunctionalTheory
Write-Check: trait
  includes Checking
  introduces write-check: CAcct, Amnt -> CAcct
  asserts \forall c: CAcct, a: Amnt
    balance(write-check(c, a)) = balance(debit-acct(c, a));
    int-date(write-check(c, a)) = int-date(c);
    acct-num(write-check(c, a)) = acct-num(c)
```

```
\end{spec}\\

\begin{spec} %ObjectTheory
Savings(SAcct): trait
  includes Account(SAcct for Acct), Date
  introduces rate: SAcct -> Rate
             date-int-computed: SAcct -> D
\end{spec}\\

\begin{spec} %EventTheory
SetRate: trait
  includes Savings
  introduces setrate: Rate -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ComputeInterest: trait
  includes Savings
  introduces computeinterest: -> Bool
\end{spec}\\

\begin{spec} %FunctionalTheory
Set-Rate: trait
  includes Savings
  introduces set-rate: SAcct, Rate -> SAcct
  asserts \forall sa: SAcct, r: Rate
    rate(set-rate(sa, r)) = r;
    date-int-computed(set-rate(sa, r)) = current-date;
    int-date(set-rate(sa, r)) = int-date(sa);
    balance(set-rate(sa, r)) = balance(sa);
    Acct-num(set-rate(sa, r)) = Acct-num(sa)
\end{spec}\\

\begin{spec} %FunctionalTheory
Compute-Interest: trait
  includes Savings
  introduces compute-interest: SAcct -> Amnt
  asserts \forall sa: SAcct
    compute-interest(sa) = (balance(sa) * rate(sa));
    rate(set-rate(sa, r)) = rate(sa);
    date-int-computed(compute-interest(sa, r)) = date-int-computed(sa);
    int-date(compute-interest(sa, r)) = int-date(sa);
    balance(compute-interest(sa, r)) = balance(sa);
    Acct-num(compute-interest(sa, r)) = Acct-num(sa)
\end{spec}\\

\begin{spec} %ObjectTheory
Combined: trait
  includes Checking(CAcct for CAcct), Savings(SAcct for SAcct)
\end{spec}\\
```

```
\begin{spec} %ObjectTheory
Console(Cons): trait
  includes Integer
  introduces id: Cons -> int
\end{spec}\\

\begin{spec} %StateTheory
LoggedIn: trait
  includes Console
  introduces LoggedInState: -> Bool
\end{spec}\\

\begin{spec} %StateTheory
Disabled: trait
  includes Console
  introduces DisabledState: -> Bool
\end{spec}\\

\begin{spec} %StateTheory
Enabled: trait
  includes Console
  introduces EnabledState: -> Bool
\end{spec}\\

\begin{spec} %StateTheory
Executive: trait
  includes Console
  introduces ExecutiveState: -> Bool
\end{spec}\\

\begin{spec} %EventTheory
NewConsole: trait
  includes Console
  introduces new-console: -> Bool
  asserts \forall c: Cons
    id(new-console) = 0
\end{spec}\\

\begin{spec} %EventTheory
Login: trait
  includes Console
  introduces login: -> Bool
\end{spec}\\

\begin{spec} %EventTheory
Logout: trait
  includes Console
  introduces logout: -> Bool
\end{spec}\\

\begin{spec} %EventTheory
```

```
ExecLogin: trait
  includes Console
  introduces execlogin: -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ChangeRate: trait
  includes Console
  introduces changerate: Rate -> Bool
\end{spec}\\

\begin{spec} %EventTheory
UpdateAccts: trait
  includes Console
  introduces updateaccts: -> Bool
\end{spec}\\

\begin{spec} %EventTheory
SelectAcct: trait
  includes Console
  introduces selectacct: Acct -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ShowBalance: trait
  includes Console
  introduces showbalance: Acct -> Bool
\end{spec}\\

\begin{spec} %EventTheory
CreditAcct: trait
  includes Console
  introduces creditacct: Acct, Amnt -> Bool
\end{spec}\\

\begin{spec} %EventTheory
DebitAcct: trait
  includes Console
  introduces debitacct: Acct, Amnt -> Bool
\end{spec}\\

\begin{spec} %EventTheory
CloseAcct: trait
  includes Console
  introduces closeacct: Acct -> Bool
\end{spec}\\

\begin{spec} %EventTheory
CashCheck: trait
  includes Console
  introduces cashcheck: Acct, Amnt -> Bool
```

```
\end{spec}\\

\begin{spec} %LinkTheory
Op: trait
  includes Employee, Console
  introduces an-employee: Op-Link -> Employee
             a-console: Op-Link -> Console
             new-Op-Link: Employee, Console -> Op-Link
  asserts \forall e: Empl, c: Cons
    an-employee(new-Op-Link(e, c)) = e;
    a-console(new-Op-Link(e, c)) = c
\end{spec}\\

\begin{spec} %AssociationTheory
Operates: trait
  includes Set(Operates for C, Op for E), Op
  introduces
    new-Operates: -> Opers
    image: Opers, Empl -> Consoles
    image: Opers, Cons -> Employees
    does-operate: Opers, Empl, Cons -> Bool
  asserts \forall o: Opers, e: Empl, c: Cons, x:Op-Link
    (in(x, o) \and (a-console(x) = c)) == in(an-employee(x), image(o, c));
    (in(x, o) \and (an-employee(x) = e)) == in(a-console(x), image(o, e));
    Size(image(o, e)) = 1;
    Size(image(o, c)) >= 0;
    new-Operates = empty-set;
    does-operate(new-Operates, e, c) = false;
    does-operate(o, e, c) == (in(e, image(o, c)) \and
                               in(c, image(o, e)))
\end{spec}\\

\begin{spec} %LinkTheory
Own: trait
  includes Customer, Account
  introduces a-customer: Own-Link -> Cust
             an-account: Own-Link -> Acct
             new-Own-Link: Cust, Acct -> Own-Link
  asserts \forall c: Cust, a: Acct
    a-customer(new-Own-Link(c, a)) = c;
    an-account(new-Own-Link(c, a)) = a
\end{spec}\\

\begin{spec} %AssociationTheory
Owns: trait
  includes Set(Owns for C, Own for E), Own
  introduces
    new-Owns: O, Cust, Acct -> O
    image: O, Cust -> Accounts
    image: O, Acct -> Customers
    does-own: O, Cust, Acct -> Bool
```

```
  asserts \forall o: O, c: Cust, a: Acct, x: Own-Link
    Size(image(o, c)) >= 0;
    Size(image(o, a)) = 1;
    (in(x, o) \and (a-customer(x) = c)) == in(an-account(x), image(o, c));
    (in(x, o) \and (an-account(x) = a)) == in(a-customer(x), image(o, a));
    new-Owns = empty-set;
    does-own(new-Owns, c, a) = false;
    does-own(o, c, a) == (in(c, image(o, a)) \and
                          in(a, image(o, c)))
\end{spec}\\

\begin{spec} %LinkTheory
Manipulate: trait
  includes Console, Account
  introduces a-console: Manipulate-Link -> Cons
             an-account: Manipulate-Link -> Acct
             new-Manipulate-Link: Cons, Acct -> Manipulate-Link
  asserts \forall c: Cons, a: Acct
    a-console(new-Manipulate-Link(c, a)) = c;
    an-account(new-Manipulate-Link(c, a)) = a
\end{spec}\\

\begin{spec} %AssociationTheory
Manipulates: trait
  includes Set(Manipulates for C, Manipulate for E), Manipulate
  introduces
    new-Manipulates: Manips, Cons, Acct -> Manips
    image: Manips, Cons -> Accounts
    image: Manips, Acct -> Consoles
    does-manipulate: Manips, Cons, Acct -> Bool
  asserts \forall m: Manips, c: Cons, a: Acct, x: Manipulate-Link
    Size(image(m, c)) >= 0;
    Size(image(m, a)) >= 0;
    (in(x, m) \and (a-console(x) = c)) == in(an-account(x), image(m, c));
    (in(x, m) \and (an-account(x) = a)) == in(a-console(x), image(m, a));
    new-Manipulates = empty-set;
    does-manipulate(new-Manipulates, c, a) = false;
    does-manipulate(m, c, a) == (in(c, image(m, a)) \and
                                 in(a, image(m, c)))
\end{spec}\\

\begin{spec} %LinkTheory
Ar: trait
  includes Account, Archive
  introduces an-account: Ar-Link -> Acct
             an-archive: Ar-Link -> Arch
             new-Ar-Link: Acct, Arch -> Ar-Link
  asserts \forall ac: Acct, ar: Arch
    an-account(new-Ar-Link(ac, ar)) = ac;
    an-archive(new-Ar-Link(ac, ar)) = ar
\end{spec}\\
```

```
\begin{spec} %AssociationTheory
Archives: trait
  includes Set(Archives for C, Ar for E), Ar
  introduces
    new-Archives: Archs, Acct, Arch -> Archs
    image: Archs, Acct -> ArchiveSet
    image: Archs, Arch -> Accounts
    does-archive: Archs, Acct, Arch -> Bool
  asserts \forall ars: Archs, ac: Acct, ar: Arch, x: Ar-Link
    Size(image(ars, ac)) = 1;
    Size(image(ars, ar)) >= 0;
    (in(x, ars) \and (an-account(x) = ac)) == in(an-archive(x), image(ars, ac));
    (in(x, ars) \and (an-archive(x) = ar)) == in(an-account(x), image(ars, ar));
    new-Archives = empty-set;
    does-archive(new-Archives, ac, ar) = false;
    does-archive(ars, ac, ar) == (in(ac, image(ars, ar)) \and
                                  in(ar, image(ars, ac)))
\end{spec}\\

\begin{spec} %LinkTheory
Access: trait
  includes Account, Date
  introduces an-account: Access-Link -> Acct
             a-date: Access-Link -> Acct
             new-Access-Link: Acct, D -> Access-Link
  asserts \forall a: Acct, d: D
    an-account(new-Access-Link(a, d)) = a;
    a-date(new-Access-Link(a, d)) = d
\end{spec}\\

\begin{spec} %AssociationTheory
Accesses: trait
  includes Set(Accesses for C,Access  for E), Access
  introduces
    new-Accesses: Accs, Acct, D -> Accs
    image: Accs, Acct -> Dates
    image: Accs, D -> Accounts
    does-access: Accs, Acct, D -> Bool
  asserts \forall acs: Accs, ac: Acct, d: D, x: Access-Link
    Size(image(acs, ac)) = 1;
    Size(image(acs, d)) >= 0;
    (in(x, acs) \and (an-account(x) = ac)) == in(a-date(x), image(acs, ac));
    (in(x, acs) \and (a-date(x) = d)) == in(an-account(x), image(acs, d));
    new-Accesses = empty-set;
    does-access(new-Accesses, ac, d) = false;
    does-access(acs, ac, d) == (in(ac, image(acs, d)) \and
                                in(d, image(acs, ac)))
\end{spec}\\

\begin{spec} %ObjectTheory
```

```
Bank: trait
  includes
    Set(PersonSet for C, Person for E),
    Set(CustomerSet for C, Customer for E),
    Set(EmployeeSet for C, Employee for E),
    Set(Cust-EmployeeSet for C, Cust-Employee for E),
    Set(TellerSet for C, Teller for E),
    Set(ExecutiveSet for C, Executive for E),
    Set(ConsoleSet for C, Console for E),
    Set(AccountSet for C, Account for E),
    Set(CheckingSet for C, Checking for E),
    Set(SavingsSet for C, Savings for E),
    Set(CombinedSet for C, Combined for E),
    Set(ArchiveSet for C, Archive for E),
    Owns
    B tuple of    P: PersonSet,
                 CU: CustomerSet,
                 EM: EmployeeSet,
                 CE: Cust-EmployeeSet,
                  T: TellerSet,
                  E: ExecutiveSet,
                 CS: ConsoleSet,
                ACS: AccountSet,
                 CK: CheckingSet,
                 SV: SavingsSet,
                  C: CombinedSet,
                ARS: ArchiveSet,
          owns-obj: O
  asserts \forall b:B
    Size(P(b)) >= 0;
    Size(CU(b)) >= 0;
    Size(EM(b)) >= 0;
    Size(CE(b)) >= 0;
    Size(T(b)) >= 0;
    Size(E(b)) >= 0;
    Size(CS(b)) >= 0;
    Size(ACS(b)) >= 0;
    Size(CK(b)) >= 0;
    Size(SV(b)) >= 0;
    Size(C(b)) >= 0;
    Size(ARS(b)) >= 0;
    P(b) = Union(CU(b), EM(b));
    EM(b) = Union(T(b), E(b));
    Subset(CE(b), CU(b));
    Subset(CE(b), EM(b));
    ACS(b) = Union(CK(b), SV(b));
    Subset(C(b), CK(b));
    Subset(C(b), SV(b))
\end{spec}

\end{document}
```

Table C.1 Account State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
AccountInitialState	NewAccount			OK		
OK	Credit	acct amnt		OK	credit-acct	ArchCredit(Archive-obj, acct, amnt, date)
OK	Debit	acct amnt	$amnt > balance$	Overdrawn		ArchDebit(Archive-obj, acct, amnt, date)
OK	Debit	acct amnt	$amnt <= balance$	OK	debit-acct	ArchDebit(Archive-obj, acct, amnt, date)
OK	Close	acct		AccountEndState	close-acct	ArchClose(Archive-obj, acct, date)
OverDrawn	Debit	acct amnt		OverDrawn		
OverDrawn	Close	acct		OverDrawn		
Overdrawn	Credit	acct amnt	$amnt + balance >= 0$	OK	credit-acct	ArchCredit(Archive-obj, acct, amnt, date)
Overdrawn	Credit	acct amnt	$amnt + balance < 0$	Overdrawn	credit-acct	ArchCredit(Archive-obj, acct, amnt, date)

Table C.2 Checking State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
CheckingInitialState	NewChecking			OK		
OK	WriteCheck	cacct amnt	$amnt > balance$	Overdrawn		ArchDebit(Archive-obj, acct, amnt, date)
OK	WriteCheck	cacct amnt	$amnt <= balance$	OK	write-check	ArchDebit(Archive-obj, acct, amnt, date)
OverDrawn	WriteCheck	cacct amnt		OverDrawn		

Table C.3 Savings State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
SavingsInitialState	NewSavings			OK		
OK	SetRate	sacct rate		OK	set-rate	
OK	ComputeInterest	sacct		OK	compute-interest	
OverDrawn	SetRate	sacct amnt		OverDrawn	set-rate	
OverDrawn	ComputeInterest	sacct		OverDrawn	compute-interest	

Table C.4 Console State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
ConsoleInitialState	NewConsole			Disabled		
Disabled	Login			LoggedIn		
Disabled	ExecLogin			Executive		
Disabled	Logout			Disabled		
Disabled	SelectAcct	acct		Disabled		
Disabled	ShowBalance	acct		Disabled		
Disabled	CreditAcct	acct amnt		Disabled		
Disabled	DebitAcct	acct amnt		Disabled		
Disabled	CloseAcct	acct		Disabled		
Disabled	CashCheck	acct amnt		Disabled		
Disabled	ChangeRate	rate		Disabled		
Disabled	UpdateAccts			Disabled		
LoggedIn	Logout			Disabled		
LoggedIn	SelectAcct	acct		Enabled		
LoggedIn	Login			LoggedIn		
LoggedIn	ExecLogin			LoggedIn		
LoggedIn	ShowBalance	acct		LoggedIn		
LoggedIn	CreditAcct	acct amnt		LoggedIn		
LoggedIn	DebitAcct	acct amnt		LoggedIn		
LoggedIn	CloseAcct	acct		LoggedIn		
LoggedIn	CashCheck	acct amnt		LoggedIn		
LoggedIn	ChangeRate	rate		LoggedIn		
LoggedIn	UpdateAccts			LoggedIn		
Enabled	ShowBalance	acct		Enabled		
Enabled	CreditAcct	acct amnt		Enabled		Credit(acct, amnt)
Enabled	DebitAcct	acct amnt		Enabled		Debit(acct, amnt)
Enabled	CloseAcct	acct		Enabled		Close(acct)
Enabled	CashCheck	acct amnt		Enabled		WriteCheck(acct, amnt)
Enabled	Login			Enabled		
Enabled	ExecLogin			Enabled		
Enabled	Logout			Disabled		
Enabled	SelectAcct	acct		Enabled		
Enabled	ChangeRate	rate		Enabled		
Enabled	UpdateAccts			Enabled		
Executive	Logout			Disabled		
Executive	ChangeRate	rate		Executive		SetRate(acct, rate)
Executive	UpdateAccts			Executive		ComputeInterest(acct)
Executive	Login			Executive		
Executive	ExecLogin			Executive		
Executive	SelectAcct	acct		Executive		
Executive	ShowBalance	acct		Executive		
Executive	CreditAcct	acct amnt		Executive		
Executive	DebitAcct	acct amnt		Executive		
Executive	CloseAcct	acct		Executive		
Executive	CashCheck	acct amnt		Executive		

Appendix D. O-SLANG for Bank Domain Example

This appendix contains the O-SLANG specifications for the *Bank* example that were automatically generated from the ULARCH traits and state transition tables in Appendix C.

```
class Account is
  class-sort Account
  import
    Date
  sort
    Amnt, D, Account-State, Acct, Archive
  sort-axioms
    Account = Acct
  ops
    attr-equal: Account, Account -> boolean
  attributes
    acct-num: Account -> Integer
    balance: Account -> Amnt
    int-date: Account -> D
    Archive-obj: Account -> Archive
  state-attributes
    AccountState: Account -> Account-State
  methods
    create-Account: -> Account
    credit-acct: Acct, Amnt -> Acct
    debit-acct: Acct, Amnt -> Acct
    close-acct: Acct -> Acct
  states
    OK: -> Account-State
    Overdrawn: -> Account-State
  events
    new-Account: -> Account
    Credit: Account, Amnt -> Account
    Debit: Account, Amnt -> Account
    Close: Account -> Account
    ArchCredit: Archive, acct, amnt, date -> Archive
    ArchDebit: Archive, acct, amnt, date -> Archive
    ArchClose: Archive, acct, date -> Archive
  axioms
    AccountState ( a) = OK => (balance ( a) >= 0);
    AccountState ( a) = Overdrawn => (balance ( a) < 0);
    balance ( create-Account) = 0;
    acct-num ( create-Account) = 0;
    balance ( credit-acct ( ac, am)) = (balance ( ac) + am);
    balance ( debit-acct ( ac, am)) = (balance ( ac) - am);
    (AccountState ( Account-80) = OK) =>
      (AccountState ( Credit ( Account-80, amnt)) = OK &
       attr-equal ( Credit ( Account-80), credit-acct ( Account-80)) &
       (Archive-obj ( Credit ( Account-80, amnt)) = ArchCredit ( Archive-obj ( Account-80))));
    (AccountState ( Account-81) = OK & amnt > balance) =>
      (AccountState ( Debit ( Account-81, amnt)) = Overdrawn &
       (Archive-obj ( Debit ( Account-81, amnt)) = ArchDebit ( Archive-obj ( Account-81))));
    (AccountState ( Account-82) = OK & amnt <= balance) =>
      (AccountState ( Debit ( Account-82, amnt)) = OK &
       attr-equal ( Debit ( Account-82), debit-acct ( Account-82)) &
```

```
      (Archive-obj ( Debit ( Account-82, amnt)) = ArchDebit ( Archive-obj ( Account-82))));
    (AccountState ( Account-83) = OK) =>
      (AccountState ( Close ( Account-83)) = AccountEndState &
       attr-equal ( Close ( Account-83), close-acct ( Account-83)) &
       (Archive-obj ( Close ( Account-83)) = ArchClose ( Archive-obj ( Account-83))));
    (AccountState ( Account-84) = OverDrawn) =>
      (AccountState ( Debit ( Account-84, amnt)) = OverDrawn);
    (AccountState ( Account-85) = OverDrawn) =>
      (AccountState ( Close ( Account-85)) = OverDrawn);
    (AccountState ( Account-86) = Overdrawn & amnt + balance >= 0) =>
      (AccountState ( Credit ( Account-86, amnt)) = OK &
       attr-equal ( Credit ( Account-86), credit-acct ( Account-86)) &
       (Archive-obj ( Credit ( Account-86, amnt)) = ArchCredit ( Archive-obj ( Account-86))));
    (AccountState ( Account-87) = Overdrawn & amnt + balance < 0) =>
      (AccountState ( Credit ( Account-87, amnt)) = Overdrawn &
       attr-equal ( Credit ( Account-87), credit-acct ( Account-87)) &
       (Archive-obj ( Credit ( Account-87, amnt)) = ArchCredit ( Archive-obj ( Account-87))));
    AccountState ( new-Account) = OK & attr-equal ( new-Account, create-Account);
    attr-equal ( Account-89, Account-90) =>
      (acct-num ( Account-89) = acct-num ( Account-90) &
       balance ( Account-89) = balance ( Account-90) &
       int-date ( Account-89) = int-date ( Account-90) &
       Archive-obj ( Account-89) = Archive-obj ( Account-90));
    OK <> Overdrawn;
    int-date ( create-Account) = default-value;
    Archive-obj ( create-Account) = UNDEFINED;
    acct-num ( credit-acct ( Account-91, Amnt-20)) = acct-num ( Account-91);
    int-date ( credit-acct ( Account-92, Amnt-21)) = int-date ( Account-92);
    Archive-obj ( credit-acct ( Account-93, Amnt-22)) = Archive-obj ( Account-93);
    acct-num ( debit-acct ( Account-94, Amnt-23)) = acct-num ( Account-94);
    int-date ( debit-acct ( Account-95, Amnt-24)) = int-date ( Account-95);
    Archive-obj ( debit-acct ( Account-96, Amnt-25)) = Archive-obj ( Account-96);
    acct-num ( close-acct ( Account-97)) = acct-num ( Account-97);
    balance ( close-acct ( Account-98)) = balance ( Account-98);
    int-date ( close-acct ( Account-99)) = int-date ( Account-99);
    Archive-obj ( close-acct ( Account-100)) = Archive-obj ( Account-100)
end-class

class Account-Class is
  class-sort Account-Class
  contained-class Account
  methods
    create-Account-Class: -> Account-Class
  events
    new-Account-Class: -> Account-Class
    Credit: Account-Class, Amnt -> Account-Class
    Debit: Account-Class, Amnt -> Account-Class
    Close: Account-Class -> Account-Class
  axioms
    create-Account-Class = empty-set;
    new-Account-Class = create-Account-Class;
    in ( Account-76, Account-Class-10) <=>
      in ( Credit ( Account-76, Amnt), Credit ( Account-Class-10, Amnt));
    in ( Account-77, Account-Class-11) <=>
      in ( Debit ( Account-77, Amnt), Debit ( Account-Class-11, Amnt));
    in ( Account-78, Account-Class-12) <=>
      in ( Close ( Account-78), Close ( Account-Class-12))
end-class

class Archive is
  class-sort Archive
  import
    Date
  sort
    Arch, Acct, Amnt, D, Rate
```

```
  sort-axioms
    Archive = Arch
  methods
    arch-credit: Arch, Acct, Amnt, D -> Arch
    arch-debit: Arch, Acct, Amnt, D -> Arch
    arch-rate: Arch, Acct, Rate, D -> Arch
    arch-close: Arch, Acct, D -> Arch
    create-Archive: -> Archive
  events
    ArchCredit: Archive, Account, Amnt, D -> Archive
    ArchDebit: Archive, Account, Amnt, D -> Archive
    ArchRate: Archive, Account, Rate, D -> Archive
    ArchClose: Archive, Account, D -> Archive
    new-Archive: -> Archive
  axioms new-Archive = create-Archive
end-class

class Archive-Class is
  class-sort Archive-Class
  contained-class Archive
  methods
    create-Archive-Class: -> Archive-Class
  events
    new-Archive-Class: -> Archive-Class
    ArchCredit: Archive-Class, Account, Amnt, D -> Archive-Class
    ArchDebit: Archive-Class, Account, Amnt, D -> Archive-Class
    ArchRate: Archive-Class, Account, Rate, D -> Archive-Class
    ArchClose: Archive-Class, Account, D -> Archive-Class
  axioms
    create-Archive-Class = empty-set;
    new-Archive-Class = create-Archive-Class;
    in ( Archive-13, Archive-Class-13) <=>
      in ( ArchCredit ( Archive-13, Account, Amnt, D),
        ArchCredit ( Archive-Class-13, Account, Amnt, D));
    in ( Archive-14, Archive-Class-14) <=>
      in ( ArchDebit ( Archive-14, Account, Amnt, D),
        ArchDebit ( Archive-Class-14, Account, Amnt, D));
    in ( Archive-15, Archive-Class-15) <=>
      in ( ArchRate ( Archive-15, Account, Rate, D),
        ArchRate ( Archive-Class-15, Account, Rate, D));
    in ( Archive-16, Archive-Class-16) <=>
      in ( ArchClose ( Archive-16, Account, D),
        ArchClose ( Archive-Class-16, Account, D))
end-class

class Person is
  class-sort Person
  sort
    String
  sort-axioms
    Person = P
  ops
    attr-equal: Person, Person -> boolean
  attributes
    address: Person -> String
    name: Person -> String
  methods
    create-Person: -> Person
  events
    new-Person: -> Person
  axioms
    attr-equal ( new-Person, create-Person);
    attr-equal ( Person-7, Person-8) =>
      (address ( Person-7) = address ( Person-8) &
       name ( Person-7) = name ( Person-8));
```

```
    address ( create-Person) = EmptyString;
    name ( create-Person) = EmptyString
end-class

class Person-Class is
  class-sort Person-Class
  contained-class Person
  methods
    create-Person-Class: -> Person-Class
  events
    new-Person-Class: -> Person-Class
  axioms
    create-Person-Class = empty-set;
    new-Person-Class = create-Person-Class
end-class

class Customer is
  class-sort Customer < Person
  import
    Person
  sort-axioms
    Customer = Cust
  methods
    create-Customer: -> Customer
  events
    new-Customer: -> Customer
  axioms
    new-Customer = create-Customer
end-class

class Customer-Class is
  class-sort Customer-Class
  contained-class Customer
  methods
    create-Customer-Class: -> Customer-Class
  events
    new-Customer-Class: -> Customer-Class
  axioms
    create-Customer-Class = empty-set;
    new-Customer-Class = create-Customer-Class
end-class

class Employee is
  class-sort Employee < Person
  import
    String, Person
  sort
    String, int
  sort-axioms
    Employee = Empl
  ops
    attr-equal: Employee, Employee -> boolean
  attributes
    passwd: Employee -> String
    number: Employee -> Integer
  methods
    create-Employee: -> Employee
  events
    new-Employee: -> Employee
  axioms
    attr-equal ( new-Employee, create-Employee);
    attr-equal ( Employee-7, Employee-8) =>
      (passwd ( Employee-7) = passwd ( Employee-8) &
       number ( Employee-7) = number ( Employee-8));
    passwd ( create-Employee) = EmptyString;
```

```
    number ( create-Employee) = 0
end-class

class Employee-Class is
  class-sort Employee-Class
  contained-class Employee
  methods
    create-Employee-Class: -> Employee-Class
  events
    new-Employee-Class: -> Employee-Class
  axioms
    create-Employee-Class = empty-set;
    new-Employee-Class = create-Employee-Class
end-class

class Cust-Employee is
  class-sort Cust-Employee < Employee, Customer
  import
    Employee, Customer
  sort-axioms
    Cust-Employee = Cust-Empl
  methods
    create-Cust-Employee: -> Cust-Employee
  events
    new-Cust-Employee: -> Cust-Employee
  axioms
    new-Cust-Employee = create-Cust-Employee
end-class

class Cust-Employee-Class is
  class-sort Cust-Employee-Class
  contained-class Cust-Employee
  methods
    create-Cust-Employee-Class: -> Cust-Employee-Class
  events
    new-Cust-Employee-Class: -> Cust-Employee-Class
  axioms
    create-Cust-Employee-Class = empty-set;
    new-Cust-Employee-Class = create-Cust-Employee-Class
end-class

class Teller is
  class-sort Teller < Employee
  import
    Employee
  sort-axioms
    Teller = Tell
  methods
    create-Teller: -> Teller
  events
    new-Teller: -> Teller
  axioms
    new-Teller = create-Teller
end-class

class Teller-Class is
  class-sort Teller-Class
  contained-class Teller
  methods
    create-Teller-Class: -> Teller-Class
  events
    new-Teller-Class: -> Teller-Class
  axioms
    create-Teller-Class = empty-set;
    new-Teller-Class = create-Teller-Class
```

```
end-class

class Executive is
  class-sort Executive < Employee
  import
    Employee
  sort-axioms
    Executive = Exec
  methods
    create-Executive: -> Executive
  events
    new-Executive: -> Executive
  axioms
    new-Executive = create-Executive
end-class

class Executive-Class is
  class-sort Executive-Class
  contained-class Executive
  methods
    create-Executive-Class: -> Executive-Class
  events
    new-Executive-Class: -> Executive-Class
  axioms
    create-Executive-Class = empty-set;
    new-Executive-Class = create-Executive-Class
end-class

class Checking is
  class-sort Checking < Account
  import
    Account
  sort
    CAcct, Amnt, Archive
  sort-axioms
    Checking = CAcct
  ops
    attr-equal: Checking, Checking -> boolean
  attributes
    Archive-obj: Checking -> Archive
  methods
    write-check: CAcct, Amnt -> CAcct
    create-Checking: -> Checking
  events
    WriteCheck: Checking, Amnt -> Checking
    ArchDebit: Archive, acct, amnt, date -> Archive
    new-Checking: -> Checking
  axioms
    (CheckingState ( Checking-6) = OK & amnt > balance) =>
      (CheckingState ( WriteCheck ( Checking-6, amnt)) = Overdrawn &
       (Archive-obj ( WriteCheck ( Checking-6, amnt)) = ArchDebit ( Archive-obj ( Checking-6))));
    (CheckingState ( Checking-7) = OK & amnt <= balance) =>
      (CheckingState ( WriteCheck ( Checking-7, amnt)) = OK &
       attr-equal ( WriteCheck ( Checking-7), write-check ( Checking-7)) &
       (Archive-obj ( WriteCheck ( Checking-7, amnt)) = ArchDebit ( Archive-obj ( Checking-7))));
    (CheckingState ( Checking-8) = OverDrawn) =>
       (CheckingState ( WriteCheck ( Checking-8, amnt)) = OverDrawn);
    attr-equal ( new-Checking, create-Checking);
    CheckingState ( new-Checking) = OK & attr-equal ( new-Checking, create-Checking);
    attr-equal ( Checking-10, Checking-11) =>
      (Archive-obj ( Checking-10) = Archive-obj ( Checking-11));
    balance ( write-check ( c, a)) = balance ( debit-acct ( c, a));
    int-date ( write-check ( c, a)) = int-date ( c);
    acct-num ( write-check ( c, a)) = acct-num ( c);
    Archive-obj ( write-check ( Checking-12, Amnt-19)) = Archive-obj ( Checking-12);
```

```
    Archive-obj ( create-Checking) = UNDEFINED
end-class

class Checking-Class is
  class-sort Checking-Class
  contained-class Checking
  methods
    create-Checking-Class: -> Checking-Class
  events
    new-Checking-Class: -> Checking-Class
    WriteCheck: Checking-Class, Amnt -> Checking-Class
  axioms
    create-Checking-Class = empty-set;
    new-Checking-Class = create-Checking-Class;
    in ( Checking-4, Checking-Class-4) <=>
      in ( WriteCheck ( Checking-4, Amnt), WriteCheck ( Checking-Class-4, Amnt))
end-class

class Savings is
  class-sort Savings < Account
  import
    Date, Account
  sort
    D, Rate, SAcct, Amnt
  sort-axioms
    Savings = SAcct
  ops
    attr-equal: Savings, Savings -> boolean
  attributes
    date-int-computed: Savings -> D rate: Savings -> Rate
  methods
    set-rate: SAcct, Rate -> SAcct
    compute-interest: SAcct -> Amnt
    create-Savings: -> Savings
  events
    SetRate: Savings, Rate -> Savings
    ComputeInterest: Savings -> Savings
    new-Savings: -> Savings
  axioms
    (SavingsState ( Savings-24) = OK) =>
      (SavingsState ( SetRate ( Savings-24, rate)) = OK &
       attr-equal ( SetRate ( Savings-24), set_rate ( Savings-24)));
    (SavingsState ( Savings-25) = OK) =>
      (SavingsState ( ComputeInterest ( Savings-25)) = OK &
       attr-equal ( ComputeInterest ( Savings-25), compute-interest ( Savings-25)));
    (SavingsState ( Savings-26) = OverDrawn) =>
      (SavingsState ( SetRate ( Savings-26, amnt)) = OverDrawn &
       attr-equal ( SetRate ( Savings-26), set_rate ( Savings-26)));
    (SavingsState ( Savings-27) = OverDrawn) =>
      (SavingsState ( ComputeInterest ( Savings-27)) = OverDrawn &
       attr-equal ( ComputeInterest ( Savings-27), compute-interest ( Savings-27)));
    attr-equal ( new-Savings, create-Savings);
    SavingsState ( new-Savings) = OK & attr-equal ( new-Savings, create-Savings);
    attr-equal ( Savings-29, Savings-30) =>
      (date-int-computed ( Savings-29) = date-int-computed ( Savings-30) &
       rate ( Savings-29) = rate ( Savings-30));
    compute-interest ( sa, r) = (balance ( sa) * rate ( sa));
    rate ( set-rate ( sa, r)) = r;
    int-date ( set-rate ( sa, r)) = current-date;
    balance ( set-rate ( sa, r)) = balance ( sa);
    Acct-num ( set-rate ( sa, r)) = Acct-num ( sa);
    date-int-computed ( set-rate ( Savings-31, Rate-4)) = date-int-computed ( Savings-31);
    int-date ( compute-interest ( sa, r)) = int-date ( sa);
    balance ( compute-interest ( sa, r)) = balance ( sa);
    Acct-num ( compute-interest ( sa, r)) = Acct-num ( sa);
```

```
    date-int-computed ( compute-interest ( Savings-32)) = date-int-computed ( Savings-32);
    rate ( compute-interest ( Savings-33)) = rate ( Savings-33);
    date-int-computed ( create-Savings) = default-value;
    rate ( create-Savings) = default-value
end-class

class Savings-Class is
  class-sort Savings-Class
  contained-class Savings
  methods
    create-Savings-Class: -> Savings-Class
  events
    new-Savings-Class: -> Savings-Class
    SetRate: Savings-Class, Rate -> Savings-Class
    ComputeInterest: Savings-Class -> Savings-Class
  axioms
    create-Savings-Class = empty-set;
    new-Savings-Class = create-Savings-Class;
    in ( Savings-21, Savings-Class-7) <=>
      in ( SetRate ( Savings-21, Rate), SetRate ( Savings-Class-7, Rate));
    in ( Savings-22, Savings-Class-8) <=>
      in ( ComputeInterest ( Savings-22), ComputeInterest ( Savings-Class-8))
end-class

class Combined is
  class-sort Combined < Savings, Checking
  import
    Savings, Checking
  methods
    create-Combined: -> Combined
  events
    new-Combined: -> Combined
  axioms
    new-Combined = create-Combined
end-class

class Combined-Class is
  class-sort Combined-Class
  contained-class Combined
  methods
    create-Combined-Class: -> Combined-Class
  events
    new-Combined-Class: -> Combined-Class
  axioms
    create-Combined-Class = empty-set;
    new-Combined-Class = create-Combined-Class
end-class

class Console is
  class-sort Console
  sort
    int, Console-State, Savings, Checking, Account
  sort-axioms
    Console = Cons
  ops
    attr-equal: Console, Console -> boolean
  attributes
    id: Console -> Integer
    Savings-obj: Console -> Savings
    Checking-obj: Console -> Checking
    Account-obj: Console -> Account
  state-attributes
    ConsoleState: Console -> Console-State
  methods
    create-Console: -> Console
```

```
states
  LoggedIn: -> Console-State
  Disabled: -> Console-State
  Enabled: -> Console-State
  Executive: -> Console-State
events
  new-Console: -> Console
  Login: Console -> Console
  Logout: Console -> Console
  ExecLogin: Console -> Console
  ChangeRate: Console, Rate -> Console
  UpdateAccts: Console -> Console
  SelectAcct: Console, Account -> Console
  ShowBalance: Console, Account -> Console
  CreditAcct: Console, Account, Amnt -> Console
  DebitAcct: Console, Account, Amnt -> Console
  CloseAcct: Console, Account -> Console
  CashCheck: Console, Account, Amnt -> Console
  Credit: Account, acct, amnt -> Account
  Debit: Account, acct, amnt -> Account
  Close: Account, acct -> Account
  WriteCheck: Checking, acct, amnt -> Checking
  SetRate: Savings, acct, rate -> Savings
  ComputeInterest: Savings, acct -> Savings
axioms
  id ( create-Console) = 0;
  (ConsoleState ( Console-190) = Disabled) =>
    (ConsoleState ( Login ( Console-190)) = LoggedIn);
  (ConsoleState ( Console-191) = Disabled) =>
    (ConsoleState ( ExecLogin ( Console-191)) = Executive);
  (ConsoleState ( Console-192) = Disabled) =>
    (ConsoleState ( Logout ( Console-192)) = Disabled);
  (ConsoleState ( Console-193) = Disabled) =>
    (ConsoleState ( SelectAcct ( Console-193, acct)) = Disabled);
  (ConsoleState ( Console-194) = Disabled) =>
    (ConsoleState ( ShowBalance ( Console-194, acct)) = Disabled);
  (ConsoleState ( Console-195) = Disabled) =>
    (ConsoleState ( CreditAcct ( Console-195, acct, amnt)) = Disabled);
  (ConsoleState ( Console-196) = Disabled) =>
    (ConsoleState ( DebitAcct ( Console-196, acct, amnt)) = Disabled);
  (ConsoleState ( Console-197) = Disabled) =>
    (ConsoleState ( CloseAcct ( Console-197, acct)) = Disabled);
  (ConsoleState ( Console-198) = Disabled) =>
    (ConsoleState ( CashCheck ( Console-198, acct, amnt)) = Disabled);
  (ConsoleState ( Console-199) = Disabled) =>
    (ConsoleState ( ChangeRate ( Console-199, rate)) = Disabled);
  (ConsoleState ( Console-200) = Disabled) =>
    (ConsoleState ( UpdateAccts ( Console-200)) = Disabled);
  (ConsoleState ( Console-201) = LoggedIn) =>
    (ConsoleState ( Logout ( Console-201)) = Disabled);
  (ConsoleState ( Console-202) = LoggedIn) =>
    (ConsoleState ( SelectAcct ( Console-202, acct)) = Enabled);
  (ConsoleState ( Console-203) = LoggedIn) =>
    (ConsoleState ( Login ( Console-203)) = LoggedIn);
  (ConsoleState ( Console-204) = LoggedIn) =>
    (ConsoleState ( ExecLogin ( Console-204)) = LoggedIn);
  (ConsoleState ( Console-205) = LoggedIn) =>
    (ConsoleState ( ShowBalance ( Console-205, acct)) = LoggedIn);
  (ConsoleState ( Console-206) = LoggedIn) =>
    (ConsoleState ( CreditAcct ( Console-206, acct, amnt)) = LoggedIn);
  (ConsoleState ( Console-207) = LoggedIn) =>
    (ConsoleState ( DebitAcct ( Console-207, acct, amnt)) = LoggedIn);
  (ConsoleState ( Console-208) = LoggedIn) =>
    (ConsoleState ( CloseAcct ( Console-208, acct)) = LoggedIn);
  (ConsoleState ( Console-209) = LoggedIn) =>
```

```
  (ConsoleState ( CashCheck ( Console-209, acct, amnt)) = LoggedIn);
(ConsoleState ( Console-210) = LoggedIn) =>
  (ConsoleState ( ChangeRate ( Console-210, rate)) = LoggedIn);
(ConsoleState ( Console-211) = LoggedIn) =>
  (ConsoleState ( UpdateAccts ( Console-211)) = LoggedIn);
(ConsoleState ( Console-212) = Enabled) =>
  (ConsoleState ( ShowBalance ( Console-212, acct)) = Enabled);
(ConsoleState ( Console-213) = Enabled) =>
  (ConsoleState ( CreditAcct ( Console-213, acct, amnt)) = Enabled &
   (Account-obj ( CreditAcct ( Console-213, acct, amnt)) = Credit ( Account-obj ( Console-213))));
(ConsoleState ( Console-214) = Enabled) =>
  (ConsoleState ( DebitAcct ( Console-214, acct, amnt)) = Enabled &
   (Account-obj ( DebitAcct ( Console-214, acct, amnt)) = Debit ( Account-obj ( Console-214))));
(ConsoleState ( Console-215) = Enabled) =>
  (ConsoleState ( CloseAcct ( Console-215, acct)) = Enabled &
   (Account-obj ( CloseAcct ( Console-215, acct)) = Close ( Account-obj ( Console-215))));
(ConsoleState ( Console-216) = Enabled) =>
  (ConsoleState ( CashCheck ( Console-216, acct, amnt)) = Enabled &
   (Checking-obj ( CashCheck ( Console-216, acct, amnt)) = WriteCheck ( Checking-obj ( Console-216))));
(ConsoleState ( Console-217) = Enabled) =>
  (ConsoleState ( Login ( Console-217)) = Enabled);
(ConsoleState ( Console-218) = Enabled) =>
  (ConsoleState ( ExecLogin ( Console-218)) = Enabled);
(ConsoleState ( Console-219) = Enabled) =>
  (ConsoleState ( Logout ( Console-219)) = Disabled);
(ConsoleState ( Console-220) = Enabled) =>
  (ConsoleState ( SelectAcct ( Console-220, acct)) = Enabled);
(ConsoleState ( Console-221) = Enabled) =>
  (ConsoleState ( ChangeRate ( Console-221, rate)) = Enabled);
(ConsoleState ( Console-222) = Enabled) =>
  (ConsoleState ( UpdateAccts ( Console-222)) = Enabled);
(ConsoleState ( Console-223) = Executive) =>
  (ConsoleState ( Logout ( Console-223)) = Disabled);
(ConsoleState ( Console-224) = Executive) =>
  (ConsoleState ( ChangeRate ( Console-224, rate)) = Executive &
   (Savings-obj ( ChangeRate ( Console-224, rate)) = SetRate ( Savings-obj ( Console-224))));
(ConsoleState ( Console-225) = Executive) =>
  (ConsoleState ( UpdateAccts ( Console-225)) = Executive &
   (Savings-obj ( UpdateAccts ( Console-225)) = ComputeInterest ( Savings-obj ( Console-225))));
(ConsoleState ( Console-226) = Executive) =>
  (ConsoleState ( Login ( Console-226)) = Executive);
(ConsoleState ( Console-227) = Executive) =>
  (ConsoleState ( ExecLogin ( Console-227)) = Executive);
(ConsoleState ( Console-228) = Executive) =>
  (ConsoleState ( SelectAcct ( Console-228, acct)) = Executive);
(ConsoleState ( Console-229) = Executive) =>
  (ConsoleState ( ShowBalance ( Console-229, acct)) = Executive);
(ConsoleState ( Console-230) = Executive) =>
  (ConsoleState ( CreditAcct ( Console-230, acct, amnt)) = Executive);
(ConsoleState ( Console-231) = Executive) =>
  (ConsoleState ( DebitAcct ( Console-231, acct, amnt)) = Executive);
(ConsoleState ( Console-232) = Executive) =>
  (ConsoleState ( CloseAcct ( Console-232, acct)) = Executive);
(ConsoleState ( Console-233) = Executive) =>
  (ConsoleState ( CashCheck ( Console-233, acct, amnt)) = Executive);
ConsoleState ( new-Console) = Disabled & attr-equal ( new-Console, create-Console);
attr-equal ( Console-235, Console-236) =>
  (id ( Console-235) = id ( Console-236) &
   Savings-obj ( Console-235) = Savings-obj ( Console-236) &
   Checking-obj ( Console-235) = Checking-obj ( Console-236) &
   Account-obj ( Console-235) = Account-obj ( Console-236));
LoggedIn <> Disabled;
LoggedIn <> Enabled;
LoggedIn <> Executive;
Disabled <> Enabled;
```

```
    Disabled <> Executive;
    Enabled <> Executive;
    Savings-obj ( create-Console) = UNDEFINED;
    Checking-obj ( create-Console) = UNDEFINED;
    Account-obj ( create-Console) = UNDEFINED
end-class

class Console-Class is
  class-sort Console-Class
  contained-class Console
  methods
    create-Console-Class: -> Console-Class
  events
    new-Console-Class: -> Console-Class
    Login: Console-Class -> Console-Class
    Logout: Console-Class -> Console-Class
    ExecLogin: Console-Class -> Console-Class
    ChangeRate: Console-Class -> Console-Class
    UpdateAccts: Console-Class -> Console-Class
    SelectAcct: Console-Class -> Console-Class
    ShowBalance: Console-Class -> Console-Class
    CreditAcct: Console-Class, Amnt -> Console-Class
    DebitAcct: Console-Class, Amnt -> Console-Class
    CloseAcct: Console-Class -> Console-Class
    CashCheck: Console-Class, Amnt -> Console-Class
  axioms
    create-Console-Class = empty-set;
    new-Console-Class = create-Console-Class;
    in ( Console-178, Console-Class-34) <=>
      in ( Login ( Console-178), Login ( Console-Class-34));
    in ( Console-179, Console-Class-35) <=>
      in ( Logout ( Console-179), Logout ( Console-Class-35));
    in ( Console-180, Console-Class-36) <=>
      in ( ExecLogin ( Console-180), ExecLogin ( Console-Class-36));
    in ( Console-181, Console-Class-37) <=>
      in ( ChangeRate ( Console-181), ChangeRate ( Console-Class-37));
    in ( Console-182, Console-Class-38) <=>
      in ( UpdateAccts ( Console-182), UpdateAccts ( Console-Class-38));
    in ( Console-183, Console-Class-39) <=>
      in ( SelectAcct ( Console-183), SelectAcct ( Console-Class-39));
    in ( Console-184, Console-Class-40) <=>
      in ( ShowBalance ( Console-184), ShowBalance ( Console-Class-40));
    in ( Console-185, Console-Class-41) <=>
      in ( CreditAcct ( Console-185, Amnt), CreditAcct ( Console-Class-41, Amnt));
    in ( Console-186, Console-Class-42) <=>
      in ( DebitAcct ( Console-186, Amnt), DebitAcct ( Console-Class-42, Amnt));
    in ( Console-187, Console-Class-43) <=>
      in ( CloseAcct ( Console-187), CloseAcct ( Console-Class-43));
    in ( Console-188, Console-Class-44) <=>
         in ( CashCheck ( Console-188, Amnt), CashCheck ( Console-Class-44, Amnt))
end-class

link Op-Link is
  class-sort Op-Link
  sort
    Console, Employee
  ops
    attr-equal: Op-Link, Op-Link -> boolean
  attributes
    a-console: Op-Link -> Console
    an-employee: Op-Link -> Employee
  methods
    create-Op-Link: Employee, Console -> Op-Link
  events
    new-Op-Link: Employee, Console -> Op-Link
```

```
  axioms
    attr-equal ( new-Op-Link ( a-console-7, an-employee-4),
      create-Op-Link ( a-console-7, an-employee-4));
    a-console ( create-Op-Link ( a-console-7, an-employee-4)) = a-console-7;
    an-employee ( create-Op-Link ( a-console-7, an-employee-4)) = an-employee-4;
    attr-equal ( Op-Link-7, Op-Link-8) =>
      (a-console ( Op-Link-7) = a-console ( Op-Link-8) &
       an-employee ( Op-Link-7) = an-employee ( Op-Link-8))
end-link

association Operates is
  class-sort Operates link-class Op-Link
  sort
    Bool, Opers, Employee, Console, Employees, Consoles
  sort-axioms
    Operates = Opers
  methods
    does-operate: Opers, Employee, Console -> Bool
    image: Opers, Console -> Employees
    image: Opers, Employee -> Consoles
    create-Operates: -> Operates
  events
    new-Operates: -> Operates
  axioms
    (in ( x, o) & (a-console ( x) = c)) <=> in ( an-employee ( x), image ( o, c));
    (in ( x, o) & (an-employee ( x) = e)) <=> in ( a-console ( x), image ( o, e));
    Size ( image ( o, e)) = 1;
    Size ( image ( o, c)) >= 0;
    new-Operates = empty-set;
    does-operate ( new-Operates, e, c) = false;
    does-operate ( o, e, c) <=> (in ( e, image ( o, c)) & in ( c, image ( o, e)))
end-association

link Own-Link is
  class-sort Own-Link
  sort
    Account, Customer
  ops
    attr-equal: Own-Link, Own-Link -> boolean
  attributes
    an-account: Own-Link -> Account
    a-customer: Own-Link -> Customer
  methods
    create-Own-Link: Customer, Account -> Own-Link
  events
    new-Own-Link: Customer, Account -> Own-Link
  axioms
    attr-equal ( new-Own-Link ( an-account-13, a-customer-4),
      create-Own-Link ( an-account-13, a-customer-4));
    an-account ( create-Own-Link ( an-account-13, a-customer-4)) = an-account-13;
    a-customer ( create-Own-Link ( an-account-13, a-customer-4)) = a-customer-4;
    attr-equal ( Own-Link-7, Own-Link-8) =>
      (an-account ( Own-Link-7) = an-account ( Own-Link-8) &
       a-customer ( Own-Link-7) = a-customer ( Own-Link-8))
end-link

association Owns is
  class-sort Owns link-class Own-Link
  sort
    Customer, Account, Bool, O, Customers, Accounts
  sort-axioms
    Owns = O
  methods
    does-own: O, Customer, Account -> Bool
    image: O, Account -> Customers
```

```
    image: O, Customer -> Accounts
    create-Owns: Customer, Account -> Owns
  events
    new-Owns: Customer, Account -> Owns
  axioms
    Size ( image ( o, c)) >= 0;
    Size ( image ( o, a)) = 1;
    (in ( x, o) & (a-customer ( x) = c)) <=> in ( an-account ( x), image ( o, c));
    (in ( x, o) & (an-account ( x) = a)) <=> in ( a-customer ( x), image ( o, a));
    new-Owns = empty-set;
    does-own ( new-Owns, c, a) = false;
    does-own ( o, c, a) <=> (in ( c, image ( o, a)) & in ( a, image ( o, c)))
end-association

link Manipulate-Link is
  class-sort Manipulate-Link
  sort
    Account, Console
  ops
    attr-equal: Manipulate-Link, Manipulate-Link -> boolean
  attributes
    an-account: Manipulate-Link -> Account
    a-console: Manipulate-Link -> Console
  methods
    create-Manipulate-Link: Console, Account -> Manipulate-Link
  events
    new-Manipulate-Link: Console, Account -> Manipulate-Link
  axioms
    attr-equal ( new-Manipulate-Link ( an-account-14, a-console-8),
      create-Manipulate-Link ( an-account-14, a-console-8));
    an-account ( create-Manipulate-Link ( an-account-14, a-console-8)) = an-account-14;
    a-console ( create-Manipulate-Link ( an-account-14, a-console-8)) = a-console-8;
    attr-equal ( Manipulate-Link-7, Manipulate-Link-8) =>
     (an-account ( Manipulate-Link-7) = an-account ( Manipulate-Link-8) &
      a-console ( Manipulate-Link-7) = a-console ( Manipulate-Link-8))
end-link

association Manipulates is
  class-sort Manipulates link-class Manipulate-Link
  sort
    Console, Account, Bool, Manips, Consoles, Accounts
  sort-axioms
    Manipulates = Manips
  methods
    does-manipulate: Manips, Console, Account -> Bool
    image: Manips, Account -> Consoles
    image: Manips, Console -> Accounts
    create-Manipulates: Console, Account -> Manipulates
  events
    new-Manipulates: Console, Account -> Manipulates
  axioms
    Size ( image ( m, c)) >= 0;
    Size ( image ( m, a)) >= 0;
    (in ( x, m) & (a-console ( x) = c)) <=> in ( an-account ( x), image ( m, c));
    (in ( x, m) & (an-account ( x) = a)) <=> in ( a-console ( x), image ( m, a));
    new-Manipulates = empty-set;
    does-manipulate ( new-Manipulates, c, a) = false;
    does-manipulate ( m, c, a) <=> (in ( c, image ( m, a)) & in ( a, image ( m, c)))
end-association

link Ar-Link is
  class-sort Ar-Link
  sort
    Archive, Account
  ops
```

```
    attr-equal: Ar-Link, Ar-Link -> boolean
  attributes
    an-archive: Ar-Link -> Archive
    an-account: Ar-Link -> Account
  methods
    create-Ar-Link: Account, Archive -> Ar-Link
  events
    new-Ar-Link: Account, Archive -> Ar-Link
  axioms
    attr-equal ( new-Ar-Link ( an-archive-4, an-account-15),
      create-Ar-Link ( an-archive-4, an-account-15));
    an-archive ( create-Ar-Link ( an-archive-4, an-account-15)) = an-archive-4;
    an-account ( create-Ar-Link ( an-archive-4, an-account-15)) = an-account-15;
    attr-equal ( Ar-Link-7, Ar-Link-8) =>
     (an-archive ( Ar-Link-7) = an-archive ( Ar-Link-8) &
      an-account ( Ar-Link-7) = an-account ( Ar-Link-8))
end-link

association Archives is
  class-sort Archives link-class Ar-Link
  sort
    Account, Archive, Bool, Archs, Accounts, ArchiveSet
  sort-axioms
    Archives = Archs
  methods
    does-archive: Archs, Account, Archive -> Bool
    image: Archs, Archive -> Accounts
    image: Archs, Account -> ArchiveSet
    create-Archives: Account, Archive -> Archives
  events
    new-Archives: Account, Archive -> Archives
  axioms
    Size ( image ( ars, ac)) = 1;
    Size ( image ( ars, ar)) >= 0;
    (in ( x, ars) & (an-account ( x) = ac)) <=> in ( an-archive ( x), image ( ars, ac));
    (in ( x, ars) & (an-archive ( x) = ar)) <=> in ( an-account ( x), image ( ars, ar));
    new-Archives = empty-set;
    does-archive ( new-Archives, ac, ar) = false;
    does-archive ( ars, ac, ar) <=> (in ( ac, image ( ars, ar)) & in ( ar, image ( ars, ac)))
end-association

link Access-Link is
  class-sort Access-Link
  sort
    Account, D
  ops
    attr-equal: Access-Link, Access-Link -> boolean
  attributes
    a-date: Access-Link -> Account
    an-account: Access-Link -> Account
  methods
    create-Access-Link: Account, D -> Access-Link
  events
    new-Access-Link: Account, D -> Access-Link
  axioms
    attr-equal ( new-Access-Link ( a-date-4, an-account-16),
      create-Access-Link ( a-date-4, an-account-16));
    a-date ( create-Access-Link ( a-date-4, an-account-16)) = a-date-4;
    an-account ( create-Access-Link ( a-date-4, an-account-16)) = an-account-16;
    attr-equal ( Access-Link-7, Access-Link-8) =>
      (a-date ( Access-Link-7) = a-date ( Access-Link-8) &
       an-account ( Access-Link-7) = an-account ( Access-Link-8))
end-link

association Accesses is
```

```
  class-sort Accesses link-class Access-Link
  sort
    Account, D, Bool, Accs, Accounts, Dates
  sort-axioms
    Accesses = Accs
  methods
    does-access: Accs, Account, D -> Bool
    image: Accs, D -> Accounts
    image: Accs, Account -> Dates
    create-Accesses: Account, D -> Accesses
  events
    new-Accesses: Account, D -> Accesses
  axioms
    Size ( image ( acs, ac)) = 1;
    Size ( image ( acs, d)) >= 0;
    (in ( x, acs) & (an-account ( x) = ac)) <=> in ( a-date ( x), image ( acs, ac));
    (in ( x, acs) & (a-date ( x) = d)) <=> in ( an-account ( x), image ( acs, d));
    new-Accesses = empty-set;
    does-access ( new-Accesses, ac, d) = false;
    does-access ( acs, ac, d) <=> (in ( ac, image ( acs, d)) & in ( d, image ( acs, ac)))
end-association

class Bank is
  class-sort Bank
  import
    Owns, Bank-aggregate
  ops
    attr-equal: Bank, Bank -> boolean
  attributes
    Person-Class-obj: Bank -> Person-Class
    Customer-Class-obj: Bank -> Customer-Class
    Employee-Class-obj: Bank -> Employee-Class
    Cust-Employee-Class-obj: Bank -> Cust-Employee-Class
    Teller-Class-obj: Bank -> Teller-Class
    Executive-Class-obj: Bank -> Executive-Class
    Console-Class-obj: Bank -> Console-Class
    Account-Class-obj: Bank -> Account-Class
    Checking-Class-obj: Bank -> Checking-Class
    Savings-Class-obj: Bank -> Savings-Class
    Combined-Class-obj: Bank -> Combined-Class
    Archive-Class-obj: Bank -> Archive-Class
    Owns-obj: Bank -> Owns
  methods
    create-Bank: -> Bank events new-Bank: -> Bank
  axioms
    size ( Person-Class-obj ( Bank-64)) >= 0;
    size ( Customer-Class-obj ( Bank-65)) >= 0;
    size ( Employee-Class-obj ( Bank-66)) >= 0;
    size ( Cust-Employee-Class-obj ( Bank-67)) >= 0;
    size ( Teller-Class-obj ( Bank-68)) >= 0;
    size ( Executive-Class-obj ( Bank-69)) >= 0;
    size ( Console-Class-obj ( Bank-70)) >= 0;
    size ( Account-Class-obj ( Bank-71)) >= 0;
    size ( Checking-Class-obj ( Bank-72)) >= 0;
    size ( Savings-Class-obj ( Bank-73)) >= 0;
    size ( Combined-Class-obj ( Bank-74)) >= 0;
    size ( Archive-Class-obj ( Bank-75)) >= 0;
    Person-Class-obj ( Bank-76) =
      Union ( Customer-Class-obj ( Bank-76), Employee-Class-obj ( Bank-76));
    Employee-Class-obj ( Bank-77) =
      Union ( Teller-Class-obj ( Bank-77), Executive-Class-obj ( Bank-77));
    SubSet ( Cust-Employee-Class-obj ( Bank-78), Customer-Class-obj ( Bank-78));
    SubSet ( Cust-Employee-Class-obj ( Bank-79), Employee-Class-obj ( Bank-79));
    Account-Class-obj ( Bank-80) =
      Union ( Checking-Class-obj ( Bank-80), Savings-Class-obj ( Bank-80));
```

```
    SubSet ( Combined-Class-obj ( Bank-81), Checking-Class-obj ( Bank-81));
    SubSet ( Combined-Class-obj ( Bank-82), Savings-Class-obj ( Bank-82));
    attr-equal ( new-Bank, create-Bank);
    attr-equal ( Bank-83, Bank-84) =>
      (Person-Class-obj ( Bank-83) = Person-Class-obj ( Bank-84) &
       Customer-Class-obj ( Bank-83) = Customer-Class-obj ( Bank-84) &
       Employee-Class-obj ( Bank-83) = Employee-Class-obj ( Bank-84) &
       Cust-Employee-Class-obj ( Bank-83) = Cust-Employee-Class-obj ( Bank-84) &
       Teller-Class-obj ( Bank-83) = Teller-Class-obj ( Bank-84) &
       Executive-Class-obj ( Bank-83) = Executive-Class-obj ( Bank-84) &
       Console-Class-obj ( Bank-83) = Console-Class-obj ( Bank-84) &
       Account-Class-obj ( Bank-83) = Account-Class-obj ( Bank-84) &
       Checking-Class-obj ( Bank-83) = Checking-Class-obj ( Bank-84) &
       Savings-Class-obj ( Bank-83) = Savings-Class-obj ( Bank-84) &
       Combined-Class-obj ( Bank-83) = Combined-Class-obj ( Bank-84) &
       Archive-Class-obj ( Bank-83) = Archive-Class-obj ( Bank-84) &
       Owns-obj ( Bank-83) = Owns-obj ( Bank-84));
    Person-Class-obj ( create-Bank) = new-Person-Class;
    Customer-Class-obj ( create-Bank) = new-Customer-Class;
    Employee-Class-obj ( create-Bank) = new-Employee-Class;
    Cust-Employee-Class-obj ( create-Bank) = new-Cust-Employee-Class;
    Teller-Class-obj ( create-Bank) = new-Teller-Class;
    Executive-Class-obj ( create-Bank) = new-Executive-Class;
    Console-Class-obj ( create-Bank) = new-Console-Class;
    Account-Class-obj ( create-Bank) = new-Account-Class;
    Checking-Class-obj ( create-Bank) = new-Checking-Class;
    Savings-Class-obj ( create-Bank) = new-Savings-Class;
    Combined-Class-obj ( create-Bank) = new-Combined-Class;
    Archive-Class-obj ( create-Bank) = new-Archive-Class;
    Owns-obj ( create-Bank) = UNDEFINED
end-class

class Bank-Class is
  class-sort Bank-Class
  contained-class Bank
  methods
    create-Bank-Class: -> Bank-Class
  events
    new-Bank-Class: -> Bank-Class
  axioms
    create-Bank-Class = empty-set;
    new-Bank-Class = create-Bank-Class
end-class

aggregate Bank-aggregate is
  nodes
    Person-Class, Customer-Class, Employee-Class,
    Cust-Employee-Class, Teller-Class, Executive-Class,
    Console-Class, Account-Class, Checking-Class, Savings-Class,
    Combined-Class, Archive-Class, Owns, Integer, SET-58: Set,
    SET-59: Set, SET-60: Set, SET-61: Set, SET-62: Set,
    SET-63: Set, SET-64: Set, SET-65: Set, SET-66: Set,
    SET-67: Set, SET-68: Set, SET-69: Set, SET-70: Set, Credit,
    Debit, Close, WriteCheck, SetRate, ComputeInterest,
    ArchDebit, ArchCredit, ArchClose
  arcs
    SET-58 -> Person-Class: { Set -> Person-Class, E -> Person},
    SET-59 -> Customer-Class: { Set -> Customer-Class, E -> Customer},
    SET-60 -> Employee-Class: { Set -> Employee-Class, E -> Employee},
    SET-61 -> Cust-Employee-Class: { Set -> Cust-Employee-Class, E -> Cust-Employee},
    SET-62 -> Teller-Class: { Set -> Teller-Class, E -> Teller},
    SET-63 -> Executive-Class: { Set -> Executive-Class, E -> Executive},
    SET-64 -> Console-Class: { Set -> Console-Class, E -> Console},
    SET-65 -> Account-Class: { Set -> Account-Class, E -> Account},
    SET-66 -> Checking-Class: { Set -> Checking-Class, E -> Checking},
```

```
    SET-67 -> Savings-Class: { Set -> Savings-Class, E -> Savings},
    SET-68 -> Combined-Class: { Set -> Combined-Class, E -> Combined},
    SET-69 -> Archive-Class: { Set -> Archive-Class, E -> Archive},
    Integer -> SET-58: {},
    Integer -> SET-59: {},
    Integer -> SET-60: {},
    Integer -> SET-61: {},
    Integer -> SET-62: {},
    Integer -> SET-63: {},
    Integer -> SET-64: {},
    Integer -> SET-65: {},
    Integer -> SET-66: {},
    Integer -> SET-67: {},
    Integer -> SET-68: {},
    Integer -> SET-69: {},
    Integer -> SET-70: {},
    SET-70 -> Owns: { Set -> Owns, E -> Own-Link},
    SET-59 -> Owns: { Set -> Customers, E -> Customer},
    SET-65 -> Owns: { Set -> Accounts, E -> Account},
    Credit -> Console-Class: {},
    Credit -> Account-Class: { Credit -> Account-Class},
    Debit -> Console-Class: {},
    Debit -> Account-Class: { Debit -> Account-Class},
    Close -> Console-Class: {},
    Close -> Account-Class: { Close -> Account-Class},
    WriteCheck -> Console-Class: {},
    WriteCheck -> Checking-Class: { WriteCheck -> Checking-Class},
    SetRate -> Console-Class: {},
    SetRate -> Savings-Class: { SetRate -> Savings-Class},
    ComputeInterest -> Console-Class: {},
    ComputeInterest -> Savings-Class: { ComputeInterest -> Savings-Class},
    ArchDebit -> Checking-Class: {},
    ArchDebit -> Archive-Class: { ArchDebit -> Archive-Class},
    ArchCredit -> Account-Class: {},
    ArchCredit -> Archive-Class: { ArchCredit -> Archive-Class},
    ArchDebit -> Account-Class: {},
    ArchDebit -> Archive-Class: { ArchDebit -> Archive-Class},
    ArchClose -> Account-Class: {},
    ArchClose -> Archive-Class: { ArchClose -> Archive-Class},
    Acct -> Acct-Class: {},
    Acct -> Checking: {},
    Acct -> Savings: {},
    Checking -> Checking-Class: {},
    Checking -> Combined: {},
    Savings -> Savings-Class: {},
    Savings -> Combined: {},
    Combined -> Combined-Class: {},
    Person -> Person-Class: {},
    Person -> Customer: {},
    Person -> Employee: {},
    Employee -> Employee-Class: {},
    Employee -> Exec: {},
    Employee -> Teller: {},
    Employee -> Cust-Employee: {},
    Exec -> Exec-Class: {},
    Customer -> Customer-Class: {},
    Customer -> Cust-Employee: {},
    Cust-Employee -> Cust-Employee-Class: {},
    Teller -> Teller-Class: {}
end-aggregate

class Date is
  class-sort Date
  import
    String
```

```
  sort-axioms
    Date = D
  methods
    create-Date: -> Date
    current-date: D -> String
  events
    new-Date: -> Date
  axioms
    new-Date = create-Date
end-class

class Date-Class is
  class-sort Date-Class
  contained-class Date
  methods
    create-Date-Class: -> Date-Class
  events
    new-Date-Class: -> Date-Class
  axioms
    create-Date-Class = empty-set;
    new-Date-Class = create-Date-Class
end-class

event Credit is
  class-sort Credit
  sort
    acct, amnt
  events
    Credit: Credit, acct, amnt -> Credit
end-event

event Debit is
  class-sort Debit
  sort
    acct, amnt
  events
    Debit: Debit, acct, amnt -> Debit
end-event

event Close is
  class-sort Close
  sort
    acct
  events
    Close: Close, acct -> Close
end-event

event WriteCheck is
  class-sort WriteCheck
  sort
    acct, amnt
  events
    WriteCheck: WriteCheck, acct, amnt -> WriteCheck
end-event

event SetRate is
  class-sort SetRate
  sort
    acct, rate
  events
    SetRate: SetRate, acct, rate -> SetRate
end-event

event ComputeInterest is
  class-sort ComputeInterest
```

```
  sort
    acct
  events
    ComputeInterest: ComputeInterest, acct -> ComputeInterest
end-event

event ArchCredit is
  class-sort ArchCredit
  sort
    Archive-obj, acct, amnt, date
  events
    ArchCredit: ArchCredit, Archive-obj, acct, amnt, date -> ArchCredit
end-event

event ArchDebit is
  class-sort ArchDebit
  sort
    Archive-obj, acct, amnt, date
  events
    ArchDebit: ArchDebit, Archive-obj, acct, amnt, date -> ArchDebit
end-event

event ArchClose is
  class-sort ArchClose
  sort
    Archive-obj, acct, date
  events
    ArchClose: ArchClose, Archive-obj, acct, date -> ArchClose
end-event
```

Appendix E. ULARCH *for Pump Domain Example*

This appendix contains the ULARCH traits and state transition tables for the *Pump* example. The traits are based on the *Pump* object model described in Section 3.4.3.

```
\documentstyle[fullpage,larch]{article}
\begin{document}

\begin{spec} %ObjectTheory
PumpController: trait
  includes Integer
  introduces
    pumpId: PC -> Int
\end{spec}\\

\begin{spec} %ObjectTheory
Display: trait
  includes Integer
  introduces
    cost: D -> Int
    volume: D -> Int
    grade: D -> Int
    vol-inc: D -> Int
\end{spec}\\

\begin{spec} %StateTheory
ZeroDisplay: trait
  includes Display
  introduces ZeroDisplayState: D -> Boolean
  asserts \forall d: D
    cost(d) = 0;
    volume(d) = 0;
    grade(d) = 0
\end{spec}\\

\begin{spec} %StateTheory
IncrementDisplay: trait
  includes Display
  introduces IncrementDisplayState: D -> Boolean
  asserts \forall d: D
    cost(d) >= 0;
    volume(d) >= 0;
    grade(d) >= 0
\end{spec}\\

\begin{spec} %EventTheory
NewDisplay: trait
```

```
  includes Display
  introduces new-display : -> Bool
  asserts \forall d: D
    cost(new-display) = 0;
    volume(new-display) = 0;
    grade(new-display) = 0;
    vol-inc(new-display) = 0
\end{spec}\\

\begin{spec} %EventTheory
Pulse: trait
  includes Display
  introduces pulse : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ResetDisplay: trait
  includes Display
  introduces reset-display : -> Bool
\end{spec}\\

\begin{spec} %FunctionalTheory
UpdateCost: trait
  includes Display
  introduces update-cost: D, Int, Int -> D
  asserts \forall  d:D, p, c: Int
    cost(update-cost(d, c, p)) = (c + p)
\end{spec}\\

\begin{spec} %FunctionalTheory
UpdateVolume: trait
  includes Display
  introduces update-volume: D, Int -> D
  asserts \forall d: D, v: Int
    volume(update-volume(d, v)) = (v + vol-inc(d))
\end{spec}\\

\begin{spec} %ObjectTheory
Gun(G): trait
  includes Integer
\end{spec}\\

\begin{spec} %StateTheory
GunDisabled: trait
  includes Gun
  introduces GunDisabledState: G -> Boolean
\end{spec}\\

\begin{spec} %StateTheory
GunEnabled: trait
  includes Gun
```

```
  introduces GunEnabledState: G -> Boolean
\end{spec}\\

\begin{spec} %StateTheory
GunOn: trait
  includes Gun
  introduces GunOnState: G -> Boolean
\end{spec}\\

\begin{spec} %EventTheory
NewGun: trait
  includes Gun
  introduces new-gun : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
OverHeat: trait
  includes Gun
  introduces over-heat: -> Boolean
\end{spec}\\

\begin{spec} %EventTheory
RemoveGun: trait
  includes Gun
  introduces remove-gun : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ReplaceGun: trait
  includes Gun
  introduces replace-gun : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
DepressTrigger: trait
  includes Gun
  introduces depress-trigger : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ReleaseTrigger: trait
  includes Gun
  introduces release-trigger : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
CutOffSupply: trait
  includes Gun
  introduces cutOff-supply : -> Bool
\end{spec}\\
```

```
\begin{spec} %ObjectTheory
Holster(H): trait
  includes Integer
\end{spec}\\

\begin{spec} %StateTheory
HolsterWait: trait
  includes Holster
  introduces HolsterWaitState: H -> Boolean
\end{spec}\\

\begin{spec} %StateTheory
HolsterWorking: trait
  includes Holster
  introduces HolsterWorkingState: H -> Boolean
\end{spec}\\

\begin{spec} %EventTheory
NewHolster: trait
  includes Holster
  introduces new-holster : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
ReleaseHolsterSwitch: trait
  includes Holster
  introduces release-holster-switch : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
CloseHolsterSwitch: trait
  includes Holster
  introduces close-holster-switch : -> Bool
\end{spec}\\

\begin{spec} %ObjectTheory
Motor: trait
\end{spec}\\

\begin{spec} %StateTheory
MotorDisabled: trait
  includes Motor
  introduces MotorDisabledState: M -> Boolean
\end{spec}\\

\begin{spec} %StateTheory
MotorRunning: trait
  includes Motor
  introduces MotorRunningState: M -> Boolean
\end{spec}\\
```

```
\begin{spec} %EventTheory
NewMotor: trait
  includes Motor
  introduces new-motor : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
StartPumpMotor: trait
  includes Motor
  introduces start-pump-motor : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
StopMotor: trait
  includes Motor
  introduces stop-motor : -> Bool
\end{spec}\\

\begin{spec} %ObjectTheory
Clutch: trait
  includes Integer
\end{spec}\\

\begin{spec} %StateTheory
ClutchDisabled: trait
  includes Clutch
  introduces ClutchDisabledState: C -> Boolean
\end{spec}\\

\begin{spec} %StateTheory
ClutchFree: trait
  includes Clutch
  introduces ClutchFreeState: C -> Boolean
\end{spec}\\

\begin{spec} %StateTheory
ClutchEngaged: trait
  includes Clutch
  introduces ClutchEngagedState: C -> Boolean
\end{spec}\\

\begin{spec} %EventTheory
NewClutch: trait
  includes Clutch
  introduces new-clutch : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
OverHeat: trait
  includes Clutch
  introduces over-heat: -> Boolean
```

```
\end{spec}\\

\begin{spec} %EventTheory
FreeClutch: trait
  includes Clutch
  introduces free-clutch : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
DisableClutch: trait
  includes Clutch
  introduces disable-clutch : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
EngageClutch: trait
  includes Clutch
  introduces engage-clutch : -> Bool
\end{spec}\\

\begin{spec} %ObjectTheory
ClutchMotorAssembly(CMA): trait
  includes Motor, Clutch
  ClutchMotorAssem tuple of motor : M,
                            clutch : C
\end{spec}\\

\begin{spec} %LinkTheory
KI: trait
  includes Gun, Holster
  introduces
    a-gun: KI-Link -> G
    a-holster: KI-Link -> H
    new-KI-link: G, H -> KI-Link
  asserts \forall g: G, h: H
    gun-obj(new-KI-link(g, h)) = g;
    holster-obj(new-KI-link(g, h)) = h
\end{spec}\\

\begin{spec} %AssociationTheory
Kept-In: trait
  includes Set(Kept-In for C, KI for E), KI
  introduces
    new-Kept-In: Kpt-In, G, H -> Kpt-In
    image: Kpt-In, G -> HolsterSet
    image: Kpt-In, H -> GunSet
    is-kept-in: Kpt-In, G, H -> boolean
  asserts \forall k: Kpt-In, g: G, h: H, x: KI-Link
    size(image(k, g)) = 1;
    size(image(k, h)) = 1;
    (in(x, k) \and (a-gun(x) = g)) == in(a-holster(x), image(k, g));
```

```
    (in(x, k) \and (a-holster(x) = h)) == in(a-gun(x), image(k, h));
    new-kept-in = empty-set;
    is-kept-in(new-kept-in, g, h) = false;
    is-kept-in(k, g, h) == (in (g, image(k, h)) \and in (h, image(k, g)))
\end{spec}\\

\begin{spec} %ObjectTheory
GunHolsterAssembly(GHA): trait
  includes Gun, Holster
  GunHolstAssem tuple of gun : G,
                         holster : H
\end{spec}\\

\begin{spec} %ObjectTheory
SophisticatedPump: trait
  includes Pump(SP for P), Integer
  introduces
    volumeSelect: SP -> Int
    amountSelect: SP -> Int
\end{spec}\\

\begin{spec} %ObjectTheory
Pump(P): trait
  includes
    Set(DisplaySet for C, Display for E),
    Set(GHASet for C, GunHolsterAssembly for E),
    Set(CMASet for C, ClutchMotorAssembly for E),
    PumpController, Kept-In
  P tuple of gun-holster-assembly : GHASet,
             clutch-motor-assembly : CMASet,
             pump-controller  : PC,
             display   : DisplaySet,
             kept-in: Kpt-In
  asserts \forall p: P
    size(display(p)) >= 1;
    size(gun-holster-assembly(p)) >= 1;
    size(clutch-motor-assembly(p)) >= 1
\end{spec}\\

\begin{spec} %StateTheory
PumpDisabled: trait
  includes Pump
  introduces PumpDisabledState: P -> Bool
\end{spec}\\

\begin{spec} %StateTheory
PumpEnabled: trait
  includes Pump
  introduces PumpEnabledState: P -> Bool
\end{spec}\\
```

```
\begin{spec} %EventTheory
NewPump: trait
  includes Pump
  introduces new-pump : -> Bool
\end{spec}\\

\begin{spec} %EventTheory
OverHeat: trait
  includes Pump
  introduces over-heat: -> Bool
\end{spec}\\

\begin{spec} %EventTheory
EnablePump: trait
  includes Pump
  introduces enable-pump: -> Bool
\end{spec}\\

\begin{spec} %EventTheory
DisablePump: trait
  includes Pump
  introduces disable-pump: -> Bool
\end{spec}

\end{document}
```

Table E.1 Clutch State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
ClutchInitialState	DisableClutch			ClutchInitialState		
ClutchInitialState	EngageClutch			ClutchInitialState		
ClutchInitialState	FreeClutch			ClutchInitialState		
ClutchInitialState	NewClutch			ClutchDisabled		
ClutchInitialState	OverHeat			ClutchInitialState		
ClutchDisabled	DisableClutch			ClutchDisabled		
ClutchDisabled	EngageClutch			ClutchDisabled		
ClutchDisabled	FreeClutch			ClutchFree		
ClutchDisabled	OverHeat			ClutchFree		
ClutchFree	DisableClutch			ClutchDisabled		
ClutchFree	EngageClutch			ClutchDisabled	start-fuel	
ClutchFree	FreeClutch			ClutchFree		
ClutchFree	OverHeat			ClutchFree		
ClutchEngaged	DisableClutch			ClutchEngaged		
ClutchEngaged	EngageClutch			ClutchEngaged		
ClutchEngaged	FreeClutch			ClutchFree		
ClutchEngaged	OverHeat			ClutchFree		

Table E.2 Display State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
DisplayInitialState	NewDisplay			ZeroDisplay		
DisplayInitialState	Pulse			DisplayInitialState		
DisplayInitialState	ResetDisplay			DisplayInitialState		
ZeroDisplay	Pulse			IncrementDisplay	updateCost, updateVolume	
ZeroDisplay	ResetDisplay			ZeroDisplay		
IncrementDisplay	Pulse			IncrementDisplay	updateCost, updateVolume	
IncrementDisplay	ResetDisplay			ZeroDisplay		

Table E.3 Gun State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
GunInitialState	NewGun			GunDisabled		
GunInitialState	RemoveGun			GunInitialState		
GunInitialState	ReplaceGun			GunInitialState		
GunInitialState	DepressTrigger			GunInitialState		
GunInitialState	ReleaseTrigger			GunInitialState		
GunInitialState	CutOffSupply			GunInitialState		
GunInitialState	OverHeat			GunInitialState		
GunDisabled	RemoveGun			GunEnabled		ReleaseHolsterSwitch
GunDisabled	ReplaceGun			GunDisabled		
GunDisabled	DepressTrigger			GunDisabled		
GunDisabled	ReleaseTrigger			GunDisabled		
GunDisabled	CutOffSupply			GunDisabled		
GunDisabled	OverHeat			GunDisabled		
GunEnabled	RemoveGun			GunEnabled		
GunEnabled	ReplaceGun			GunDisabled	start-timer	CloseHolsterSwitch
GunEnabled	DepressTrigger			GunOn		EngageClutch
GunEnabled	ReleaseTrigger			GunEnabled		
GunEnabled	CutOffSupply			GunEnabled		
GunEnabled	OverHeat			Disabled		
GunOn	RemoveGun			GunOn		
GunOn	ReplaceGun			GunOn		
GunOn	DepressTrigger			GunOn		
GunOn	ReleaseTrigger			GunEnabled		FreeClutch
GunOn	CutOffSupply			GunEnabled		FreeClutch
GunOn	OverHeat			GunDisabled		

Table E.4 Holster State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
HolsterInitialState	NewHolster			HolsterWait		
HolsterInitialState	ReleaseHolsterSwitch			HolsterInitialState		
HolsterInitialState	CloseHolsterSwitch			HolsterInitialState		
HolsterWait	ReleaseHolsterSwitch			HolsterWorking		
HolsterWait	CloseHolsterSwitch			HolsterWait		
HolsterWorking	CloseHolsterSwitch			HolsterWait		
HolsterWorking	ReleaseHolsterSwitch			HolsterWorking		

Table E.5 Motor State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
MotorInitialState	NewMotor			MotorDisabled		
MotorInitialState	StartPumpMotor			MotorInitialState		
MotorInitialState	StopMotor			MotorInitialState		
MotorDisabled	StartPumpMotor			MotorRunning		FreeClutch
MotorDisabled	StopMotor			MotorDisabled		
MotorRunning	StopMotor			MotorDisabled		DisableClutch
MotorRunning	StartPumpMotor			MotorRunning		
MotorRunning			$temp > 300$	MotorDisabled		OverHeat

Table E.6 Pump State Transition Table

Current State	Receive Event	Parameters	Guard	Next State	Action	Send Event
PumpInitialState	NewPump			PumpDisabled		
PumpInitialState	EnablePump			PumpInitialState		
PumpInitialState	DisablePump			PumpInitialState		
PumpInitialState	OverHeat			PumpInitialState		
PumpDisabled	EnablePump			PumpEnabled	updatePump	StartPumpMotor; ResetDisplay
PumpDisabled	DisablePump			PumpDisabled		
PumpDisabled	OverHeat			PumpDisabled		
PumpEnabled	DisablePump			PumpDisabled		
PumpEnabled	EnablePump			PumpEnabled		
PumpEnabled	OverHeat			PumpDisabled		

Appendix F. O-SLANG for Pump Domain Example

This appendix contains the O-SLANG specifications for the *Pump* example that were automatically generated from the ULARCH traits and statetransition tables in Appendix E.

```
class PumpController is
  class-sort PumpController
  sort
    PC
  sort-axioms
    PumpController = PC
  ops
    attr-equal: PumpController, PumpController -> boolean
  attributes
    pumpId: PC -> Integer
  methods
    create-PumpController: -> PumpController
  events
    new-PumpController: -> PumpController
  axioms
    attr-equal ( new-PumpController, create-PumpController);
    attr-equal ( PumpController-1, PumpController-2) =>
      (pumpId ( PumpController-1) = pumpId ( PumpController-2));
    pumpId ( create-PumpController) = 0
end-class

class PumpController-Class is
  class-sort PumpController-Class
  contained-class PumpController
  methods
    create-PumpController-Class: -> PumpController-Class
  events
    new-PumpController-Class: -> PumpController-Class
  axioms
    create-PumpController-Class = empty-set;
    new-PumpController-Class = create-PumpController-Class
end-class

class Display is
  class-sort Display
  sort
    D, Display-State
  sort-axioms
    Display = D
  ops
    attr-equal: Display, Display -> boolean
  attributes
    vol-inc: D -> Integer
    grade: D -> Integer
    volume: D -> Integer
    cost: D -> Integer
  state-attributes
    DisplayState: Display -> Display-State
  methods
    create-Display: -> Display
```

```
    update-cost: D, Integer, Integer -> D
    update-volume: D, Integer -> D
  states
    ZeroDisplay: -> Display-State
    IncrementDisplay: -> Display-State
  events
    new-Display: -> Display
    Pulse: Display -> Display
    ResetDisplay: Display -> Display
  axioms
    DisplayState ( d) = ZeroDisplay => (cost ( d) = 0 & volume ( d) = 0 & grade ( d) = 0);
    DisplayState ( d) = IncrementDisplay => (cost ( d) >= 0 & volume ( d) >= 0 & grade ( d) >= 0);
    (DisplayState ( Display-31) = ZeroDisplay) =>
      (DisplayState ( Pulse ( Display-31)) = IncrementDisplay &
       attr-equal ( Pulse ( Display-31), updateCost ( Display-31)) &
       attr-equal ( Pulse ( Display-31), updateVolume ( Display-31)));
    (DisplayState ( Display-32) = ZeroDisplay) =>
      (DisplayState ( ResetDisplay ( Display-32)) = ZeroDisplay);
    (DisplayState ( Display-33) = IncrementDisplay) =>
      (DisplayState ( Pulse ( Display-33)) = IncrementDisplay &
       attr-equal ( Pulse ( Display-33), updateCost ( Display-33)) &
       attr-equal ( Pulse ( Display-33), updateVolume ( Display-33)));
    (DisplayState ( Display-34) = IncrementDisplay) =>
      (DisplayState ( ResetDisplay ( Display-34)) = ZeroDisplay);
    DisplayState ( new-Display) = ZeroDisplay & attr-equal ( new-Display, create-Display);
    attr-equal ( Display-36, Display-37) =>
      (vol-inc ( Display-36) = vol-inc ( Display-37) &
       grade ( Display-36) = grade ( Display-37) &
       volume ( Display-36) = volume ( Display-37) &
       cost ( Display-36) = cost ( Display-37));
    ZeroDisplay <> IncrementDisplay;
    vol-inc ( create-Display) = 0;
    grade ( create-Display) = 0;
    volume ( create-Display) = 0;
    cost ( create-Display) = 0;
    vol-inc ( update-cost ( Display-38, Int-1, Int-2)) = vol-inc ( Display-38);
    grade ( update-cost ( Display-39, Int-3, Int-4)) = grade ( Display-39);
    volume ( update-cost ( Display-40, Int-5, Int-6)) = volume ( Display-40);
    cost ( update-cost ( d, c, p)) = (c + p);
    vol-inc ( update-volume ( Display-41, Int-7)) = vol-inc ( Display-41);
    grade ( update-volume ( Display-42, Int-8)) = grade ( Display-42);
    volume ( update-volume ( d, v)) = (v + vol-inc ( d));
    cost ( update-volume ( Display-43, Int-9)) = cost ( Display-43)
end-class

class Display-Class is
  class-sort Display-Class
  contained-class Display
  methods
    create-Display-Class: -> Display-Class
  events
    new-Display-Class: -> Display-Class
    Pulse: Display-Class -> Display-Class
    ResetDisplay: Display-Class -> Display-Class
  axioms
    create-Display-Class = empty-set;
    new-Display-Class = create-Display-Class;
    in ( Display-28, Display-Class-7) <=>
      in ( Pulse ( Display-28), Pulse ( Display-Class-7));
    in ( Display-29, Display-Class-8) <=>
      in ( ResetDisplay ( Display-29), ResetDisplay ( Display-Class-8))
end-class

class Gun is
  class-sort Gun
```

```
sort
  Gun-State, Clutch, Holster
sort-axioms
  Gun = G
ops
  attr-equal: Gun, Gun -> boolean
attributes
  Clutch-obj: Gun -> Clutch
  Holster-obj: Gun -> Holster
state-attributes
  GunState: Gun -> Gun-State
methods
  create-Gun: -> Gun
states
  GunDisabled: -> Gun-State
  GunEnabled: -> Gun-State
  GunOn: -> Gun-State
events
  new-Gun: -> Gun
  OverHeat: Gun -> Gun
  RemoveGun: Gun -> Gun
  ReplaceGun: Gun -> Gun
  DepressTrigger: Gun -> Gun
  ReleaseTrigger: Gun -> Gun
  CutOffSupply: Gun -> Gun
  ReleaseHolsterSwitch: Holster -> Holster
  CloseHolsterSwitch: Holster -> Holster
  EngageClutch: Clutch -> Clutch
  FreeClutch: Clutch -> Clutch
axioms
  (GunState ( Gun-89) = GunDisabled) =>
    (GunState ( RemoveGun ( Gun-89)) = GunEnabled &
     (Holster-obj ( RemoveGun ( Gun-89)) = ReleaseHolsterSwitch ( Holster-obj ( Gun-89))));
  (GunState ( Gun-90) = GunDisabled) => (GunState ( ReplaceGun ( Gun-90)) = GunDisabled);
  (GunState ( Gun-91) = GunDisabled) => (GunState ( DepressTrigger ( Gun-91)) = GunDisabled);
  (GunState ( Gun-92) = GunDisabled) => (GunState ( ReleaseTrigger ( Gun-92)) = GunDisabled);
  (GunState ( Gun-93) = GunDisabled) => (GunState ( CutOffSupply ( Gun-93)) = GunDisabled);
  (GunState ( Gun-94) = GunDisabled) => (GunState ( OverHeat ( Gun-94)) = GunDisabled);
  (GunState ( Gun-95) = GunEnabled) => (GunState ( RemoveGun ( Gun-95)) = GunEnabled);
  (GunState ( Gun-96) = GunEnabled) =>
    (GunState ( ReplaceGun ( Gun-96)) = GunDisabled &
     attr-equal ( ReplaceGun ( Gun-96), start-timer ( Gun-96)) &
     (Holster-obj ( ReplaceGun ( Gun-96)) = CloseHolsterSwitch ( Holster-obj ( Gun-96))));
  (GunState ( Gun-97) = GunEnabled) =>
    (GunState ( DepressTrigger ( Gun-97)) = GunOn &
     (Clutch-obj ( DepressTrigger ( Gun-97)) = EngageClutch ( Clutch-obj ( Gun-97))));
  (GunState ( Gun-98) = GunEnabled) => (GunState ( ReleaseTrigger ( Gun-98)) = GunEnabled);
  (GunState ( Gun-99) = GunEnabled) => (GunState ( CutOffSupply ( Gun-99)) = GunEnabled);
  (GunState ( Gun-100) = GunEnabled) => (GunState ( OverHeat ( Gun-100)) = Disabled);
  (GunState ( Gun-101) = GunOn) => (GunState ( RemoveGun ( Gun-101)) = GunOn);
  (GunState ( Gun-102) = GunOn) => (GunState ( ReplaceGun ( Gun-102)) = GunOn);
  (GunState ( Gun-103) = GunOn) => (GunState ( DepressTrigger ( Gun-103)) = GunOn);
  (GunState ( Gun-104) = GunOn) =>
    (GunState ( ReleaseTrigger ( Gun-104)) = GunEnabled &
     (Clutch-obj ( ReleaseTrigger ( Gun-104)) = FreeClutch ( Clutch-obj ( Gun-104))));
  (GunState ( Gun-105) = GunOn) =>
    (GunState ( CutOffSupply ( Gun-105)) = GunEnabled &
      (Clutch-obj ( CutOffSupply ( Gun-105)) = FreeClutch ( Clutch-obj ( Gun-105))));
  (GunState ( Gun-106) = GunOn) => (GunState ( OverHeat ( Gun-106)) = GunDisabled);
  GunState ( new-Gun) = GunDisabled & attr-equal ( new-Gun, create-Gun);
  attr-equal ( Gun-108, Gun-109) =>
    (Clutch-obj ( Gun-108) = Clutch-obj ( Gun-109) &
     Holster-obj ( Gun-108) = Holster-obj ( Gun-109));
  GunDisabled <> GunEnabled;
  GunDisabled <> GunOn;
```

```
    GunEnabled <> GunOn;
    Clutch-obj ( create-Gun) = UNDEFINED;
    Holster-obj ( create-Gun) = UNDEFINED
end-class

class Gun-Class is
  class-sort Gun-Class
  contained-class Gun
  methods
    create-Gun-Class: -> Gun-Class
  events
    new-Gun-Class: -> Gun-Class
    OverHeat: Gun-Class -> Gun-Class
    RemoveGun: Gun-Class -> Gun-Class
    ReplaceGun: Gun-Class -> Gun-Class
    DepressTrigger: Gun-Class -> Gun-Class
    ReleaseTrigger: Gun-Class -> Gun-Class
    CutOffSupply: Gun-Class -> Gun-Class
  axioms
    create-Gun-Class = empty-set;
    new-Gun-Class = create-Gun-Class;
    in ( Gun-82, Gun-Class-19) <=>
      in ( OverHeat ( Gun-82), OverHeat ( Gun-Class-19));
    in ( Gun-83, Gun-Class-20) <=>
      in ( RemoveGun ( Gun-83), RemoveGun ( Gun-Class-20));
    in ( Gun-84, Gun-Class-21) <=>
      in ( ReplaceGun ( Gun-84), ReplaceGun ( Gun-Class-21));
    in ( Gun-85, Gun-Class-22) <=>
      in ( DepressTrigger ( Gun-85), DepressTrigger ( Gun-Class-22));
    in ( Gun-86, Gun-Class-23) <=>
      in ( ReleaseTrigger ( Gun-86), ReleaseTrigger ( Gun-Class-23));
    in ( Gun-87, Gun-Class-24) <=>
      in ( CutOffSupply ( Gun-87), CutOffSupply ( Gun-Class-24))
end-class

class Holster is
  class-sort Holster
  sort
    Holster-State
  sort-axioms
    Holster = H
  state-attributes
    HolsterState: Holster -> Holster-State
  methods
    create-Holster: -> Holster
  states
    HolsterWait: -> Holster-State
    HolsterWorking: -> Holster-State
  events
    new-Holster: -> Holster
    ReleaseHolsterSwitch: Holster -> Holster
    CloseHolsterSwitch: Holster -> Holster
  axioms
    (HolsterState ( Holster-31) = HolsterWait) =>
      (HolsterState ( ReleaseHolsterSwitch ( Holster-31)) = HolsterWorking);
    (HolsterState ( Holster-32) = HolsterWait) =>
      (HolsterState ( CloseHolsterSwitch ( Holster-32)) = HolsterWait);
    (HolsterState ( Holster-33) = HolsterWorking) =>
      (HolsterState ( CloseHolsterSwitch ( Holster-33)) = HolsterWait);
    (HolsterState ( Holster-34) = HolsterWorking) =>
      (HolsterState ( ReleaseHolsterSwitch ( Holster-34)) = HolsterWorking);
    HolsterState ( new-Holster) = HolsterWait & (new-Holster = create-Holster);
    HolsterWait <> HolsterWorking
end-class
```

```
class Holster-Class is
  class-sort Holster-Class
  contained-class Holster
  methods
    create-Holster-Class: -> Holster-Class
  events
    new-Holster-Class: -> Holster-Class
    ReleaseHolsterSwitch: Holster-Class -> Holster-Class
    CloseHolsterSwitch: Holster-Class -> Holster-Class
  axioms
    create-Holster-Class = empty-set;
    new-Holster-Class = create-Holster-Class;
    in ( Holster-28, Holster-Class-7) <=>
      in ( ReleaseHolsterSwitch ( Holster-28),
           ReleaseHolsterSwitch ( Holster-Class-7));
    in ( Holster-29, Holster-Class-8) <=>
      in ( CloseHolsterSwitch ( Holster-29),
           CloseHolsterSwitch ( Holster-Class-8))
end-class

class Motor is
  class-sort Motor
  sort
    Motor-State, OverHeat, Clutch
  sort-axioms
    Motor = M
  ops
    attr-equal: Motor, Motor -> boolean
  attributes
    OverHeat-obj: Motor -> OverHeat Clutch-obj: Motor -> Clutch
  state-attributes
    MotorState: Motor -> Motor-State
  methods
    create-Motor: -> Motor
  states
    MotorDisabled: -> Motor-State
    MotorRunning: -> Motor-State
  events
    new-Motor: -> Motor
    StartPumpMotor: Motor -> Motor
    StopMotor: Motor -> Motor
    FreeClutch: Clutch -> Clutch
    DisableClutch: Clutch -> Clutch
    OverHeat: OverHeat -> OverHeat
  axioms
    (MotorState ( Motor-64) = MotorDisabled) =>
      (MotorState ( StartPumpMotor ( Motor-64)) = MotorRunning &
       (Clutch-obj ( StartPumpMotor ( Motor-64)) = FreeClutch ( Clutch-obj ( Motor-64))));
    (MotorState ( Motor-65) = MotorDisabled) =>
      (MotorState ( StopMotor ( Motor-65)) = MotorDisabled);
    (MotorState ( Motor-66) = MotorRunning) =>
      (MotorState ( StopMotor ( Motor-66)) = MotorDisabled &
        (Clutch-obj ( StopMotor ( Motor-66)) = DisableClutch ( Clutch-obj ( Motor-66))));
    (MotorState ( Motor-67) = MotorRunning) =>
      (MotorState ( StartPumpMotor ( Motor-67)) = MotorRunning);
    (MotorState ( Motor-68) = MotorRunning & temp > 300) =>
      (MotorState ( Motor-68) = MotorDisabled &
       (OverHeat-obj ( Motor-68) = OverHeat ( OverHeat-obj ( Motor-68))));
    MotorState ( new-Motor) = MotorDisabled & attr-equal ( new-Motor, create-Motor);
    attr-equal ( Motor-70, Motor-71) =>
      (OverHeat-obj ( Motor-70) = OverHeat-obj ( Motor-71) &
       Clutch-obj ( Motor-70) = Clutch-obj ( Motor-71));
    MotorDisabled <> MotorRunning;
    Clutch-obj ( create-Motor) = UNDEFINED
end-class
```

```
class Motor-Class is
  class-sort Motor-Class
  contained-class Motor
  methods
    create-Motor-Class: -> Motor-Class
  events
    new-Motor-Class: -> Motor-Class
    StartPumpMotor: Motor-Class -> Motor-Class
    StopMotor: Motor-Class -> Motor-Class
  axioms
    create-Motor-Class = empty-set;
    new-Motor-Class = create-Motor-Class;
    in ( Motor-61, Motor-Class-7) <=>
      in ( StartPumpMotor ( Motor-61), StartPumpMotor ( Motor-Class-7));
    in ( Motor-62, Motor-Class-8) <=>
      in ( StopMotor ( Motor-62), StopMotor ( Motor-Class-8))
end-class

class Clutch is
  class-sort Clutch
  sort
    Clutch-State
  sort-axioms
    Clutch = C
  state-attributes
    ClutchState: Clutch -> Clutch-State
  methods
    create-Clutch: -> Clutch
  states
    ClutchDisabled: -> Clutch-State
    ClutchFree: -> Clutch-State
    ClutchEngaged: -> Clutch-State
  events
    new-Clutch: -> Clutch
    OverHeat: Clutch -> Clutch
    FreeClutch: Clutch -> Clutch
    DisableClutch: Clutch -> Clutch
    EngageClutch: Clutch -> Clutch
  axioms
    (ClutchState ( Clutch-63) = ClutchDisabled) =>
      (ClutchState ( DisableClutch ( Clutch-63)) = ClutchDisabled);
    (ClutchState ( Clutch-64) = ClutchDisabled) =>
      (ClutchState ( EngageClutch ( Clutch-64)) = ClutchDisabled);
    (ClutchState ( Clutch-65) = ClutchDisabled) =>
      (ClutchState ( FreeClutch ( Clutch-65)) = ClutchFree);
    (ClutchState ( Clutch-66) = ClutchDisabled) =>
      (ClutchState ( OverHeat ( Clutch-66)) = ClutchFree);
    (ClutchState ( Clutch-67) = ClutchFree) =>
      (ClutchState ( DisableClutch ( Clutch-67)) = ClutchDisabled);
    (ClutchState ( Clutch-68) = ClutchFree) =>
      (ClutchState ( EngageClutch ( Clutch-68)) = ClutchDisabled &
       attr-equal ( EngageClutch ( Clutch-68), start-fuel ( Clutch-68)));
    (ClutchState ( Clutch-69) = ClutchFree) =>
      (ClutchState ( FreeClutch ( Clutch-69)) = ClutchFree);
    (ClutchState ( Clutch-70) = ClutchFree) =>
      (ClutchState ( OverHeat ( Clutch-70)) = ClutchFree);
    (ClutchState ( Clutch-71) = ClutchEngaged) =>
      (ClutchState ( DisableClutch ( Clutch-71)) = ClutchEngaged);
    (ClutchState ( Clutch-72) = ClutchEngaged) =>
      (ClutchState ( EngageClutch ( Clutch-72)) = ClutchEngaged);
    (ClutchState ( Clutch-73) = ClutchEngaged) =>
      (ClutchState ( FreeClutch ( Clutch-73)) = ClutchFree);
    (ClutchState ( Clutch-74) = ClutchEngaged) =>
      (ClutchState ( OverHeat ( Clutch-74)) = ClutchFree);
```

```
    ClutchState ( new-Clutch) = ClutchDisabled & (new-Clutch = create-Clutch);
    ClutchDisabled <> ClutchFree;
    ClutchDisabled <> ClutchEngaged;
    ClutchFree <> ClutchEngaged
end-class

class Clutch-Class is
  class-sort Clutch-Class
  contained-class Clutch
  methods
    create-Clutch-Class: -> Clutch-Class
  events
    new-Clutch-Class: -> Clutch-Class
    OverHeat: Clutch-Class -> Clutch-Class
    FreeClutch: Clutch-Class -> Clutch-Class
    DisableClutch: Clutch-Class -> Clutch-Class
    EngageClutch: Clutch-Class -> Clutch-Class
  axioms
    create-Clutch-Class = empty-set;
    new-Clutch-Class = create-Clutch-Class;
    in ( Clutch-58, Clutch-Class-13) <=>
      in ( OverHeat ( Clutch-58), OverHeat ( Clutch-Class-13));
    in ( Clutch-59, Clutch-Class-14) <=>
      in ( FreeClutch ( Clutch-59), FreeClutch ( Clutch-Class-14));
    in ( Clutch-60, Clutch-Class-15) <=>
      in ( DisableClutch ( Clutch-60), DisableClutch ( Clutch-Class-15));
    in ( Clutch-61, Clutch-Class-16) <=>
      in ( EngageClutch ( Clutch-61), EngageClutch ( Clutch-Class-16))
end-class

class ClutchMotorAssembly is
  class-sort ClutchMotorAssembly
  import
    Clutch, Motor, ClutchMotorAssembly-aggregate
  sort-axioms
    ClutchMotorAssembly = CMA
  ops
    attr-equal: ClutchMotorAssembly, ClutchMotorAssembly -> boolean
  attributes
    Motor-obj: ClutchMotorAssembly -> Motor
    Clutch-obj: ClutchMotorAssembly -> Clutch
  methods
    create-ClutchMotorAssembly: -> ClutchMotorAssembly
  events
    new-ClutchMotorAssembly: -> ClutchMotorAssembly
  axioms
    attr-equal ( new-ClutchMotorAssembly, create-ClutchMotorAssembly);
    attr-equal ( ClutchMotorAssembly-1, ClutchMotorAssembly-2) =>
      (Motor-obj ( ClutchMotorAssembly-1) = Motor-obj ( ClutchMotorAssembly-2) &
       Clutch-obj ( ClutchMotorAssembly-1) = Clutch-obj ( ClutchMotorAssembly-2));
    Motor-obj ( create-ClutchMotorAssembly) = new-Motor;
    Clutch-obj ( create-ClutchMotorAssembly) = new-Clutch
end-class

class ClutchMotorAssembly-Class is
  class-sort ClutchMotorAssembly-Class
  contained-class ClutchMotorAssembly
  methods
    create-ClutchMotorAssembly-Class: -> ClutchMotorAssembly-Class
  events
    new-ClutchMotorAssembly-Class: -> ClutchMotorAssembly-Class
  axioms
    create-ClutchMotorAssembly-Class = empty-set;
    new-ClutchMotorAssembly-Class = create-ClutchMotorAssembly-Class
end-class
```

```
aggregate ClutchMotorAssembly-aggregate is
  nodes
    Motor, Clutch, FreeClutch, DisableClutch, OverHeat-mult,
    OverHeat-mult: OverHeat-17, OverHeat-19: OverHeat-mult
  arcs
    FreeClutch -> Motor: {},
    FreeClutch -> Clutch: { FreeClutch -> Clutch},
    DisableClutch -> Motor: {},
    DisableClutch -> Clutch: { DisableClutch -> Clutch},
    OverHeat-17 -> Motor: {}, OverHeat-17 -> OverHeat-mult: {},
    OverHeat-19 -> Clutch: { OverHeat -> Clutch},
    OverHeat-19 -> OverHeat-mult: { OverHeat -> OBJ-11}
end-aggregate

link KI-Link is
  class-sort KI-Link
  sort
    Holster, Gun
  ops
    attr-equal: KI-Link, KI-Link -> boolean
  attributes
    a-holster: KI-Link -> Holster
    a-gun: KI-Link -> Gun
  methods
    create-KI-link: Gun, Holster -> KI-Link
  events
    new-KI-link: Gun, Holster -> KI-Link
  axioms
    attr-equal ( new-KI-Link ( a-holster-4, a-gun-4), create-KI-Link ( a-holster-4, a-gun-4));
    a-holster ( create-KI-Link ( a-holster-4, a-gun-4)) = a-holster-4;
    a-gun ( create-KI-Link ( a-holster-4, a-gun-4)) = a-gun-4;
    attr-equal ( KI-Link-7, KI-Link-8) =>
      (a-holster ( KI-Link-7) = a-holster ( KI-Link-8) &
       a-gun ( KI-Link-7) = a-gun ( KI-Link-8))
end-link

association Kept-In is
  class-sort Kept-In link-class KI-Link
  sort
    Gun, Holster, boolean, Kpt-In, GunSet, HolsterSet
  sort-axioms
    Kept-In = Kpt-In
  methods
    is-kept-in: Kpt-In, Gun, Holster -> boolean
    image: Kpt-In, Holster -> GunSet
    image: Kpt-In, Gun -> HolsterSet
    create-Kept-In: Gun, Holster -> Kept-In
  events
    new-Kept-In: Gun, Holster -> Kept-In
  axioms
    size ( image ( k, g)) = 1;
    size ( image ( k, h)) = 1;
    (in ( x, k) & (a-gun ( x) = g)) <=>
      in ( a-holster ( x), image ( k, g));
    (in ( x, k) & (a-holster ( x) = h)) <=>
      in ( a-gun ( x), image ( k, h));
    new-kept-in = empty-set;
    is-kept-in ( new-kept-in, g, h) = false;
    is-kept-in ( k, g, h) <=>
      (in ( g, image ( k, h)) & in ( h, image ( k, g)))
end-association

class GunHolsterAssembly is
  class-sort GunHolsterAssembly
```

```
  import
    Holster, Gun, GunHolsterAssembly-aggregate
  sort-axioms
    GunHolsterAssembly = GHA
  ops
    attr-equal:
        GunHolsterAssembly, GunHolsterAssembly -> boolean
  attributes
    Gun-obj: GunHolsterAssembly -> Gun
    Holster-obj: GunHolsterAssembly -> Holster
  methods
    create-GunHolsterAssembly: -> GunHolsterAssembly
  events
    new-GunHolsterAssembly: -> GunHolsterAssembly
  axioms
    attr-equal
      ( new-GunHolsterAssembly, create-GunHolsterAssembly);
    attr-equal ( GunHolsterAssembly-1, GunHolsterAssembly-2)
      => (Gun-obj ( GunHolsterAssembly-1) =
            Gun-obj ( GunHolsterAssembly-2)
            & Holster-obj ( GunHolsterAssembly-1) =
                Holster-obj ( GunHolsterAssembly-2));
    Gun-obj ( create-GunHolsterAssembly) = new-Gun;
    Holster-obj ( create-GunHolsterAssembly) = new-Holster
end-class

class GunHolsterAssembly-Class is
  class-sort GunHolsterAssembly-Class
  contained-class GunHolsterAssembly
  methods
    create-GunHolsterAssembly-Class: -> GunHolsterAssembly-Class
  events
    new-GunHolsterAssembly-Class: -> GunHolsterAssembly-Class
  axioms
    create-GunHolsterAssembly-Class = empty-set;
    new-GunHolsterAssembly-Class =
      create-GunHolsterAssembly-Class
end-class

aggregate GunHolsterAssembly-aggregate is
  nodes
    Gun, Holster, OverHeat-mult, OverHeat-20: OverHeat-mult,
    ReleaseHolsterSwitch, CloseHolsterSwitch
  arcs
    OverHeat-20 -> Gun: { OverHeat -> Gun},
    OverHeat-20 -> OverHeat-mult: { OverHeat -> OBJ-12},
    ReleaseHolsterSwitch -> Gun: {},
    ReleaseHolsterSwitch -> Holster: { ReleaseHolsterSwitch -> Holster},
    CloseHolsterSwitch -> Gun: {},
    CloseHolsterSwitch -> Holster: { CloseHolsterSwitch -> Holster}
end-aggregate

class SophisticatedPump is
  class-sort SophisticatedPump < Pump
  import
    Pump
  sort
    SP
  sort-axioms
    SophisticatedPump = SP
  ops
    attr-equal: SophisticatedPump, SophisticatedPump -> boolean
  attributes
    amountSelect: SP -> Integer volumeSelect: SP -> Integer
  axioms
```

```
    attr-equal ( SophisticatedPump-33, SophisticatedPump-34) =>
      (amountSelect ( SophisticatedPump-33) = amountSelect ( SophisticatedPump-34) &
       volumeSelect ( SophisticatedPump-33) = volumeSelect ( SophisticatedPump-34))
end-class

class SophisticatedPump-Class is
  class-sort SophisticatedPump-Class
  contained-class SophisticatedPump
  methods
    create-SophisticatedPump-Class: -> SophisticatedPump-Class
  events
    new-SophisticatedPump-Class: -> SophisticatedPump-Class
  axioms
    create-SophisticatedPump-Class = empty-set;
    new-SophisticatedPump-Class = create-SophisticatedPump-Class
end-class

class Pump is
  class-sort Pump
  import
    Kept-In, PumpController, Pump-aggregate
  sort
    Pump-State, GunHolsterAssembly-Class,
    ClutchMotorAssembly-Class, PumpController, Display-Class,
    Kept-In, Display, Motor
  sort-axioms
    Pump = P = OverHeat-18.OBJ-10;
    Motor = Motor-obj-sort;
    Display-Class = Display-obj-sort
  ops
    attr-equal: Pump, Pump -> boolean
  attributes
    GunHolsterAssembly-Class-obj: Pump -> GunHolsterAssembly-Class
    ClutchMotorAssembly-Class-obj: Pump -> ClutchMotorAssembly-Class
    PumpController-obj: Pump -> PumpController
    Display-Class-obj: Pump -> Display-Class
    Kept-In-obj: Pump -> Kept-In
    Display-obj: Pump -> Display
    Motor-obj: Pump -> Motor
  state-attributes
    PumpState: Pump -> Pump-State
  methods
    create-Pump: -> Pump
  states
    PumpDisabled: -> Pump-State PumpEnabled: -> Pump-State
  events
    new-Pump: -> Pump
    OverHeat: Pump -> Pump
    EnablePump: Pump -> Pump
    DisablePump: Pump -> Pump
    StartPumpMotor: Motor -> Motor
    ResetDisplay: Display -> Display
  axioms
    size ( Display-Class-obj ( Pump-58)) >= 1;
    size ( GunHolsterAssembly-Class-obj ( Pump-59)) >= 1;
    size ( ClutchMotorAssembly-Class-obj ( Pump-60)) >= 1;
    (PumpState ( Pump-65) = PumpDisabled) =>
      (PumpState ( EnablePump ( Pump-65)) = PumpEnabled &
       attr-equal ( EnablePump ( Pump-65), updatePump ( Pump-65)) &
       (Motor-obj ( EnablePump ( Pump-65)) = StartPumpMotor ( Motor-obj ( Pump-65))) &
       (Display-obj ( EnablePump ( Pump-65)) = ResetDisplay ( Display-obj ( Pump-65))));
    (PumpState ( Pump-66) = PumpDisabled) => (PumpState ( DisablePump ( Pump-66)) = PumpDisabled);
    (PumpState ( Pump-67) = PumpDisabled) => (PumpState ( OverHeat ( Pump-67)) = PumpDisabled);
    (PumpState ( Pump-68) = PumpEnabled) => (PumpState ( DisablePump ( Pump-68)) = PumpDisabled);
    (PumpState ( Pump-69) = PumpEnabled) => (PumpState ( EnablePump ( Pump-69)) = PumpEnabled);
```

```
    (PumpState ( Pump-70) = PumpEnabled) => (PumpState ( OverHeat ( Pump-70)) = PumpDisabled);
    PumpState ( new-Pump) = PumpDisabled & attr-equal ( new-Pump, create-Pump);
    attr-equal ( Pump-72, Pump-73) =>
      (GunHolsterAssembly-Class-obj ( Pump-72) = GunHolsterAssembly-Class-obj ( Pump-73) &
       ClutchMotorAssembly-Class-obj ( Pump-72) = ClutchMotorAssembly-Class-obj ( Pump-73) &
       PumpController-obj ( Pump-72) = PumpController-obj ( Pump-73) &
       Display-Class-obj ( Pump-72) = Display-Class-obj ( Pump-73) &
       Kept-In-obj ( Pump-72) = Kept-In-obj ( Pump-73) &
       Display-obj ( Pump-72) = Display-obj ( Pump-73) &
       Motor-obj ( Pump-72) = Motor-obj ( Pump-73));
    PumpDisabled <> PumpEnabled;
    GunHolsterAssembly-Class-obj ( create-Pump) = new-GunHolsterAssembly-Class;
    ClutchMotorAssembly-Class-obj ( create-Pump) = new-ClutchMotorAssembly-Class;
    PumpController-obj ( create-Pump) = new-PumpController;
    Display-Class-obj ( create-Pump) = new-Display-Class;
    Kept-In-obj ( create-Pump) = UNDEFINED;
    Display-obj ( create-Pump) = UNDEFINED;
    Motor-obj ( create-Pump) = UNDEFINED
end-class

class Pump-Class is
  class-sort Pump-Class
  contained-class Pump
  methods
    create-Pump-Class: -> Pump-Class
  events
    new-Pump-Class: -> Pump-Class
    OverHeat: Pump-Class -> Pump-Class
    EnablePump: Pump-Class -> Pump-Class
    DisablePump: Pump-Class -> Pump-Class
  axioms
    create-Pump-Class = empty-set;
    new-Pump-Class = create-Pump-Class;
    in ( Pump-61, Pump-Class-10) <=>
      in ( OverHeat ( Pump-61), OverHeat ( Pump-Class-10));
    in ( Pump-62, Pump-Class-11) <=>
      in ( EnablePump ( Pump-62), EnablePump ( Pump-Class-11));
    in ( Pump-63, Pump-Class-12) <=>
      in ( DisablePump ( Pump-63), DisablePump ( Pump-Class-12))
end-class

aggregate Pump-aggregate is
  nodes
    GunHolsterAssembly-Class, ClutchMotorAssembly-Class,
    PumpController, Display-Class, Kept-In, Integer,
    SET-19: Set, SET-20: Set, SET-21: Set, SET-22: Set,
    Gun-Class, SET-23: Set, Holster-Class, SET-24: Set,
    OverHeat-mult, OverHeat-18: OverHeat-mult, EngageClutch,
    FreeClutch
  arcs
    SET-19 -> GunHolsterAssembly-Class: { Set -> GunHolsterAssembly-Class, E -> GunHolsterAssembly},
    SET-20 -> ClutchMotorAssembly-Class: { Set -> ClutchMotorAssembly-Class, E -> ClutchMotorAssembly},
    SET-21 -> Display-Class: { Set -> Display-Class, E -> Display},
    Integer -> SET-19: {},
    Integer -> SET-20: {},
    Integer -> SET-21: {},
    Integer -> SET-22: {},
    SET-22 -> Kept-In: { Set -> Kept-In, E -> KI-Link},
    Integer -> SET-23: {},
    SET-23 -> Gun-Class: { Set -> Gun-Class, E -> Gun},
    SET-23 -> Kept-In: { Set -> GunSet, E -> Gun},
    Integer -> SET-24: {},
    SET-24 -> Holster-Class: { Set -> Holster-Class, E -> Holster},
    SET-24 -> Kept-In: { Set -> HolsterSet, E -> Holster},
    OverHeat-18 -> OverHeat-mult: { OverHeat -> OBJ-10},
```

```
    EngageClutch -> Gun: {},
    EngageClutch -> Clutch: { EngageClutch -> Clutch},
    FreeClutch -> Gun: {},
    FreeClutch -> Clutch: { FreeClutch -> Clutch}
end-aggregate

event ReleaseHolsterSwitch is
  class-sort ReleaseHolsterSwitch
  events
    ReleaseHolsterSwitch: ReleaseHolsterSwitch -> ReleaseHolsterSwitch
end-event

event CloseHolsterSwitch is
  class-sort CloseHolsterSwitch
  events
    CloseHolsterSwitch: CloseHolsterSwitch -> CloseHolsterSwitch
end-event

event EngageClutch is
  class-sort EngageClutch
  events
    EngageClutch: EngageClutch -> EngageClutch
end-event

event FreeClutch is
  class-sort FreeClutch
  events
    FreeClutch: FreeClutch -> FreeClutch
end-event

event DisableClutch is
  class-sort DisableClutch
  events
    DisableClutch: DisableClutch -> DisableClutch
end-event

event OverHeat is
  class-sort OverHeat
  events
    OverHeat: OverHeat -> OverHeat
end-event

event OverHeat-mult is
  class-sort OverHeat
  sort OBJ-10, OBJ-11, OBJ-12
  attributes
    OBJ-10-obj: OverHeat -> OBJ-10
    OBJ-11-obj: OverHeat -> OBJ-11
    OBJ-12-obj: OverHeat -> OBJ-12
  events
    OverHeat: OverHeat -> OverHeat
    OverHeat: OBJ-10 -> OBJ-10
    OverHeat: OBJ-11 -> OBJ-11
    OverHeat: OBJ-12 -> OBJ-12
  axioms
    OBJ-10-obj ( OverHeat ( OverHeat-16)) = OverHeat ( OBJ-10-obj ( OverHeat-16));
    OBJ-11-obj ( OverHeat ( OverHeat-16)) = OverHeat ( OBJ-11-obj ( OverHeat-16));
    OBJ-12-obj ( OverHeat ( OverHeat-16)) = OverHeat ( OBJ-12-obj ( OverHeat-16))
end-event

event StartPumpMotor is
  class-sort StartPumpMotor
  events
    StartPumpMotor: StartPumpMotor -> StartPumpMotor
end-event
```

```
event ResetDisplay is
  class-sort ResetDisplay
  events
    ResetDisplay: ResetDisplay -> ResetDisplay
end-event
```

Appendix G. User Manual for Formal Object Transformation System

G.1 Introduction

This appendix outlines the procedures used to transform ULARCH traits and state transition tables into O-SLANG. First, the Refine files which are needed for the transformation process are presented, along with their compilation and loading order. Next, the user files required to run the transformations are described. Finally, a sample run using the Pump domain example is presented.

G.2 Refine Files

This section outlines the procedures needed to initialize the transformation system.

1. Load system files for Dialect and Object Inspector.
    - (load-system "Dialect")
    - (load-system "Intervista")
2. Compile the domain model and grammar files for ULARCH, state transition tables, and O-SLANG if necessary.
    - M-x refine-compile-file ularch-dm.re
    - M-x refine-compile-file ularch-gram.re
    - M-x refine-compile-file stt-dm.re
    - M-x refine-compile-file stt-gram.re
    - M-x refine-compile-file oslang-dm.re
    - M-x refine-compile-file oslang-gram.re
3. Load the domain model and grammar files for ULARCH, state transition tables, and O-SLANG.
    - M-x refine-load-file ularch-dm.lfasls1
    - M-x refine-load-file ularch-gram.lfasls1

- M-x refine-load-file stt-dm.lfasls1
- M-x refine-load-file stt-gram.lfasls1
- M-x refine-load-file oslang-dm.lfasls1
- M-x refine-load-file oslang-gram.lfasls1

4. Compile the lisp utilities file if necessary, and then load it.
   - M-x refine-compile-file lisp-utilities.lisp
   - M-x refine-load-file lisp-utilities.fasls1

5. Compile the transformations file if necessary, and then load it.
   - M-x refine-compile-file uo-trans.re
   - M-x refine-load-file uo-trans.lfasls1

G.3 User Files

In order to transform a ULARCH domain theory into an O-SLANG domain theory, several files are needed. These files are:

- A *.lsl* file containing the ULARCH traits
- A *.stt* file for each state transition table
- A *.dm* file containing the names of the *.lsl* file and all of the *.stt* files

When the *.dm* file is constructed, the first line should contain the name of the *.lsl* file, and each line thereafter should contain the name of a *.stt* file. To run the transformations, the function *Transform-DomainModel* is invoked from the command line in Refine as follows:

.> (Transform-DomainModel "filename.dm")

When the transformations complete, a *.oslang* file will be created in the same directory as the *.dm* file.

G.4 Sample Session

This section shows a sample run using the Pump domain example. To transform the ULARCH and state transition table files for pump, the Refine files needed were first loaded using the steps outlined in Section G.2. Next, the user files were set up as described in Section G.3. The *.lsl* and *.stt* files appear in appendix E. The file *pump.dm* is shown below.

```
pump/pump.lsl
pump/clutch.stt
pump/display.stt
pump/gun.stt
pump/holster.stt
pump/motor.stt
pump/pump.stt
```

Once the necessary files were set up the transformations were run as follows:

```
.> (Transform-DomainModel "pump/pump.dm")
Parsing "pump/pump.lsl"...Succeeded!
Transforming Ularch to O-Slang...Rule successfully applied.

Parsing "pump/clutch.stt"...Succeeded!
Transforming "pump/clutch.stt" to O-Slang...Rule successfully applied.

Parsing "pump/display.stt"...Succeeded!
Transforming "pump/display.stt" to O-Slang...Rule successfully applied.

Parsing "pump/gun.stt"...Succeeded!
Transforming "pump/gun.stt" to O-Slang...Rule successfully applied.

Parsing "pump/holster.stt"...Succeeded!
Transforming "pump/holster.stt" to O-Slang...Rule successfully applied.

Parsing "pump/motor.stt"...Succeeded!
Transforming "pump/motor.stt" to O-Slang...Rule successfully applied.

Parsing "pump/pump.stt"...Succeeded!
Transforming "pump/pump.stt" to O-Slang...Rule successfully applied.

Transforming "" to O-Slang...Rule successfully applied.

Performing post processing...
Updating aggregates for communication and associations...Succeeded!
Adding object-valued attributes where needed...Succeeded!
Adding 'new-' events and 'create-' events where needed...Succeeded!
Updating axioms where needed...Succeeded!
Replacing Int with Integer...Succeeded!Succeeded!
Writing "pump/pump.oslang"...

Succeeded, transformation complete!
```

Bibliography

Bai94. Paul D. Bailor. 1994 reasearch summary. Unpublished Research Summary: Knowledge-Based Software Engineering Group, Air Force Institute of Technology, 1994.

Bai95. Paul D. Bailor. Theories and Software Engineering. Class Notes: CSCE 793, Winter 1995, Air Force Institute of Technology, 1995.

BFG+94. Lee Blaine, Rafael Furst, Li-Mei Gilham, Allen Goldberg, Richard Jüllig, Jim McDonald, and Y.V. Srinivas. *Specware^{TM} User Manual*, October 1994. Specware^{TM}Version Core4.

BGG+94. Lee Blaine, Li-Mei Gilham, Allen Goldberg, Richard Jüllig, Jim McDonald, and Y.V. Srinivas. *SLANG Language Manual*, October 1994. Specware^{TM} Version Core4.

BO93. Ronald V. Book and Friedrich Otto. *String-Rewriting Systems.* Springer-Verlag, 1993.

DBH95. Maj Scott DeLoach, LtCol Paul Bailor, and Thomas Hartrum. Representing object models as theories. In *Proceedings KBSE '95: The Tenth Knowledge-Based Software Engineering Conference*, Boston, Massachusetts, November 1995. IEEE Computer Society Press Los Alamitos, California.

DeL95a. Maj Scott DeLoach. An object-oriented, theory-based, parallel successive refinement specification acquisition system. Dissertation Prospectus, Graduate School of Engineering, Air Force Institute of Technology (AU), February 1995.

DeL95b. Maj Scott DeLoach. A theory-based object model. Unpublished, July 1995.

DeL95c. Maj Scott DeLoach. Transformations from omt to a theory-based object model. Unpublished, July 1995.

Der87. Nachum Dershowitz. Termination of rewriting. In J.P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 69–115. Academic Press, 1987.

Der93. Nachum Dershowitz. A taste of rewrite systems. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, International Lecture Series 1991-1992, pages 199–228. Springer-Verlag, McMaster University, Hamilton, Ontario, Canada, 1993.

Der94. Nachum Dershowitz. Hierarchical termination. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1994.

DJ90. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Formal Methods and Semantics. Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier - The MIT Press, North Holland, Amsterdam, 1990.

GH93. John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

HB94. Thomas C. Hartrum and LtCol Paul D. Bailor. Teaching formal extensions of informal-based object-oriented analysis methodologies. In *Software Engineering Education Proceedings.* Software Engineering Institute (SEI), January 1994.

int91. Reasoning Systems Inc., 3260 Hillview Avenue, Palo Alto, CA. *INTERVISTA User's Guide*, March 1991. Version 1.0.

Klo92. J.W. Klop. *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 1–116. Clarendon Press, 1992.

Lin94. Captain Catherine J. Lin. Unification of larch and z-based object models to support algebraically-based design refinement: The larch perspective. Master's thesis, Graduate School of Engineering, Air Force Institute of Technology (AU), 1994.

Mit94. Subrata Mitra. *Semantic Unification for Convergent Systems.* PhD thesis, University of Illinois at Urbana-Champaign, 1994.

NS91. Allan Norcliffe and Gil Slater. *Mathematics of Software Construction.* Ellis Horwood Limited, Market Cross House, Cooper Street, Chichester, England, 1991.

RBP+91. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

Ref90. Reasoning Systems Inc., 3260 Hillview Avenue, Palo Alto, CA. *Refine User's Guide*, May 1990. Version 3.0.

Smi90. Douglas R. Smith. Kids: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

vBSB93. S. van Bakel, S. Smetsers, and S. Brock. Partial type assignment in left linear applicative term rewriting systems. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 2, pages 15–29. John Wiley and Sons Ltd, 1993.

Wab94. Captain Kathleen May Wabiszewski. Unification of larch and z-based object models to support algebraically-based design refinement: The z perspective. Master's thesis, Graduate School of Engineering, Air Force Institute of Technology (AU), 1994. AD-A289234.

Vita

Captain Charles G. Beem was born on January 29, 1963 in Bainbridge, Maryland and graduated from Roxana High School in Roxana, Illinois in 1981. He earned an Associates Degree in Applied Science in Data Processing at Lewis and Clark Community College in Godfrey, Illinois, in 1985 before enlisting in the Air Force as a programmer. He developed tactical communication software to support ground-based tactical communication. In December, 1988, he entered the Airman Education and Commissioning Program at Wright State University in Dayton, Ohio, and was awarded a Bachelor's Degree in Computer Science in June 1990. He received his commission through Officer Training School in October 1990 before being assigned as a programming team chief on Strategic Air Command's intelligence data handling system. He also served as section chief for his division's software engineering review board before leaving in May, 1994, to pursue a Master of Science degree in Computer Science at the Air Force Institute of Technology at Wright-Patterson AFB, Ohio. Upon graduation, Captain Beem will be assigned to the Air Force C4 Agency at Scott AFB, Illinois, where he will work software engineering policy issues for the Air Force.

Permanent address: 216 Sheraton Dr
Belleville, Il 62223

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1995	Master's Thesis

4. TITLE AND SUBTITLE
Transforming Algebraically-Based Object Models Into a Canonical Form for Design Refinement

5. FUNDING NUMBERS

6. AUTHOR(S)
Charles G. Beem, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/95D-01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Mr. Glenn Durbin
NSA/Y23
9800 Savage Road
Fort Meade, MD 20755-6000

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)*
The understandability of object-oriented design techniques and the rigor of formal methods have improved the state of software development; however, both ideas have limitations. Object-oriented techniques, which are semi-formal, can still result in incorrect designs, while formal methods are complex and require an extensive mathematical background. The two approaches can be coupled, however, to produce designs that are both understandable and verifiable, and to produce executable code. This research proposes an approach where object-oriented models are first represented algebraically in a formal specification language such as LARCH and then transformed into a canonical form suitable for design refinement.
In the canonical form presented in this work, object-oriented models are represented as domain theories consisting of multiple class specifications. Each class specification has sorts, operations (attributes, methods, events, states, state attributes, and operations), and axioms which describe its structure and behavior. The ability to reason about relationships between specifications is handled through the use of category theory operations. Although the canonical form is methodology independent, this work demonstrates the proposed approach on object-oriented models developed using Rumbaugh's Object Modeling Technique. The models are first mapped to LARCH and then translated into the canonical form by a set of rewrite rules. The rewrite rules are shown to produce unique normal forms. The final product is a transformation system which converts object-oriented designs into a canonical form that can be used with a design refinement tool.

14. SUBJECT TERMS
software engineering, specifications, formal specification languages, Larch specification language, algebraic specification languages, object-oriented models, term rewriting

15. NUMBER OF PAGES
181

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to *stay within the lines* to meet *optical scanning requirements*.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	PR	- Project
G	- Grant	TA	- Task
PE	- Program Element	WU	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.