ENFORCING ONTOLOGICAL RULES IN CONCEPTUAL MODELING USING UML: PRINCIPLES AND IMPLEMENTATION

SHAN LU

ENFORCING ONTOLOGICAL RULES IN CONCEPTUAL MODELING USING UML: PRINCIPLES AND IMPLEMENTATION

by

© Shan Lu

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
(Master of Science)

Department of Computer Science
Memorial University of Newfoundland

August 2005

St.John's Newfoundland

Abstract

UML is very popular in software engineering and is used for at least two purposes: OO software design and conceptual modeling. However, UML's origins in software engineering may limit its appropriateness for conceptual modeling. Evermann and Wand (2003, 2002, 2001) point out that conceptual modeling involves representing the real world, and that ontology is the branch of philosophy dealing with that. They developed a set of ontological rules placing constraints on the construction of UML diagrams, to ensure that they properly represent underlying ontological assumptions. However, no existing UML-based CASE tools enforce such rules. The purpose of this research is to implement such functionality in an UML-based CASE tool to guide the modeling process. Also, this research develops better understandings of the rules by considering how they can be implemented. Our implementation has built-in 'intelligence' to detect and explain the nature of violations.

Acknowledgments

This study is the result of the wise counsel and the generous contributions of a number of persons.

To Dr. Jeffrey Parsons, my supervisor, my sincere gratitude for his advice and assistance. His criticisms, suggestions and confidence have guided my efforts in this study from their formative stages to the concluding statements. Also my thanks to him for helping to clarify problems of the on-going study.

I would like also to express my appreciation and gratitude to faculties and staffs of the Department of Computer Science for their support and cooperation during the courses and research of the program.

Finally, to my parents, Jialin Lu and Mengling Li, my wife, Xuting Chen heartfelt gratitude, for their support and encouragement, which is vitally necessary to my study.

Table of Contents

List of Tables

List of Figures

List of Appendices

Chapter 1 Introduction

1.1 Background

1.1.1 Popularity of UML

As time goes on, software systems are becoming increasingly complex. Thus, modern software development is increasingly challenging. In addition, a high proportion of projects fail. Under these circumstances, a visual language for software engineering, the Unified Modeling Language (UML) has emerged, to help people develop complex software. Since 1997, when UML was adopted as the standard for software engineering by Object Management Group (OMG), many organizations have begun using UML. Today, there are many books as well as papers about UML published every year; there are many universities introducing UML in their software engineering courses; there are many software development companies hiring employees with a UML background. UML is very popular, and has de facto become a standard for software engineering. The emergence of UML is very important for future software engineering.

1.1.2 Purposes of UML

What is UML? The formal description of UML by Object Management Group (OMG)

is:

> The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas and reusable software components (OMG, 2003, p.25, p.45).

UML is used for at least two purposes:

1.1.2.1 Object-Oriented software design

The original purpose of UML is for Object-Oriented (OO) software design. When people develop software systems, UML is used to describe system requirements, control flow, data flow and so on. UML has several types of diagrams to describe software systems, including Usecase (diagram) to describe system from the user perspective; class diagram to describe static classes; sequence diagram and collaboration diagram to describe the process of information passing among objects. The diagrams and their functions will be listed in Section 2.1.

1.1.2.2 Conceptual modeling

Another purpose of UML is conceptual modeling of a domain for which a system is to be required. This is for the analysis step in software engineering. UML diagrams are used to describe business modeling, or even non-software systems. UML is utilized as a tool for communicating between users and developers in understanding and eliciting requirements, and also for documentation purposes.

> The Unified Modeling Language was formed by integrating several diagramming techniques for the purpose of software specification, design, construction and maintenance. It would be advantageous to use the same modeling method throughout the development process of an information system, namely, to extend the use of UML to conceptual modeling (Evermann &Wand, 2001, p.1).

1.2. Problems

1.2.1 Limitations of UML

UML works relatively well for the first purpose: OO software design. However, it may lead problems when used for the second purpose: Conceptual Modeling, because this was not the original purpose of UML.

The suitability of UML for modeling concrete problem domains in the early development phases has been called into question. The applicability of object-oriented modeling in general in the early development phases is also controversial (Opdahl and Henderson-Sellers, 2002, p.1).

The following example demonstrates how problems may occur. If a group develops a POS (Point of Sale) system there will be an analyst doing the system analysis. He may model the 'Order' as a class and however does not include any attribute of the class. When other programmers implement the project in the implementation stage, they will set up the database table, according to the classes created in the system analysis and design stages. Because there is no attribute in the 'Order' class, they will not setup a table for the 'Order.' Finally, the system will have a serious problem, that orders cannot be stored in the database.

Now, let us analyze the reason why these problems may occur. UML is based on Object-Oriented (OO) technology and the first emergence out of OO technology is the OO programming language. Since UML can automatically generate code, we can see how well UML concepts are mapped to the OO programming language. For example, in Java language, there are concepts of 'class' and 'method'. These are also included in the UML. If looking at the whole system development as a sequence from top to bottom, we may say that UML is developed from down to up. That is, UML is developed from the perspective of system implementation. However, the details of design and implementation are not significant in the system analysis stage. Now, people are using UML for conceptual modeling in the early analysis stage of system development. In this situation, UML's origins in software engineering may limit its appropriateness for conceptual modeling.

1.2.2 Everman and Wand's research on those limitations

Because of these limitations of UML, Evermann and Wand's research is particularly salient. Their research has analyzed and examined the suitability and highlighted the weakness of UML for domain modeling. In addition, they developed constraints (rules) to make UML more appropriate for domain modeling.

> This research goes beyond pointing out the ontological defects and instead suggests rules and guidelines for the modeler and analyst which alleviate these problems and enable the use of UML for conceptual modeling of real-world domains. (Evermann&Wand, 2003, p215)

Evermann and Wand's research points out that conceptual modeling represents the real

world. When people use UML for conceptual modeling purposes, UML is used as a tool, which describes and helps people to understand and model the real world. A tool to fulfill this job should be based on the perspective of the real world. Ontology is the branch of philosophy dealing with the nature and structure of the real world and thus is an appropriate theory for their research. Evermann and Wand have developed a set of ontological rules that place constraints on the construction of UML diagrams to ensure that they properly represent underlying ontological assumptions.

1.2.3 Existing UML-based CASE tools do not enforce such rules

Although some research has tried to help make UML more suitable for conceptual modeling, notably Evermann and Wand's ontological rules, there is no existing UML-based Computer Aided System Engineering (CASE) tools that reflect these rules. (We will review the current CASE tools in Section 2.4.)

1.3 Research Purpose

The purpose of this research is to implement Evermann and Wand's ontological rules in an UML-based CASE tool. The software will check UML diagrams and indicate if those diagrams violate one or several of these rules. The input data are diagrams drawn by a UML CASE (Computer Aided Software Engineering) tool and the output would show which rules are violated and explain how the diagram violates the rule. The system will

give examples for each rule to help users comprehend the rule. Thus, the research question is:

Are Evermann's ontological rules practical? How can we implement these rules into CASE tools?

As an additional contribution, we will also evaluate Evermann's rules according to ontology.

1.4 Structure

The remainder of this thesis is structured as follows. In the next chapter, we review the related concepts and research. This is done by reviewing UML in Section 2.1, research on using UML for conceptual modeling in Section 2.2, and Evermann and Wand's research in Section 2.3. In Section 2.3, we also discuss ontology and Evermann's rules. Next, we review existing UML-based CASE tools in Section 2.4. Since our implementation is also a critiquing expert system, the last section of Chapter 2 analyzes ArgoUML from perspective of a critiquing expert system. Chapter 3 introduces our research approach. The choosing of the CASE tool (ArgoUML) is explained in Section 3.1. Then we explain the principles we followed during our implementation. Section 3.3 explains the discussion order of Evermann's rules and corollaries in our thesis. Chapter 4 begins to introduce the main part of the thesis, the specific rules and corollaries are covered. Evermann developed the rules and corollaries through three stages: static, change and interaction. Thus the rules

and corollaries can be classified into three categories. Since some latter rules / corollaries are based on the previous ones, also for convenient and consistency reasons, our discussion follows the order of Evermann's rules / corollaries. That is, rules and corollaries related to static, change and interaction are discussed in Chapter 4, Chapter 5, and Chapter 6 respectively. For each rule and corollary, we first explain its motivation and meaning. Then our implementation approach is discussed. For those which are difficult to comprehend we also give examples. After that, we will talk about the implementation in ArgoUML in Chapter 7. In that chapter, we first introduce how the critics work in ArgoUML. Then, we explain problems in current ArgoUML. In Section 7.3, the communication structure between the system and users is described. The main section (Section 7.4) of that chapter is the alternative implementation approaches for some rules and corollaries, which cannot be actually implemented in ArgoUML using solutions we gave in Chapters 4, 5 and 6. Finally, Chapter 8 concludes the thesis including our implementation results. In Appendix A, we introduce how to setup, run and test our implementation. Appendix B includes all of our implementation source codes and test data, which is burned on a compact disk. In Appendix C, we list all Evermann's ontological rules and corollaries.

Chapter 2 Related research

In this chapter, we review other research, concepts, and technologies related to our research.

2.1 UML

UML is the abbreviation of the Unified Modeling Language. It is integrated and developed from three technologies for system modeling. They are OOAD (Object-Oriented Analysis and Design) by Booch (1994), OMT (Object Modeling Technique) by Rumbaugh (1991) and OOSE (Object-Orient Software Engineering) by Jacobson (1992). UML uses diagrams to model and thus is a graphical language. Table 2-1 lists all UML diagrams and their functions.

Diagram	Function
Class diagram	Show a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.
Object diagram	Encompasses objects and their relationships at a point in time. An object diagram may be considered a special case of a class diagram or a collaboration diagram.

Use case diagram	Show the relationships among actors and use cases within a system.
Sequence diagram	Show object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). Sequence diagrams and collaboration diagrams Express similar information, but show it in different ways.
Collaboration diagram	Show interactions organized around the structure of a model, using either classifiers and associations or instances and links. Unlike a sequence diagram, a collaboration diagram shows the relationships among the instances. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.
Statechart diagram	Show a state machine.
Activity graph	A special case of a state machine that is used to model processes involving one or more classifiers.
Component diagram	Show the organizations and dependencies among components.
Deployment diagram	Show the configuration of run-time processing nodes and the components, processes, and objects that live on them. Components represent run-time manifestations of code units.

Table 2.1 UML diagrams and their functions (OMG, 2003)

Since our research is based on Evermann and Wand's ontological rules, we only focus

on diagrams related to those rules. Our research covers the following three types of diagrams in UML.

- Diagrams related to static structure

  This category of diagrams is used to describe static structure of things. It includes class diagram, object diagram and components of class, attribute, operation object, association, binary association, association class, N-ary association, composition, link, generalization, and dependency.

- Diagrams related to change

  This category of diagrams is used to describe change within things. It includes statechart diagram, activity diagram and components of state, composite states, events, simple transitions, transitions to and from concurrent states, transitions to and from composite states, submachine states and synch states, action state, subactivity state, call states, swimlanes, and synch states.

- Diagrams related to interaction

  This category of diagrams is used to describe interaction between things. It includes sequence diagram, collaboration diagram and components of interactions, messages, and stimulus.

We assume readers of this thesis have basic knowledge of UML. Thus, we do not explain UML concepts in our thesis. Readers can reference UML 1.5 Specification (OMG, 2003) for detailed descriptions and definitions of UML components covered in this thesis.

2.2 Research on using UML for conceptual modeling

Since UML is also used for conceptual modeling of a domain for which a system is to be required, some research has analyzed and examined the suitability and pointed out the weakness of UML for domain modeling.

One important study is Opdahl and Henderson-Sellers's Ontological Evaluation of the UML Using the Bunge–Wand–Weber Model (Opdahl and Henderson-Sellers, 2002). They evaluated the UML constructs and found some weaknesses of UML for system analysis. They also pointed out that using UML for early development phases has some problems. Their research evaluated UML through BWW-Ontology (Bunge-Wand-Weber Ontology, discussed in detail in Section 2.3.2). They systematically and iteratively compared 47 BWW-ontological concepts with 216 modeling constructs in the UML, which covers concepts of Objects & Types, Properties & Attributes, Behavior, Links & Association, Aggregates & Systems, States & Transitions, Action & Interaction, Use Case & Scenario and Time. They used the approach of representation mapping (from the BWW-model to UML) as well as interpretation mapping (from UML to the BWW-model) and evaluated UML for ontological discrepancies of construct redundancy, overload, excess and deficit. Their research showed that many UML constructs conform to BWW-Ontology. However, they also found the following problems. There are eight construct redundancies, which means several UML constructs map with the same ontological concept. There are one problematic, and two less problematic construct overloads, which means one UML

construct maps with several ontological concepts. There are two construct excesses, which means there is a UML construct cannot map with any ontological concept. There are three categories of construct deficit, which means no UML construct maps with a particular ontological concept.

There is some other research that evaluated different structures in UML for conceptual modeling. Burton-Jones and Meso have tested the 'Wand and Weber Good Decomposition Model' as the basic theory of improving Object-Oriented Analysis (Burton and Meso, 2002). This research used the approach of empirical laboratory experiment and examples of class diagrams, state chart diagrams and use case diagrams. Guizzardi, Heinrich and Gerd evaluated a conceptual UML class model for ontological correctness and assigned well-defined ontological semantics to UML constructs (Guizzardi, Heinrich and Gerd, 2002[1]). Their research used the General Ontological Language (GOL) and its underlying ontology. However, they only evaluated UML structure of Classes & Objects, Powertypes, Associations & Part-whole Relations (Aggregation / Composition). Guizzardi, Herre and Wagner used the same approach to evaluate other UML structures of Datatypes, Abstract class and Association (Giancarlo, Heinrich and Gerd, 2002[2]). In this research, they also compared underlying ontology of General Ontological Language (GOL) and BWW-ontology as well as other ontology, such as the IEEE Standard Upper Ontology. Cranefield and Purvis investigated the use of UML as an ontology modeling language (Cranefield and Purvis, 1999). They compared common ontology modeling languages to

UML and also gave required extensions to UML for conceptual modeling. They used an example including UML structures of Class Diagram, Generalization, Association and Aggregation.

For the purpose of this research, the most relevant is Evermann and Wand's research on using UML for conceptual modeling. They not only examine the defects of UML, but also develop constraints (rules) to make UML better suited for domain modeling.

> This research goes beyond pointing out the ontological defects and instead suggests rules and guidelines for the modeler and analyst which alleviate these problems and enable the use of UML for conceptual modeling of real-world domains. (Evermann and Wand, 2003, p215)

We will talk about Evermann and Wand's research specifically in the next section.

2.3Evermann and Wand's research

Evermann and Wand's research not only analyzed and examined the suitability and pointed out the weakness of UML for domain modeling, but also developed constraints (rules) to make UML better suited for domain modeling.

2.3.1 Conceptual modeling represents the real world

In Evermann and Wand's research, they point out that conceptual modeling involves representing aspects of the real world. The conceptual model is the output of system analysis and the purpose of system analysis is to describe the business and organizational domain. When people use UML for conceptual modeling purposes, UML is used as a tool to describe and help people to understand the real world. A tool to fulfill this job should be

based on the perspective of the real world. The system design stage is used for the implementation stage (coding) and UML is relatively well mapped with OO programming languages, so UML works well in the system design stage. If we also want UML to work well in the conceptual modeling and / or system analysis stage, UML needs to be well mapped into the real world, so that people can use UML to represent everything that exists in the world.

2.3.2 Ontology as the appropriate basis

We need a theory that is used to capture the real world, to do the mapping. Ontology is an appropriate theory to do this job. Ontology is the branch of philosophy dealing with the nature and structure of the real world. Ontology defines the real world, telling us what the world consists of and how the world works.

There are several ontology. For example, Brentano's ontology (Routledge and Paul, 1874), Gottlob Frege on Being, Existence, and Truth (Klemke, 1968), Kazimierz Twardowski on Ideas and their Intentions (Nijhoff, 1977), and so on. Within the ontologies, Bunge-Wand-Weber's Ontology (BWW-ontology) is a philosophy specific to science. It is a combination of three researchers work. The BWW-ontology model is based on Bunge's Ontology (Bunge, 1977, 1979), and then Wand and Weber applied this ontology to Information Systems Analysis (Wand, 1989; Wand and Weber, 1989, 1990, 1991, 1993, 1995; Wand, Veda and Weber, 1999; Weber and Zhang, 1996; Green and

Rosemann, 2002). Evermann and Wand's research chooses BWW-ontology for the following reasons:

1. It is well formalized in terms of set theory and has not been developed specially for use in information systems analysis and design.
2. It has been successfully adapted to information systems modeling and shown to provide a good benchmark for the evaluation of modeling languages and methods.
3. It has been used to suggest an ontological meaning to object concepts.
4. It has been empirically shown to lead to useful outcomes by Bodart and Weber (1996); Gemino (1999); Weber and Zhang (1996). (Evermann and Wand, 2001a,p3).

All these features of ontology make it appropriate as a basis for developing rules about what a conceptual modeling language should do. The BWW-ontology gives the definitions of: "Thing", "Property", "Change", "Law", and "State". It describes the world like this: The world consists of things that possess properties. Properties are either mutual or intrinsic. Every thing has states by the specific value of its properties. States change as the property value changes. All these changes follow rules, which are called laws. Every thing can change. Change is either qualitative, in which a thing acquires or loses properties; or quantitative, in which property values of a thing changes. Two things are said to interact when they act on each other.

2.3.3 Evermann and Wand's ontological rules

From all of these assumptions, Evermann and Wand have developed a set of ontological rules that place constraints on the construction of UML diagrams to ensure that they properly represent underlying ontological assumptions. In this section these rules and how they function will be discussed. An example of a specific rule is: "All classes must

possess at least one attribute." This rule comes from the definition of "Thing" in BWW-ontology. In ontology, the world consists of things that possess properties. Things change as the properties change. (UML-Object corresponds to BWW-Thing and UML-Attribute corresponds to BWW-Property.) If objects are defined without specifying any attribute, what is the difference among them? People may argue that they have different names. The class (object) names are only symbols to help people locate them easily. Without any attribute, the names have no meaning. For example, Mr. A and Mr. B buy the same book in a bookstore. Now, the book class has a new mutual attribute "Owner". The two books are becoming two different objects (A's book and B's book), because the values of the book attribute (Owner) are different. If we do not include the "Owner" attribute in the "Book" class, the two books will not be different. The difference among objects in a class is the different value of their attributes. A class has to have at least one attribute to distinguish different objects in that class. Thus, to define a class without any attribute may be a serious mistake. We need this rule to help us prevent it from happening.

The example above illustrates one of the Evermann and Wand's ontological rules. There are in total 36 rules and 39 corollaries. We include all of them in Appendix C. These rules and corollaries will be reviewed and explained in Chapters 4, 5 and 6.

2.4 Existing UML-based CASE tools

Although some research has tried to help UML become more suitable for conceptual modeling, notably Evermann and Wand's ontological rules, there is no existing UML-based Computer Aided System Engineering (CASE) tools that reflect these rules. Here, the practical UML tools will be reviewed. In existing UML Tools, the Rational Rose series of products (Rose Professional) are the most well known and popular. They are developed by the Rational Company, which is now a sub-branch of IBM. These products provide support for different UML concepts. The UML standards are also very well manifested in Rational Rose. They integrate the diagram and notation template for users to choose. When people use Rational Rose to draw UML diagrams, they only need to choose a template and fill in related content information into corresponding area. For example, "name" and "attribute" for a class diagram.

Despite the fact that Rational Rose supplies a full coverage of UML concepts, it only does one job, to help people draw diagrams. (Rational Rose also does "Code Generation", but the focus of this research is on the UML diagram layer of the products.) An ideal UML tool should have the ability not only to help people draw diagrams, but also to guide people how to draw diagrams well by adopting some constraints. An ideal tool should have some packages to examine the quality of diagrams. The tool should integrate some theory to help people draw good UML diagrams. Rational Rose only realizes this function in low level. It examines UML diagrams, according to UML semantics and syntax. It does not examine UML diagrams based on any other theory, for example, the research

discussed above. If you use a dashed line to connect two classes as an association, Rational Rose will prohibit you from doing that because it violates the UML syntax. However, if you identify a "Student" as an association, which is supposed to be a class, there is no way for Rose to indicate such a problem so far.

We have also reviewed part of other UML tools; none handles the second function better than Rational Rose does. From the above review of existing UML tools, we conclude that no existing UML-based CASE tool enforces Evermann and Wand's ontological rules.

2.5 Expert system for implementing Evermann's rules

Our research implements a mechanism to check UML diagrams to see whether they violate Evermann and Wand's ontological rules. We adopt technology of expert systems, which critiques modeler's UML diagrams. In this section, we review critiquing expert systems.

Critiquing expert systems have a history of over twenty years. In 1981, Langlotz and Shortliffe developed the ONCOCIN system. (Joson, 1998, p15) It is to critique doctors' treatments of patients. The treatment plan given to a patient by a doctor is the input data of the critiquing system. The system critiques the treatment and gives advice to the doctor. In 1989, Fischer et. al developed Janus, a system that can aid users designing a household kitchen. (Jason, 1998, p17) When the designer creates a kitchen plan, the system responds

based on its critiques. There are other critiquing expert systems such as, ATTENDING (1983), CLEER (1992), VDDE (1993), TraumaTIQ (1993), AIDA (1995), SEDAR (1995), ICADS (1997). Since our research relates to critiquing systems in the domain of software development, we focus more on critiquing expert systems of software analysis and design. That is, the expert systems help users to develop software. In this domain, we will discuss four systems: Framer, KRI / AG, UIDA and Argo. In 1990, Fischer, G. et al developed the Framer system. (Jason, 1998, p18) It is a tool for designing user interfaces. When designers are working on an interface, the system critiques the design and lists advice. For example, if a "Menu Bar" has not been created yet, the system will prompt the user to "Add a menu bar" in its "Things to take care of" panel. Another critiquing system is KRI / AG, developed by Lowgren and Nordqvist in 1992. Similar to Framer, KRI / AG is also used for designing graphical user interface (GUI). It critiques the user's design according to 70 rules based on published guidelines on designing Motif user interfaces. However, Jason pointed out that "KRI / AG does not satisfy all of our requirements for a critiquing system because it is not integrated into a design tool and it is not tightly integrated into the designer's tasks" (Jason, 1998, p18). User Interface Design Assistant (UIDA) is another expert system for user interface design. Bolcer, G. A. applied 72 UI style rules into UIDA for critiquing user designs. Since it is a stand-alone system, UIDA does not satisfy all of the requirements for a critiquing system. The last critiquing system in software development is the Argo family, which consists of three tools (Jason, 1998,

p21). The first is ArgoC2, a tool for high-level software architecture design. PREFER is another tool for state-based requirements documentation design. The last tool is ArgoUML for software system design. These three tools use the same infrastructure to support their critiquing function. Since we choose ArgoUML as the platform for our research, we will discuss it in the next chapter.

Chapter 3 Research Approach

In this chapter, we introduce our research approach. We will first explain which CASE tool we choose and why we chose it. Then, we explain principles we followed in the research. Finally, we specify the order in which we discuss rules and corollaries.

3.1 Choice of ArgoUML

To implement the ontological rules, we need to choose a UML CASE tool as the platform. There are several UML products that help people to draw UML diagrams. We chose ArgoUML as the platform for our research based on the following reasons.

3.1.1 Critic system in ArgoUML

ArgoUML is a UML CASE tool for software system design. As we mentioned above, it also supports the function of critiquing user's design. It uses the ADAIR critiquing process, which is Activate, Detect, Advise, Improve, and Record. That is, when ArgoUML starts, its critiques will be activated. When users draw UML diagrams, it will detect mistakes and advise users. Then it can help users to improve their design by solving the problems. Finally, it can record mistakes as well as their solutions for future reference.

ArgoUML's critiquing is based on UML syntax. For example, a class name normally begins with a capital letter.

As we can see, ArgoUML is a UML CASE tool, which well supports critiquing a user's design. It is convenient for us to implement Evermann and Wand's ontological rules into this CASE tool.

3.1.2 Open source

In our system, the input data are UML diagrams drawn by the CASE tool. The system will read the diagrams and check if they violate those rules. However, diagrams created by different products have different file format. To read a file, we must know its format. Since most business products code their file for copyright protection purpose, our system could not read diagrams drawn by them. However, ArgoUML is an open source project and to get related information from diagrams for our checking is not a problem.

> ArgoUML is an open source project. The availability of the source ensures that a new generation of software designers and researchers now have a proven framework from which they can drive the development and evolution of CASE tool technologies (CollabNet Inc. 2003).

As we know, ArgoUML is not the most popular tool people choose to draw UML diagrams. However, we still chose it because it is open to us. The format, the structure, and the code are all free to read. In addition, our implementation is for research purposes, not for business purposes, so we do not concern whether our software can be used for the most popular UML product.

3.2 Add to ArgoUML

Our research follows the following three principles. First, we do not change or delete any function of ArgoUML, but add our implementation to it. This is because Evermann and Wand's ontological rules currently have not been adopted by UML (OMG), we should not change or delete UML components. Second, for rules and corollaries we propose to revise or delete, we still implement them because our basic purpose of this research is to implement these rules and corollaries. Revising, deleting and creating rules and corollaries are additional contributions. Third, for rules / corollaries that cannot be implemented in ArgoUML by our proposed approach for general CASE tools, we use an alternative implementation for UML. This is because that approaches given in Chapter 4, Chapter 5 and Chapter 6 are general ideal solutions to a fully functional CASE tool. However, ArgoUML currently does not support all UML specifications. Alternative solutions for affected rules / corollaries are discussed in Chapter 7.

3.3 Discussion order

According to Evermann, "Our world consists of a static structure of things with their properties, changes in things and interactions of things" (Evermand, 2003, p37). Their rules and corollaries based on this order. Since we are implementing these rules and corollaries, in addition some latter rules / corollaries are based on the previous ones, our discussion follows the order of Evermann and Wand's rules / corollaries for convenience

and consistency. That is, rules and corollaries related to static, change and interaction are discussed in Chapter 4, Chapter 5, and Chapter 6 respectively.

Another thing we have to note here is that even through we evaluate Evermann's rules, this is not our main purpose but an additional contribution. For most rules, our research borrows terminology from Evermann and we assume they are correct

Chapter 4 Static Rules

Things, properties and compositions are static concepts in ontology. In this chapter, we will discuss the approach of implementing rules related to static structures.

Rule 1 Only substantial entities in the world are modeled as object.

First, the key words "substantial entities" in this rule need to be explained. In BWW-Ontology, "thing" is the basic element in the world and it refers to "substantial entity" (Bunge, 1977, p110). Substantial entities are something which physically exist in the world. They have two meanings. First, they have to be entities. That is, the entities are actually something in the world. Second, they have to be substantial. They can be seen, heard or felt by humans. For example, 'book' is a substantial entity since we can see it; 'sound' is a substantial entity since we can hear it, and 'wind' is a substantial entity since we can feel it. However, 'job', and 'order' are not substantial entities since we cannot see, hear or feel them.

Now, let us see another example: 'air'. Is 'air' a substantial entity or not? Some people may say that 'air' is not substantial, because we cannot see it, hear it, or feel it. Not only 'air', but also 'wind', and 'water' are different from 'book'. The 'air' is not so easily

perceived by human as 'book'. In Evermann and Wand's research, they did not discuss such a group of entities. In this case, we propose a sub-classification of substantial entities. The substantial entities consist of boundary entities and nonboundary entities. For the boundary entities, they have physical boundaries, like 'book', and 'car'. For the nonboundary entities, they do not have physical boundaries, like 'air', and 'sound'. However, for nonboundary entities, for example, 'air', even though we cannot see, hear or feel it, the 'air' physically exists in the world. Thus, even boundary entities and nonboundary entities are different; they still belong to substantial entities. In our research, we still propose them to be substantial. Because Rule 2 is also related to this issue, we will discuss it in another way in Rule 2.

To realize this rule in the CASE tool, we mapped the "substantial entity" into human language. The human language consists of sentences and sentences consist of words. Thus, words are the basic element of a language. Every word has its part of speech. For example: noun, verb, and adjective. We find that most "entities" can be mapped to nouns. To let the computer know whether a noun is substantial or not, we used the following approach: First, classify all nouns (in a dictionary) into two groups: Substantial and Non-substantial. Then we create a database to store those nouns. When checking UML diagrams, the program gets the "object" and match in the database. If an object is "Non-substantial", the program gives an error and shows this rule. If an object cannot be found in our database, the program will interact with the user by asking whether that object is substantial or not. The

program will remember the new substantial or nonsubstantial noun in its database. Thus, the system has the ability to learn by remembering newly added nouns. This is expressed in the following:

```
Get object/class name;
Map name with substantialDB;
        If (name = non-sub)
                Show error;
                Explain error;
        Else if (name = sub)
                Exit;
        Else query the user;
                Explain what is sub and nonsub;
                Give sub and non-sub examples;
                Ask user to select sub or non-sub of that name;
                If select = non-sub
                        Show error;
                        Add to nonsub DB;
                Else if select = sub
                        Add to sub DB;
                        Exit;
```

This approach also has shortcomings. Nouns and substantial entities are not strictly one to one mapping. Thus this approach may be inaccurate sometimes. In addition, if the user uses an abbreviation for an object name or a class name, the system cannot find it in the database. However, we can propose an approach to reduce the problem. A heuristic search algorithm can be adopted. If the user's input is similar to a word in our database, the system can ask the user whether the input is the same as the database's word. For example, 'school' is listed in the subnouns. If the user inputs 'highschool', the system can ask the user whether the 'highschool' is a subclass of 'school'. Actually, in this case,

'highschool' consists of two words: 'high (adj.)' + 'school (n.)'. The system can match the 'school' in subnouns with all kinds of schools, such as, 'seniorschool', 'juniorschool', and 'primaryschool.'

Rule 2 Ontological properties of things must be modeled as UML-attributes.

"Property" is another important concept in ontology, since every thing owns properties (Bunge, 1977, p58). We have already explained that only substantial entities should be modeled as UML-objects. What should we model the nonsubstantial entities as? This rule tells us that nonsubstantial entities should be modeled as UML-attributes. This is because things own properties and UML-objects own UML-attributes. We model things, which are substantial entities as UML-objects and model properties, which are nonsubstantial entities as UML-attributes. An example of this rule is that 'job' and 'color' must be modeled as UML-attributes.

Now, let us analyze the nonboundary substantial entities based on both Rule 1 and Rule 2. As we already know, entities in the world can be classified as three groups: boundary substantial entities, nonboundary substantial entities, and nonsubstantial entities. In Rule 1, it is clear that boundary substantial entities are modeled as objects. In Rule 2, it is clear that nonsubstantial entities are modeled as attributes. For the third group, nonboudary substantial entities, should we model them as objects or attributes? Let us analyze some specific examples. Boundary substantial entities, such as 'worker' are

normally modeled as objects, while nonsubstantial entities, such as 'skill' are normally modeled as attributes. The relations between objects and attributes are that objects own attributes, like 'workers' possesses 'skills'. To answer the question we gave above, we have to answer: Are nonboundary substantial entities more like objects or attributes? 'Water' is an example of nonboundary substantial entity; it is hard to model 'water' as an attribute of an object. It is hard to find an object which possesses 'water'. On the other hand, 'water' owns attributes such as 'color', 'temperature', and 'taste'. It seems that 'water' is more like an object than an attribute. Thus, the same conclusion as discussion in Rule 1 is reached. That is nonboundary substantial entities are still substantial entities and should be modeled as objects.

We can use the following approach to realize this rule. Since properties are nonsubstantial entities, we can still map nonsubstantial entities into human language. As discussed in Rule 1, entities can be mapped with nouns. Thus, properties can be mapped with nonsubstantial nouns. The database of substantial nouns and nonsubstantial nouns in Rule 1 can also be used here. When checking UML diagrams, the program needs to get all the text strings from all diagrams. Then, all the strings will be matched with the substantial nouns and nonsubstantial nouns database. If the program finds any string in the nonsubstantial database, then it checks whether the string is an attribute. If the string is not an attribute, then the program issues a violation warning. For the noun which is not in any database, the program will inquiry the user and add it to the appropriate database. This is

expressed in the following:

```
Get text strings;
  Map string with substantialDB;
        If (string = sub)
              Exit;
        Else if (name = non-sub)
              Retrieve String_type;
              If (String_type = attribute)
                    Exit;
              Else Show error;
                    Explain;
        Else query the user;
              Explain what is sub and nonsub;
              Give sub and non-sub examples;
              Ask user to select sub or non-sub of that name;
              If select = non-sub
                    Show error;
                    Add to nonsub DB;
              Else if select = sub
                    Add to sub DB;
                    Exit;
```

Corollary 1 Attributes in a UML-description of the real world cannot refer to substantial entities.

This is the corollary of Rule 2. In Rule 2, we have known that, "ontological properties", which are nonsubstantial entities, must be modeled as UML-attributes. That means that UML-attributes must be nonsubstantial entities. This is Corollary 1 as stated above.

As we can see, this corollary has a very similar structure to Rule 1. We use the approach for Rule 1 to realize this corollary. We still use the substantial nouns and

nonsubstantial nouns database and the matching approach. Note that association classes are also properties (we will explain this in Rule 3) and thus should be checked. This is expressed in the following:

```
Get attribute name;
    Get associationClassName;
      Map name with substantialDB;
      If (name = non-sub)
                Show error;
                Explain error;
      Else if (name = sub)
                Exit;
      Else query the user;
                Explain what is sub and nonsub;
                Give sub and non-sub examples;
                Ask user to select sub or non-sub of that name;
                If select = non-sub
                          Show error;
                          Add to nonsub DB;
                Else if select = sub
                          Add to sub DB;
                           Exit;
```

Rule 3 Sets of mutual properties must be represented as attributes of association classes.

Now, let us talk more about properties. There are two kinds of properties: intrinsic properties and mutual properties. Intrinsic properties are the properties that only belong to one thing. Mutual properties are the properties that belong to more than one thing (Everman, 2003, p38). That is, for a single thing, the mutual properties do not exist. For example, a customer orders a book from a bookstore. The 'customerName' is the intrinsic property of the 'Customer'. However, the 'orderNumber' is the mutual property of the

'Customer' and the 'Bookstore'. The order happens between the customer and the bookstore. Neither of them can own the order by itself.

In this rule, we will discuss mutual properties. First, in Rule 2, we have explained that properties should be modeled as UML-attributes. Mutual properties are a subtype of properties. Thus, they should be modeled as attributes. Second, mutual properties are properties not belonging to a single thing, but a group of things. As we map BWW-properties as UML-attributes and BWW-things as UML-objects, this means that mutual UML-attributes belong to a group of UML-objects. As these objects have mutual attributes, there must be some association to connect these objects together. In UML, association classes are used to represent associations between objects. Since association class is a kind of class, it has attributes. This rule specifies that mutual attributes should be modeled as the attributes of association class. Like the 'orderNumber' we discussed in Rule 2, it is not the intrinsic property but the mutual property of 'Customer' and 'Bookstore'. Thus, we cannot have the 'orderNumber' as attributes in both 'Customer' and 'Bookstore' classes. The fact is that the customer orders a book from the bookstore. 'Order from' is the association between customer and bookstore. We can have an association class 'Order' and model 'orderNumber' as the attribute of this association class.

Readers may have a question here. Classes are used to represent substantial things. However, this rule specifies that association classes are used to represent mutual properties, which are nonsubstantial entities. This is because association classes are a special kind of

class. They have characters of both association and class. To separate association classes, we call other classes ordinary classes. So, we give constraints that Rule 1 does not apply to association classes but only to ordinary classes.

We can give a clearer description of properties. Even through intrinsic properties and mutual properties are both properties and must be modeled as UML-attributes, they are modeled as different attributes. First, intrinsic properties should be modeled as attributes of ordinary classes. Second, mutual properties should be modeled as attributes of association classes.

In other words, mutual properties should not be represented as attributes of ordinary classes associated by an association. That is, there should not be the same attribute in two classes. We use the following approach to implement this rule. First, the program gets all of the associations in a diagram. Then it checks all attributes in each association end. To do this, the program first gets all the attributes from the first associated class, and writes them into a vector. Then an attribute in the second class is retrieved and matched with the vector. If two attributes are found with the same name, the system shows a violation warning. If not, the program gets the next attribute in the second class and matches it with the vector again, and so on. This is expressed in the following:

```
Get association;
Get associationEnds;
Get end_class1;
Get class_attributes;
While class_attributes.hasNext
```

```
        Add attribute to vector;
    Get end_class2;
    Get class_attributes;
    While class_attributes.hasNext
        Match with attri_vector;
        If match = true
            Show error;
    Exit;
```

In UML, there are two special associations: composition and aggregation. We will discuss composition in Rule 6. Here we discuss aggregation. Aggregations are parts-whole associations and the whole classes are the aggregate classes. Based on this, if a designer models the same attributes in whole and part classes, the "mutual" attribute should not be modeled as the association class. That is no association class can be connected to an aggregate association. Thus, we add the constraint that the examination for Rule 3 does not check an aggregation association. In the implementation, the program simply skips this rule if it finds the association end is an aggregation.

We also need to consider other two special cases. First, attributes with the same name in associated classes may represent different meanings. That is, they are not really mutual properties. For example, in 'Customer buys Book', 'name' attributes of both 'Customer' and 'Book' classes are not mutual attributes possessed by both of them. Actually, they refer to 'customerName' and 'bookName', respectively. In this case, the examination will show a wrong violation warning. To reduce this effect, we can enable users to dismiss the warning. Second, attributes with different names in the associated class may represent the same meaning. That is, they are actually mutual properties. For example, in 'Traveler takes

Plane', 'flyPeriod' of traveler and 'inAirDuration' of plane are mutual attributes possessed by both of them. Actually, they both refer to the 'flyingTime'. In this case, the examination will not show a violation warning. To solve this problem, the possible solution is to adopt a dictionary (we only give this solution idea and do not implement it). It lists different words with the same meanings. For the specific domain, a domain related glossary could be even used.

Corollary 2 An association class cannot represent substantial entities or composites of substantial entities.

Because mutual properties must be modeled as attributes of association class, mutual properties cannot be modeled as attributes of ordinary class. From Rule 2, we know that only intrinsic properties can be represented as attributes of substantial entities. If we assume that an association class can represent substantial entities, then we get that the attributes of the association class must be intrinsic properties. This is obviously in conflict with Rule 2 and Rule 3. Thus, the assumption is incorrect. Also the composition of substantial entities is a substantial entity, thus Corollary 2 is approved. Here are two examples of this corollary: The 'Worker' in a 'company hires worker' relation cannot be modeled as an association class. The 'plane' in a 'plane consists of engine and body' relation cannot be modeled as an association class.

To implement Corollary 2, we can simply match association class names with our

substantial nouns and nonsubstantial nouns database. The examination will show a violation warning when matching an association name with a substantial noun. This is expressed in the following:

```
Get association_ class name;
Map name with substantialDB;
If (name = sub)
        Show error;
        Explain error;
Else if (name = non-sub)
        Exit;
Else query the user;
        Explain what is sub and nonsub;
        Give sub and non-sub examples;
        Ask user to select sub or non-sub of that name;
        If select =sub
                Show error;
                Add to sub DB;
        Else if select = non-sub
                Add to non-sub DB;
                Exit;
```

Corollary 3 If an association class of an n-ary association is intended to represent substantial things, the association should instead be modeled as one with arity (n+1).

We have proved that an association class cannot represent substantial things. So if this situation happens, how can we correct it? Corollary 3 is actually the solution of a kind of violation of Corollary 2. The n-ary association means that the association, which owns the association class, has n association ends. In other words, the association connects n ordinary classes.

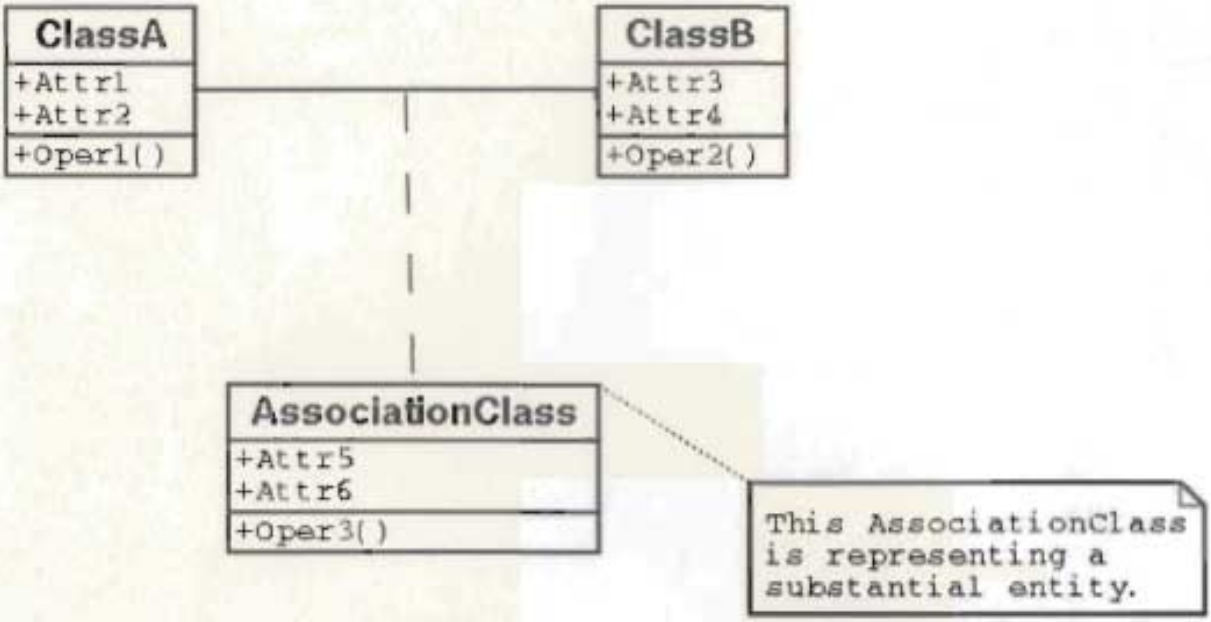

Figure 4.1 Incorrect 2-ary association class

In this case, if the association class is intended to represent substantial things, the solution is to change the association class into a n+1 ordinary class. By doing this, the association class will no longer belong to the association, but will be an association end of the association. For example, in the 'customer buys book from bookstore' relation, someone uses an association class to represent the 'book.' This is incorrect. Instead, the 'book' should be modeled as an ordinary class along with the 'customer' and 'bookstore' class. Figure 4.1 and Figure 4.2 describe this process.

Since this corollary is the solution of Corollary 2, there is no need to implement Corollary 3.

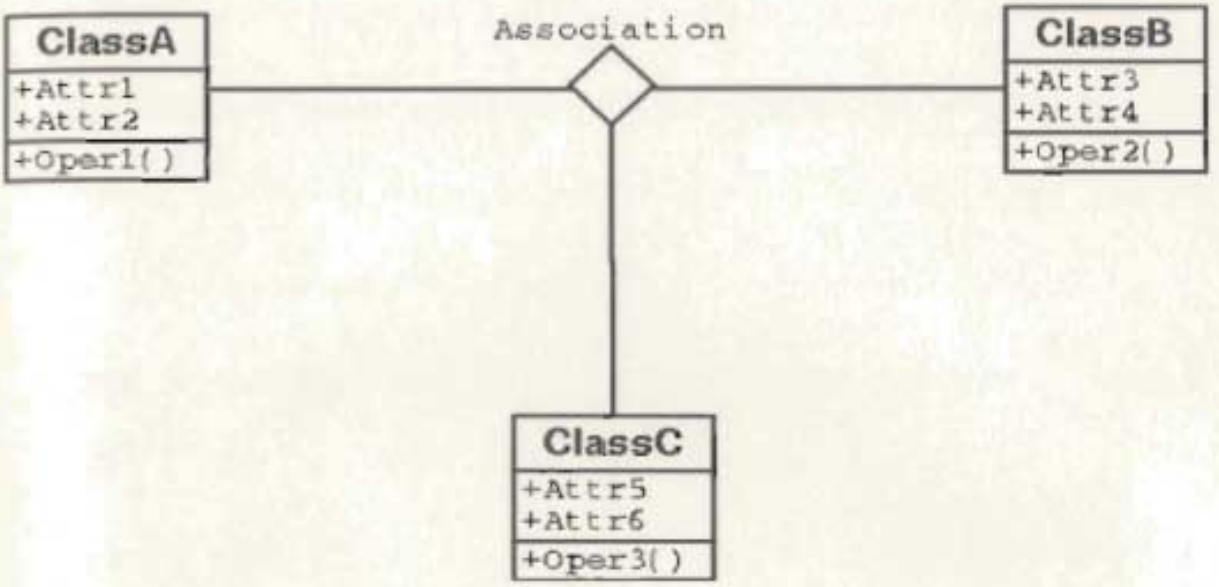

Figure 4.2 Correct 3-ary association

Corollary 4 An association class representing a composite must instead be modeled as a composite with attributes representing emergent intrinsic properties.

We have proved that an association class cannot represent composites of substantial entities. So if this situation happens, how can we correct it? This corollary is actually the solution of another kind of violation of Corollary 2. An association class sometimes is used to represent a composite relationship, which is a part-whole relationship (Everman, 2003, p41). For example, among three classes: 'Engine, Body and Plane', it is incorrect to model 'Plane' as an association class between 'Engine' class and 'Body' class. This is because 'Engine' and 'Body' are two parts of 'Plane,' the 'Plane' class is actually a composite of 'Engine' and 'Body.' Thus, the 'Plane' class should not be modeled as an association class, but a composite class of 'Engine' and 'Body.' The attributes of the

composite class are emergent intrinsic properties. The composite class 'Plane' has emergent properties such as 'speed,' because neither 'Engine' nor 'Body' has that property. Also, the property 'speed' is the intrinsic property of the composite class 'Plane.' Figure 4.3 and Figure 4.4 illustrate this example.

Since this corollary is the solution of Corollary 2, there is no need to implement it.

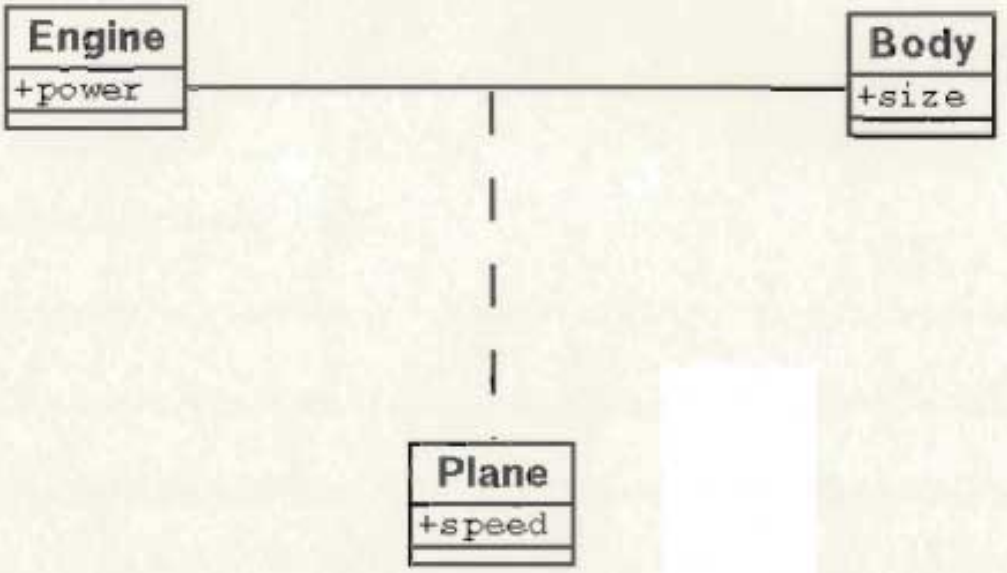

Figure 4.3 Incorrect model for Corollary 4

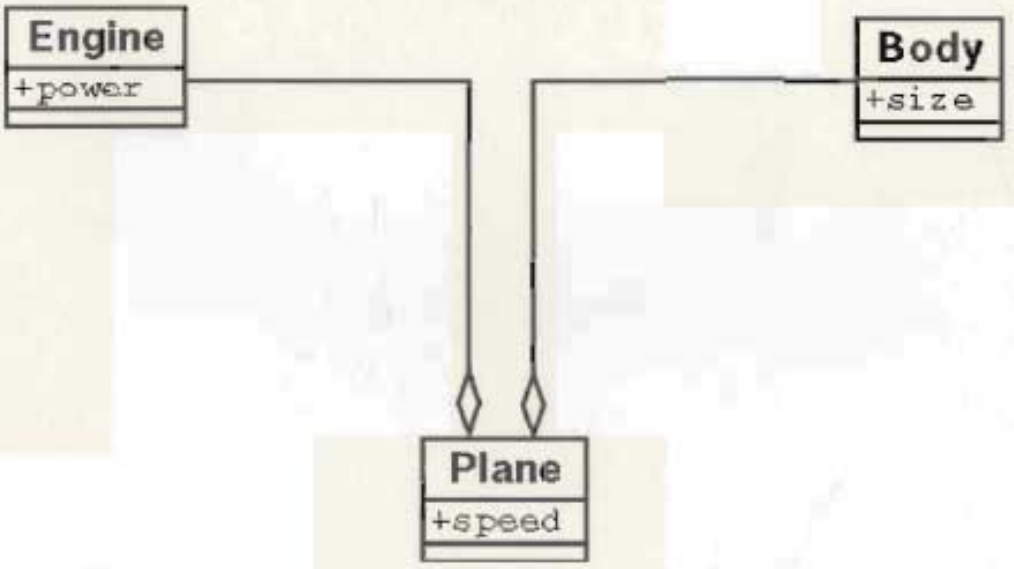

Figure 4.4 Correct model for Corollary 4

Corollary 5 An association class cannot possess methods or operations.

This is also a corollary of Rule 3. We have already shown that the association class cannot represent substantial entities. According to Everman's research, "all changes are tied to things. There can be no change without a thing that changes" (Everman, 2003, p45). That is, if there is no substantial entity change, nothing would change. Since attributes of association classes do not specify substantial entities, they should not change spontaneously. That is, methods or operations which are able to change the association class attributes, should not be modeled.

To implement this corollary, the examination can get all associations and check whether they have any methods or operations. If any method or operation is found, the program will give a violation warning to users. This is expressed in the following:

```
Get associationClass;
Get operations;
If operation = null
    Exit;
Else show error;
```

Corollary 6 An association class cannot be associated with a state machine.

This is also a corollary of Rule 3. In the last corollary we have explained that, if there is no substantial entity change, nothing would change. Now, we are talking about state machines. In UML, "A state machine can be used to model the behavior of class instances" (OMG, 2003, p2-140). Since state machines are used to describe the state change of an

element, some change must exist for that element. Ordinary classes represent substantial entities and own methods or operations, which can change the class states. However, association classes have no method or operation (Corollary 5), for their states to change. Thus, we should not model state machines for association classes.

To implement this corollary, we need to give this constraint to UML specification. Also, in UML CASE tools, the feature of modeling state chart diagrams should be disabled and there is no need to write a program to examine this.

Corollary 7 An association class must possess at least one attribute.

This is a straightforward corollary of Rule 3. Since sets of mutual properties must be represented as attributes of association classes, the attributes of an association class are used to represent the mutual properties. If an association class has no attribute, that means that the mutual properties set is empty. Because "while an empty set of mutual properties is still a set and thus technically satisfies Rule 3, it is ontologically meaningless" (Everman, 2003, P46), an association class must have an attribute.

The implementation gets all association classes and checks whether there is any attribute in each of them. If an association class attribute space is null, it shows a violation warning. This is expressed in the following:

```
Get association_class;
Get aClass_attribute;
If aClass_attribute = null
```

Show error;
Else exit;

Corollary 8 An association class must not be associated with another class.

According to Everman, "properties in ontology cannot themselves possess mutual properties with other properties or things" (Everman, 2003, p46). Because association classes themselves are mutual properties of associated classes, thus, association classes cannot have mutual properties with any others.

According to UML notation, association classes do not directly connect to classes. Instead, a dashed line connects association classes to associations. That is, association classes cannot be association ends. We can disable this function in CASE tool by proscribing association connection to association class. For those CASE tools that do not have the function of connecting association classes to ordinary classes, the corollary is automatically satisfied.

Corollary 9 An association class must not participate in generalization relationships.

According to Everman's research: "properties themselves cannot be generalized" (Everman, 2003, p47). Also, association classes are a group of mutual properties of participating ordinary classes. Thus, association classes cannot be generalized.

To implement this corollary, we need to include this constraint into UML specification. Also, we can disable this function in CASE tool. That is, to proscribe generalization

connections (triangle with solid line) to association classes.

Rule 4 If mutual properties can change quantitatively, methods and operations that change the values of attributes of the association class must be modeled for one or more of the classes participating in the association, objects of which can effect the change, not for the associations class.

This rule covers Corollary 5 and gives solutions to it. In Corollary 5, we already concluded that association classes could not possess methods or operations. So, where should we put these methods or operations? In other words, how are the attributes of association classes changed? This rule tells us the answer. Association classes are mutual properties of a group of ordinary classes. The methods and operations, which change the value of these mutual properties, should be put in those ordinary classes. That is, only methods or operations of ordinary classes participating in an association can change the attributes value of the association class. As well these methods or operations do not have to be modeled in all participating classes. Which ordinary classes own which methods or operations will depend on which ordinary classes actually change the attributes value. In the example of "customer orders book", there is an attribute 'date' of the association class 'order'. The method 'changeDate()' should be modeled in the ordinary class 'customer' instead of in 'order'. Figure 4.5 and Figure 4.6 depict this rule.

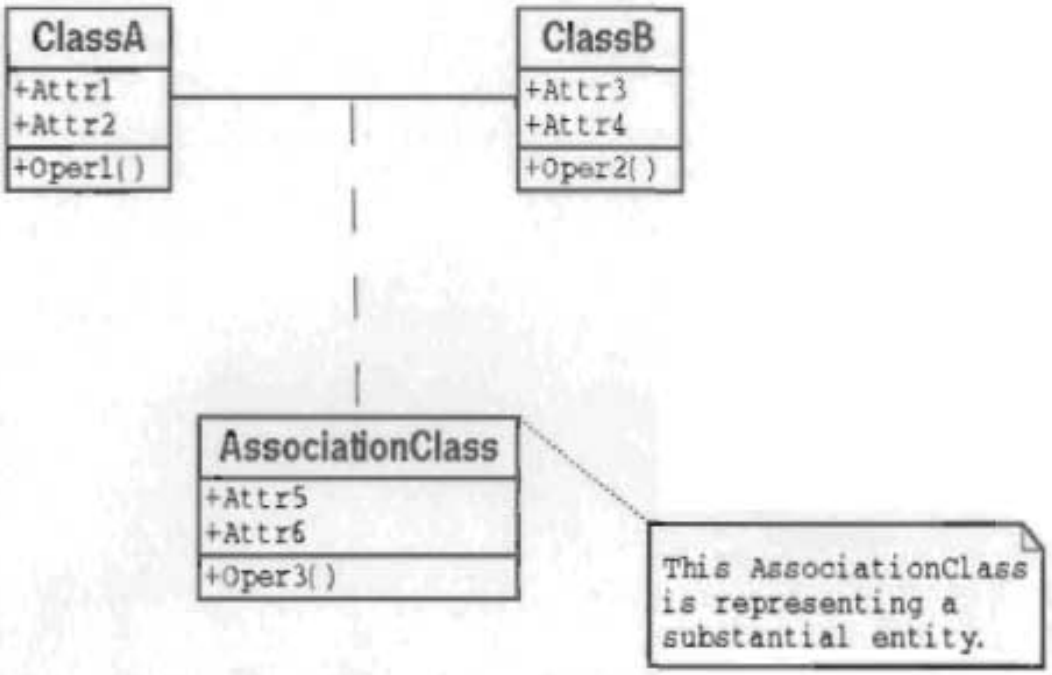

Figure 4.5 Incorrect model for Rule 4

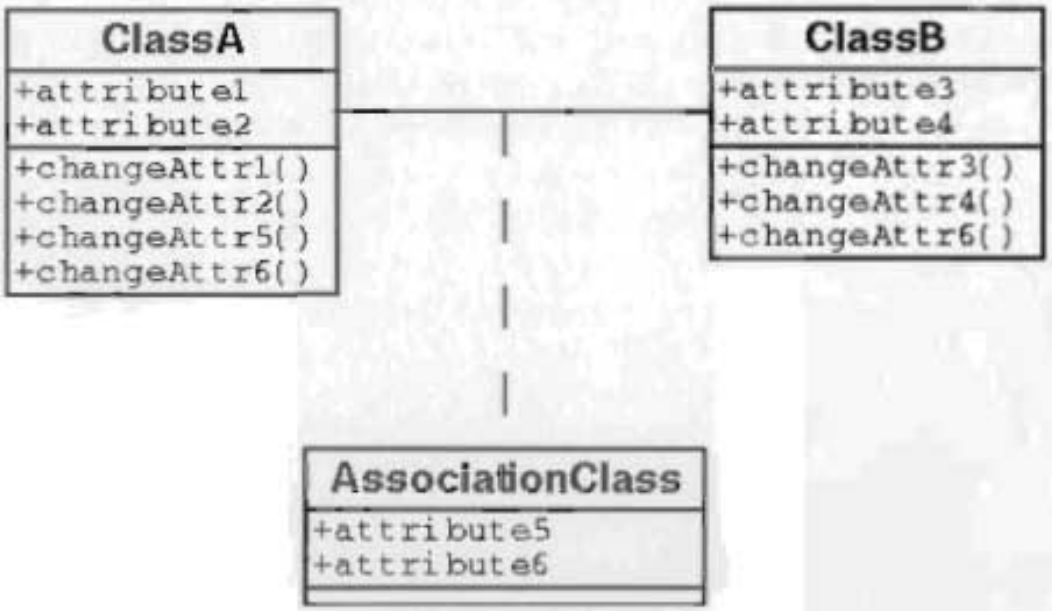

Figure 4.6 Correct model for Rule 4

We can look at this rule as two parts. The first part is to specify that association classes cannot own methods or operations. The second part is the solution that specifies how to

model these methods and operations. We just implement the first part and show the solution part in the text field. The examination can get all association classes, then gets their operation fields. If return values are not null, the program will show a violation warning. This is expressed in the following:

```
Get associationClasses;
Get operations;
If operations = null
    Exit;
Else show error;
```

Rule 5 An association class represents a set of mutual properties arising out of the same interaction.

According to ontology, interaction is defined through the state history of a thing: If the way attributes of one thing change depends on the presence of another, then the second is said to act on the first (Bunge, 1977, p258). Things interact, if and only if each acts upon the other (Bunge, 1977, p259) and everything acts on, and is acted on by other things (Bunge, 1977, p259). Association classes represent mutual properties of participating ordinary classes. However, all these mutual properties may not come from a same interaction. That is, some interactions create a group of mutual properties, while another interaction creates another group of mutual properties. For example, two ordinary classes: 'Worker' and 'Company' have mutual properties such as 'hiredate', 'firedate', 'salary', 'firereason', and so on. Not all these properties come from the same interaction. When the

'hire' interaction happens between a 'Company' and a 'Worker', it creates mutual properties of 'hiredate' and 'salary.' When the 'fire' interaction happens between a 'Company' and a 'Worker', it creates mutual properties of 'firedate' and 'firereason.' This rule specifies that the mutual properties created by a single interaction should be modeled in a single association class. Mutual properties created by different interactions should be modeled in different association classes. Like the above example, there should be two association classes created by 'hire' and 'fire', to own 'hiredate', 'salary' and 'firedate', 'firereason', respectively. This satisfied that ordinary classes could have more than one association.

It is difficult for a program to tell whether attributes in an association class come from the same interaction. What we can do is to interact with users for the critique. When the program finds the possible violation, it will let the user indicate whether it is a real violation. Once the program finds an association class, it gets all attributes from that association class and asks users whether these attributes are from one interaction. If the user indicates that they are not from a same interaction, the program gives a violation warning. This is expressed as the following:

```
Get associationClass;
Get attributes;
Query user;
If sameInteraction
    Exit;
Else show error;
```

There is a problem with this approach. It only focuses on whether all attributes in an

association are created from the same interaction. If a user models a set of attributes created by interaction 'A' in association class created by interaction 'B', the program still gets "Yes" from the user and will not show a violation warning. In fact, this case violates the rule. For example, the user models 'hiredate' and 'salary' as attributes of association class created by 'fire.' These two attributes are from the same interaction 'fire.' The user would indicate "Yes" when the system queries him. In this case, the examination will not show any warning information. We realize that every attribute in an association class and this association class must be created from the same interaction. The reason to give this constraint is that every attribute in an association class is created by the interaction that creates this association class. So we can revise this rule into the following:

All mutual properties arising out of the same interaction must be represented by the same association class.

Thus, we adopt another approach. The program will retrieve each attribute in an association class one by one. It will query users whether each attribute is created by the interaction creating the association class. If the user indicates, 'Yes', this attribute will be saved in an 'Ok' vector. Otherwise, this attribute will be saved in a 'Bad' vector. So if there is any attribute in the "Bad" vector for an association class, the examination will show a violation warning. In this way, we can guarantee that the above problem will not happen. If every attribute is created by the same interaction, which creates the association class, all these attributes must be created by the same interaction. This is expressed in the

following:

```
Get associationClass;
Get acAttribute;
Inquiry user "If acAttribute and AC created by a same interaction" ;
If yes
    Add attribute to Ok;
If No
    Add attribute to Bad;
While acAttribute in Bad
    Show error;
Exit;
```

Rule 6 A composition relation must not be modeled.

In UML, there are two associations related to the whole-part relationship: composition and aggregation. Both composition classes and aggregation classes are the whole of some part classes. For example, a 'person' class is the composition of 'arm' classes, 'head' classes, and so on. The difference between composition and aggregation is that composition is a many to one relationship, while aggregation is a many to many relationship. In composition, part classes can only attend one composition relationship. For example, the 'Arm' can only belong to one 'Person'. In aggregation, part classes can attend more than one aggregation relationship. For example, the 'Person' can belong to several 'Groups'. However, according to Everman and Wand's research, there are only composition relationships in ontology (Evermann, 2003, p50). Another important thing is that we need to separate the term 'composition' in ontology and the term 'composition' in UML clearly. The term 'composition' in ontology does not refer to composition in UML.

Instead, it matches with aggregation in UML. That is, UML-composition cannot match anything in ontology. Thus, this rule specifies that the composition relations in UML should not be modeled. For example, the 'plane-engine-body' relation should be modeled as a UML-aggregation instead of a UML-composition.

To realize this rule, we need to add this constraint to UML specification. Also, we can disable this feature in CASE tools. That is, to proscribe composition (hollow diamond) as association ends.

Rule 7 Every UML-aggregate must possess at least one attribute which is not an attribute of its parts or participate in an association.

We have discussed that UML-aggregate is a whole-part relationship. This rule specifies more constraints on aggregations. It is stated that "In the BWW-ontology, a composite must possess at least one emergent property, otherwise there exists not a composite but only a set of things" (Evermann, 2003, p51). This means that aggregate classes should not only be the whole of their parts, but more than that. For example, in the 'Person has Arm, Head', the 'Person' acquires the new emergent attribute of 'Language' which is not processed by 'Arm' or 'Head'. This rule can be illustrated in the following: Classes A and B are two parts of aggregate class C. A possesses attributes attr1 and attr2, B possesses attributes attr3 and attr4. This rule tells us that there should be at least one attribute attr5 not in the set of {attr1, attr2, attr3, attr4}; or class C participates in an

association. Note that the sentence in quotations above is talking about the composite in ontology, namely, aggregate in UML. We give Figure 4.7, Figure 4.8 and Figure 4.9 to describe this rule.

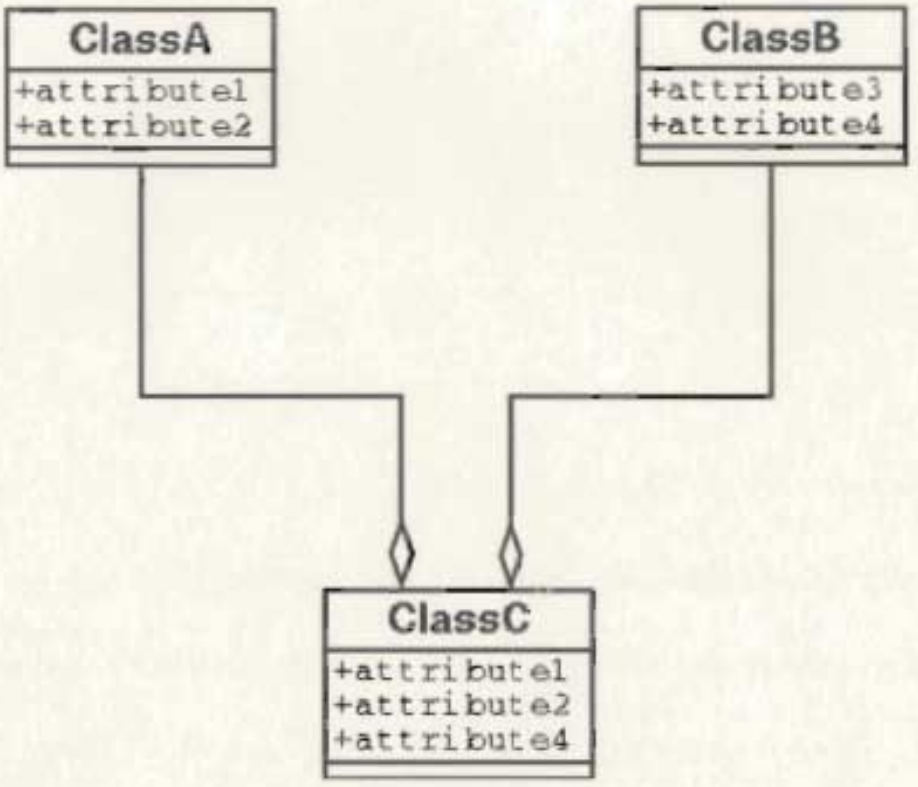

Figure 4.7 Incorrect model for Rule 7

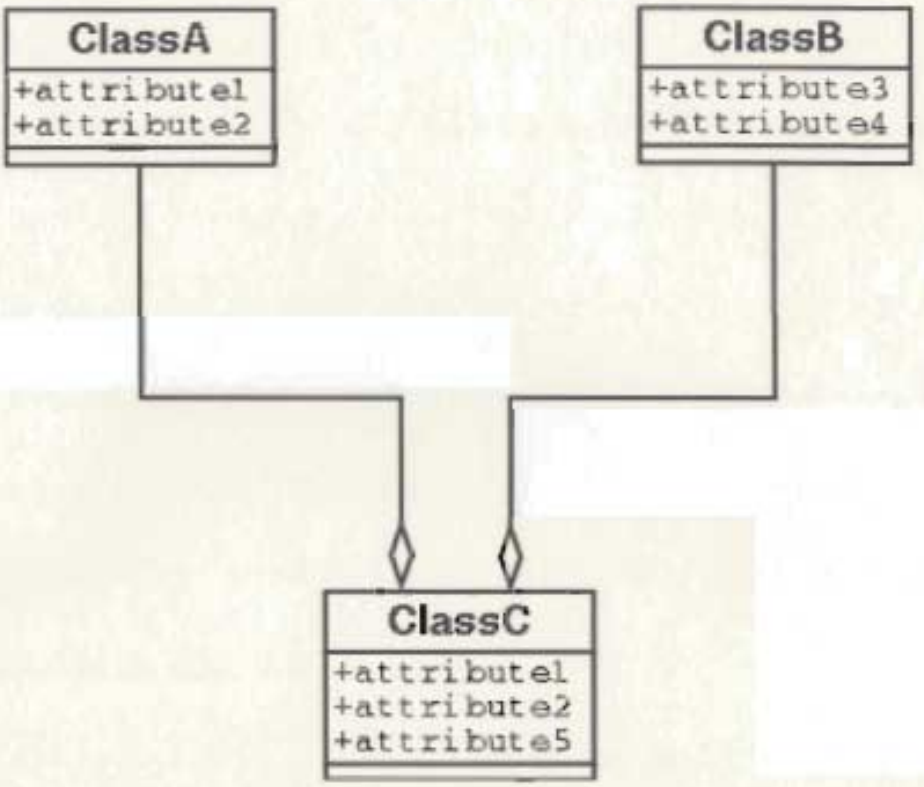

Figure 4.8 A correct model for Rule 7

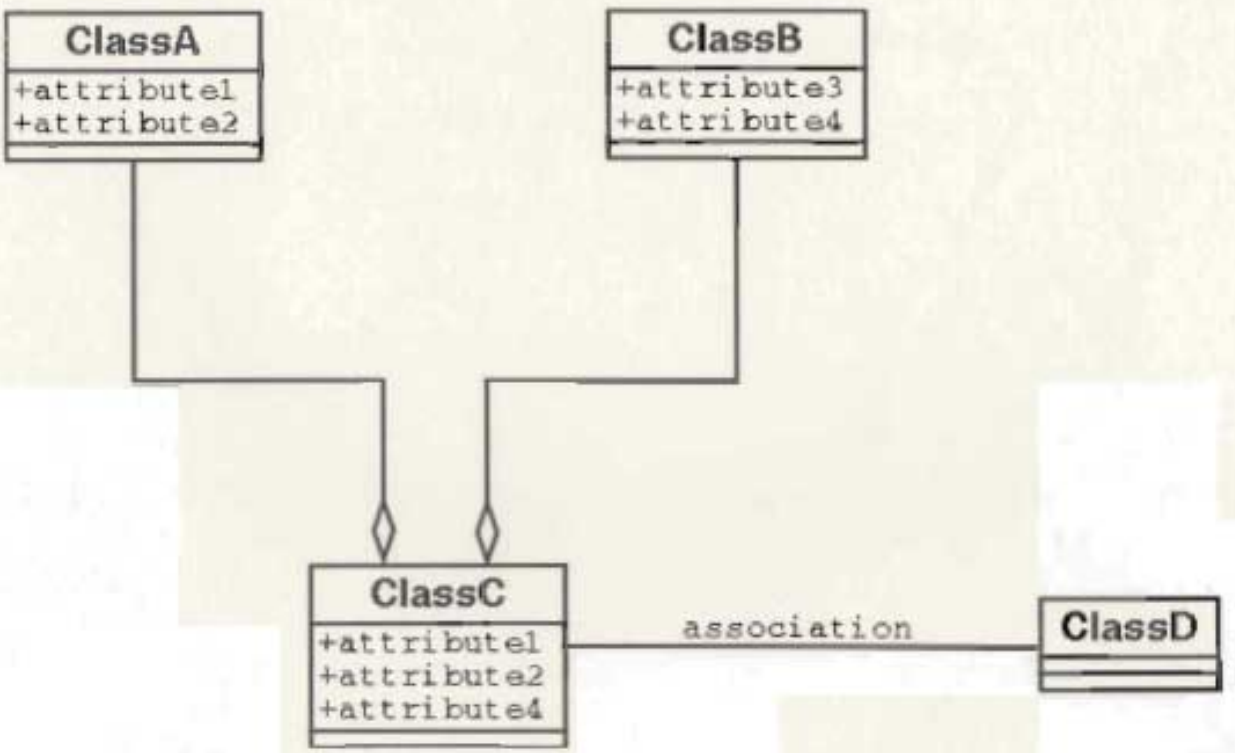

Figure 4.9 Another correct model for Rule 7

To implement this rule, we use the following approach. First, the program needs to find aggregate classes. This is done by checking each association end of each class. If an aggregation end is found, the program gets this association. Now, we can get attributes of both aggregate class and part classes. However, we only get the part class attributes and add them to a vector. After all part class attributes are added into the vector, the program begins to match each attribute in the aggregate class with the vector. If any aggregate class attribute is not found in the vector, the examination terminates without a violation warning. This is expressed in the following:

```
Get classes;
Get associationEnds;
If associationEnd=aggregation
    Get association;
    Get partClasses;
    Add pcAttributes to vector;
Get aggregateClasses;
Get acAttributes;
While acAttributes.hasNext();
    If !vetor.contains(acAttribute)
        Exit;
Show error;
```

Rule 8 All UML-classes must possess at least one attribute or participate in an association.

In Rule 1 and Rule 2, we have discussed that classes represent things and attributes represent properties. In ontology, a thing is different from another because they own different properties. That is, if things have no properties, we cannot distinguish different

things. Also, things map to UML-classes and properties map to UML-attributes. Thus, we can get that UML-classes must own attributes. On the other hand, in UML, "A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics" (OMG, 2003, p2-26). Attributes are fundamental elements of classes and classes must possess attributes. However, there is one case where a class can have no attribute. That is when the class participates in an association. However, on closer analysis, this is still following Rule 8. That is because if the class participates in an association class, the association class itself represents mutual attribute of all participating classes.

For the implementation, the program only needs to check all classes to see whether they have any attribute or association. If neither attribute nor association is found, it shows a violation warning. This is expressed in the following:

```
Get classes;
Get attributes;
If attribute=null
    Get associationEnds;
    If associationEnd=null;
        Show error;
Exit;
```

Rule 9 Object ID's must not be modeled as attributes.

In UML, classes are a group of objects with the same attributes and behaviors. In reverse, objects are instances of classes. How can we distinguish different objects of a

class? Some people may think of creating an objectID attribute. For example, in the 'Book' class, different book objects can be identified by a bookID attribute, whose values are 'book1', 'book2', and so on. However, "In our ontology, things are identified through their unique set of property values and there exists no special identification criterion or identifier" (Everman, 2003, p53). Now, we explain this in detail.

Even though objects in a class own the same group of attributes the attribute values of different objects are different. The attribute groups with different values are used to distinguish different objects in a class. The 'Book' class may own attributes of 'title', 'author', and 'year'. {UML1.5, OMG, 2003}, {UML1.4, OMG, 2001}, and {Ontology, Bunge, 1977} are three different book objects.

To implement this rule, the examination checks class attributes to see whether there is any attribute name space that includes 'ID'. However, the 'objectID' in this rule does not only specify the string 'ID'. It applies to all attributes, which are used for identification. Thus, we also check strings starting with or ending with 'Number' and 'No.'. It is obvious that this approach is not very accurate. There may be attributes including 'ID', 'Number', or 'No.' in their names are not for purpose of object identification. In this situation, our approach asks users whether attribute 'A' only valid within the class 'B'? This is because, if an attribute is only used for distinguishing different objects in a class, this attribute will be useless in any other domain out of this class. For example, from the perspective of a class diagram with several classes, an attribute 'author: Bunge' can give some information,

while an attribute 'bookID: book1' makes no sense. If the user indicates that the attribute is only valid within that class, the program shows a violation warning and saves this attribute in the 'checkedBad' vector. Otherwise, the program only saves this attribute in the 'checkedOk' vector. This is expressed in the following:

```
Get classes;
Get attributes;
While attributes.hasNext();
    Get name;
    If (name.startWith(ID||No.||Number) ||name.endWith(ID||No.||Number))
        If  !(name in checkedOk)
            If (name in checkedBad)
                Show error;
            Else
                Query user;
                If answer=Y
                    Show error;
                    Add name to checkedBad;
                Else if answer=N
                    Add name to checkedOk;
    Exit;
```

There are two aspects of limitation in this implementation approach. One is that, strings include 'No.' or 'ID' may not really be an object ID. In this case, the examination will show an incorrect violation warning. The query structure in our approach can reduce this effect to some extent. Another is that a designer may not use strings including 'No.' or 'ID' to represent an object ID. In this case, the examination will not detect the violation. A possible solution may be adopting a dictionary to list all words representing ID. In a specific domain, we can even use some glossary related to that domain.

Rule 10 The set of attribute values (representing mutual and intrinsic properties) must uniquely identify an object.

In the last rule, we have explained that different objects in a class are identified by different values of the attribute set. That means that if every attribute value in two objects of a class is the same, respectively, the two objects are identified as a same object. For example, if two book objects are {UML1.5, OMG, 2003} and {UML1.5, OMG, 2003}, they are actually the same book object.

The implementation approach is to compare values of each attribute of each object. If two objects have absolutely the same attribute value set, the program shows a violation warning. This is expressed in the following:

```
Get objects;
Set same=1;
While objects.hasNext()
    Get attributes;
    While attributes.hasNext()
        Get value;
        If value1!=value2
            Set same=0;
If same==1
    Show error;
Else exit;
```

Now, we talk more about identification of each object in a class. In UML, "An object is an instance that originate from a class" (OMG, 2003, p2-101). However, in conceptual modeling, we do not identify individual objects of a class, but only focus on classes. Thus, Rule 10 and other related rules, for example Rule 11, are not applicable to conceptual

modeling.

Rule 11 Every attribute has a value.

It is stated that "Since attributes represent properties by being assigned some value at some time, these values reflect a property"(Everman, 2003, p55). That is, there cannot be any attribute without a value. In UML, attributes can have multiplicities. That is, an attribute can have one value, two values or n values. However, there are two situations contradicting with this rule. First, a user can choose attribute multiplicities as '0.' This means that the number of this attribute value is '0.' The second is that a user can specify the attribute value as 'null.' Both these two cases imply that an attribute can have a meaningless value. These are inconsistent with this rule. In programming, sometimes we need to set 'null' values to attribute. For example, the 'age' attribute of a 'Person' class. When we set the variable, we do not know what value this variable would be and set 'null' for its initial value. However, this will not happen in conceptual modeling because everybody has an age. Thus, every attribute has a value. We will talk about this in Corollary 10.

To implement this rule, we just proscribe a value of "null" and multiplicity of "0" for attributes. According to UML specification, "The second compartment (of object notation) shows the attributes for the object and their values as a list. Each value line has the syntax: *attributename: type = value*" (OMG, 2003, p3-65). Thus, the implementation affects two

diagrams: class diagram and object diagram. In class diagrams, we proscribe 'null' for an initial value. In object diagrams, we proscribe 'null' for the attribute list. We still need to remind users to specify values for attributes. The program simply gets all attributes. If any attribute is found without a value, it shows a violation warning. This is expressed in the following:

```
Get class/object;
Get attributes;
Get value;
If value=null
    Show error;
Else exit;
```

Corollary 10 Attribute multiplicities greater than one imply that the order of the different individual attribute value components is semantically irrelevant.

Attribute multiplicity specifies the number of values of the attribute. The last rule prohibits attribute multiplicities of '0', here we will discuss attribute multiplicities greater than '1.' When the multiplicity is greater than one, it means that the attribute has more than one attribute value. This corollary specifies that only order irrelevant values can be modeled as attribute multiplicities greater than one. For example, an address in the real world may consist of 'number', 'street', 'city', and 'country.' One instance would be {123, Cook St., Victoria, Canada}. A user may model the 'address' as an attribute with a multiplicity of '4.' According to this corollary, it is incorrect, because the order of these four values is relevant. If we change the order to {Victoria, Cook St., Canada, 123}, this is

obviously an invalid address. In this case, 4 attributes with multiplicity of '1' will be modeled instead of 1 attribute with multiplicity of '4.' That is, attributes of 'number', 'street', 'city', and 'country' should be modeled.

The above discussion is about the definition of attribute multiplicity in UML specification and Everman's research. However, we feel the above (un)ordered attribute multiplicity structure confusing and redundant. To simplify this, we propose our approach, which is to model an attribute for each attribute value. For the 'address' attribute, a much simpler modeling is to decompose it into four attributes of 'no.', 'street', 'city' and 'country.' In this way, every attribute only has a single value and we need not think about multiplicity and ordered or unordered. The above is for the ordered values case; for unordered values, this modeling also works fine. For example, in the current UML structure, a 'contactNo.' attribute may have multiplicity of '3.' A 'contactNo.' may have three values, such as 'homePhone, cellPhone and fax.' In this case, we can also decompose the 'contactNo.' into three attributes of 'homePhone', 'cellPhone' and 'fax.' People may ask when some 'Persons' have 'cellPhone', while others have not, can we assign null to the attribute 'cellPhone'? The answer is no. In the last rule, we have discussed that an attribute cannot be null. However, it is a fact that some people do not have a cellPhone. In this case, we say that people with 'cellPhone' possess this additional attribute than those who do not have 'cellPhone.' These people should be specialized from regular people. According to the above discussion, we propose Rule-New 01 to take the place of Corollary

10. Thus, there is no need to implement Corollary 10.

Rule-New 01: Every UML attribute can only have a single value. An attribute of multiplicity of 'N' should be decomposed into 'N' attributes of multiplicity of '1'. An attribute multiplicity of '0' should be generalized as a super-class not possessing the attribute.

Even through this rule strictly follow ontology, it has a severe restriction making this new rule a little difficult to be satisfied. For the implementation of this rule, UML specification as well as CASE tools are proposed to disable the feature of assigning multiplicity to attributes. Thus, there is no need to implement it.

Now, we discuss a little more about order and unordered attributes. Actually, ordered attributes depend on each other, while unordered attributes are independent. For example, attributes 'street', 'city', 'province', and 'country' can only specify an address when they are together. A independent 'street' makes no sense. Attributes 'homePhoneNumber', 'businessPhoneNumber', and 'cellPhoneNumber' can give information independently. However, these are out of the scope of our thesis, future research can be done about it.

Rule 12 Classes of objects that exhibit additional behavior, additional attributes or additional association classes with respect to other objects of the same class, must be modeled as specialized sub-classes.

We have discussed multiplicity of attributes. In this rule, we will discuss multiplicity of association ends. Since associations connect classes, the multiplicity of association ends specifies the number of class objects participating in an association. For example, in Figure 4.10, a company can hire zero or more persons while a person can work for one and only one company. Here, zero means the company does not hire any person.

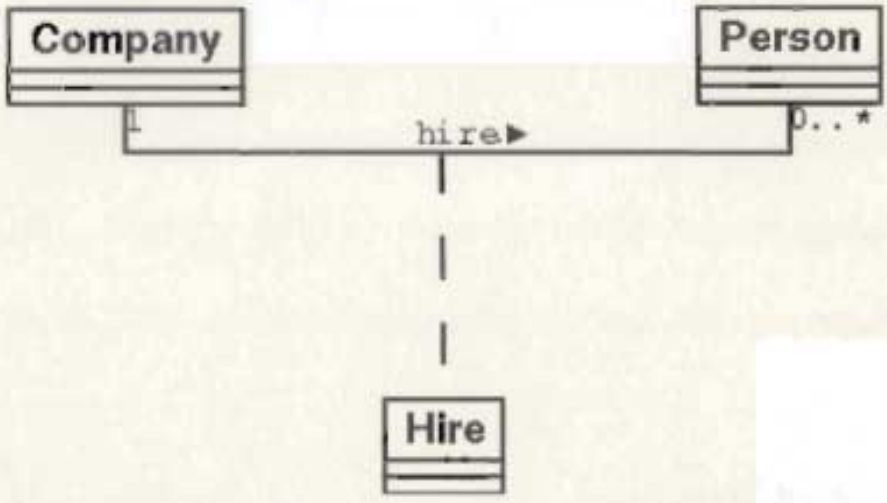

Figure 4.10 Multiplicity of association ends

Consider a situation where the company hires one or more persons. These persons share the same attributes and behaviors, so they are modeled as a person class. When the multiplicity is zero, it means that no person participates in this association. That is, a person does not participate in this hire association. In this case, the person does not possess the attributes and behaviors of an 'employee', such as 'skill' and 'fire().' In other words, the person class exhibits additional attributes or behaviors when multiplicity changes from '0' to '1..*.' In UML, "A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics" (OMG, 2003, p2-26). The

'employee' exhibits more behaviors and attributes than the 'person.' They should be separated into two classes instead of modeling them in one class. Considering that all employees are persons and inherit all attributes and behaviors of person, the 'Employee' is modeled as the sub-class of 'Person.' Thus, the 'Person' class does not participate in the 'hire' association, but the 'Employee' class does. Generally speaking, all classes with multiplicity of '0..*' (* = 1, 2, ...N) participating in an association have the problem discussed above. Thus, association ends with multiplicity of '0..*' should not be modeled. Objects exhibiting additional attributes or behaviors actually participate in this association. They should be modeled as a specialized sub-class. This rule can be depicted in Figure 4.11 and Figure 4.12.

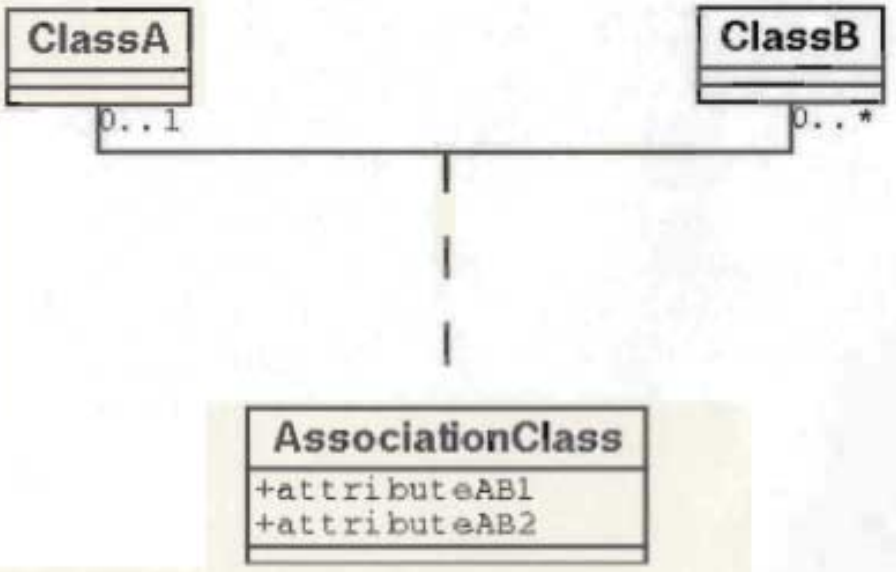

Figure 4.11 Incorrect model of Rule 12

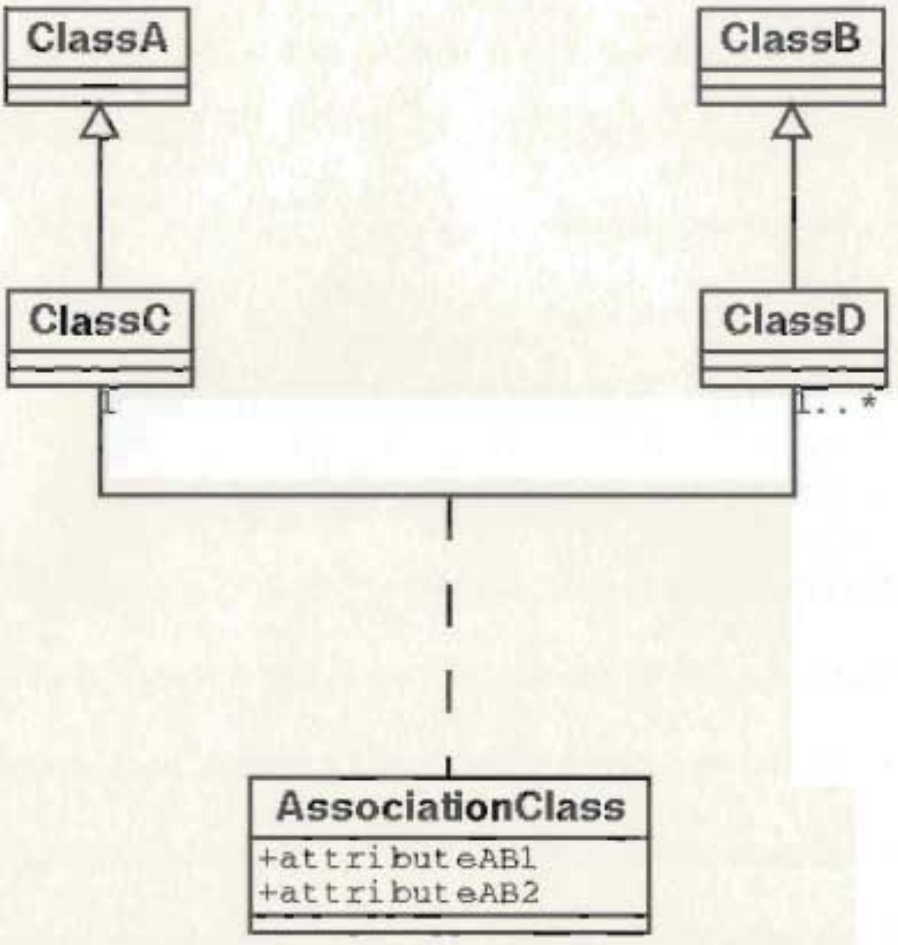

Figure 4.12 Correct model of Rule 12

The implementation of this rule applies to both UML specification and CASE tools. In UML specification, multiplicity of '0..*' (* = 1, 2, ...N) for association ends should be proscribed. That is, users should separate '0..*' into '0' and '1..*.' However, "A multiplicity of 0..0 is meaningless as it would indicate that no instances can occur" (OMG, 2003, p3-75). Thus, the multiplicity of '0' for association ends should not be modeled and the option of multiplicity of '0' should be disabled in CASE tools. In the current stage, if a CASE tool permits users to input association end multiplicity from the keyboard, an examination is still needed. The program gets association ends and checks whether the

multiplicity begins with '0.' If this is the case, it shows a violation warning. This is expressed in the following:

```
Get associations;
Get associationEnds;
Get multiplicity;
If multiplicity.startsWith("0")
    Show error;
Else exit;
```

Corollary 11 An object acquiring additional behavior or properties must be destroyed as instance of the general class and created as instance of the specialized class that is modeled with the relevant operations or association classes.

The last rule describes a situation where objects acquire additional behaviours or additional attributes. It also gives a solution for when this happens. That is, when an object acquires additional behaviors or properties, it should no longer be modeled in the original class, but to be modeled in a specialized sub-class. That means that the object is reclassified from one class to another class. However, there is no construct in UML to model this. Evermann and Wand suggest a mechanism of re-classification to make up such a UML deficiency. They propose to employ the UML semantics of object creation and destruction (Evermann, 2003, p59). That is, when objects acquire additional behaviors or properties, the objects should be destroyed as instances of the original class and created as instances of the newly specialized class.

With respect to Rule 12, this corollary specifies another solution to model objects

acquiring additional behaviours or additional attributes. However, "UML does not provide a construct to express this. In this respect it is ontologically deficient" (Evermann, 2003, p59). Thus, a formal construct of re-classification to express this needs to be defined. However, "Object creation and destruction have no direct equivalent in the BWW-ontology as things cannot be created or destroyed" (Evermann, 2003, p61). Actually, objects cannot be created or destroyed. Instead, they only change memberships of classes by acquiring or losing properties. For example, a 'Person' acquiring a mutual property of 'enrol' by a 'University' is a 'Student.' In this case, there is not a new 'Student' created, but the 'Person' instance becomes a member of the 'Student' class. We would not consider it a good way to employ a non-ontological semantic (Object creation / destroying) for an ontological deficiency (no mechanism of re-classification), which actually need a formal definition. The original and basic motivation for this corollary is to model the process of an object changing its membership from a class to its sub-class, namely re-classification. Thus, the mechanism needs formal definition and further refinement so that it can be added to UML specification. Since there is no such mechanism in UML specification currently, we could not implement this corollary in CASE tools at this time.

Corollary 12 Re-classification occurs only within a generalization / specialization hierarchy.

Re-classification is a mechanism to model the process of objects being specialized or generalized through acquiring or losing properties or behaviours. When objects acquire additional behaviours or attributes, they possess the original behaviours or attributes as well as the additional behaviours or attributes. According to UML specification, "The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information" (OMG, 2003, p2-38). Objects acquiring additional behaviours or attributes are specialized from the original class. That is, re-classification occurs within a specialization hierarchy. For example, a 'Person' acquiring an additional attribute of 'department' is re-classified as an 'Employee.' This re-classification occurs only within the 'Person---Employee' specialization hierarchy. Conversely, an 'Employee' losing the attribute of 'department' is re-classified as a 'Person.' This re-classification occurs only within the 'Employee---Person' generalization hierarchy.

To implement this corollary, first, we need to get the re-classification. This can only be done after re-classification is fully defined in UML specification. Then we can check whether the re-classification only occurs within a generalization / specialization hierarchy, that is, check the new class and original class, which the re-classified object belongs to. If the two classes have no generalization / specialization (direct or indirect) association then the program shows a violation warning. This is expressed in the following:

Get re-classification;

```
Get object;
Get originalClass;
Get newClass;
Object interClass = newClass;
While interClass.associationEnd=generalize
    Get anotheEnd;
    Get anotherClass;
    Interclass = anotherClass;
    If interclass == originalClass
        Exit;
Show error;
```

Because there is no re-classification mechanism either in UML specification or in UML CASE tool, we cannot implement this corollary at this time.

Rule 13 Every UML-aggregate object must consist of at least two parts.

This rule discusses UML aggregations. First, we explain the meaning of this rule. Because objects are instances of a class, this rule specifies that every UML-aggregate object must consist of at least two parts. There are two cases that apply to this rule. First, an aggregate object should consist of least two objects in different part classes. For example, a 'Committee' class is the aggregate of classes of department faculties. The object of 'Committee' class can consist of an object of faculty from 'Computer Science', and an object of faculty from 'Business.' The second case is that an aggregate object can consist of at least two objects from one part class. In this case, the number of objects in that part class must be greater than one. For example, the object of the 'Committee' class can consist of two faculty objects from 'Computer Science.' That is, the committee can consist

of both two people from computer science; or one person from computer science and another from business; or both two people from business. Now, let us see why we should have this rule. We have explained in Rule 6 and Rule 7 that aggregation is a kind of association related to the whole-part relationship. Aggregation classes are the whole of some part classes. If an aggregation only consists of one object, how can the aggregation acquire emergent properties? According to Rule 7, every aggregate has to have at least one emergent property. Thus, it is better to equate this aggregation to that object than say that it is an aggregation consisting of one object.

We can calculate the number of part classes to make sure the number is greater than two. However, in the second case, the part class number would only be one. Thus, we calculate the number of aggregate associations. If the number is greater than one, the program does nothing and exits. Otherwise, it shows a violation warning. However, in the above approach, the system will show a warning message for all unrelated classes. This is because the unrelated classes have no aggregate association at all and the count number is zero. Thus, we add another restriction in the program. Only if the aggregate association number of a class is less than two and does not equals zero, the system shows a violation warning. This is expressed in the following:

```
Get classes;
Get associationEnds;
Count=0;
While associationEnds.hasNext()
    If associationEnd=aggregate
```

Count+1
If count>1 or count=0
Exit;
Else show error;

Rule 14 An instance of a class that by virtue of additional aggregation relationships acquires emergent properties or emergent behavior must be modeled as an instance of a specialized class which declares the corresponding attributes and operations.

This rule is very similar to Rule 12, which discusses objects acquiring additional properties or behaviors through association interactions. This rule discusses objects acquiring emergent properties or behaviors through a special association: aggregation. Thus, this rule has the same motivation and rational as Rule 12. From Rule 7, we know that objects can acquire emergent properties by participating in an aggregation as a whole side. Conversely, if objects do not participate in this aggregation, they do not possess these emergent properties. This difference is expressed by aggregate multiplicity of '0..*' (* = 1, 2, ...N). We use Figure 4.13 and Figure 4.14 to explain this.

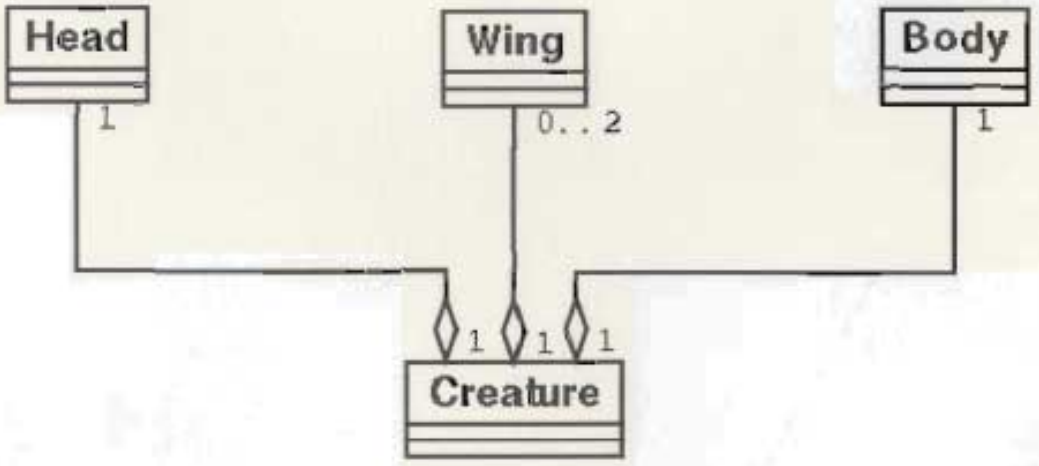

Figure 4.13 Incorrect model of Rule 14

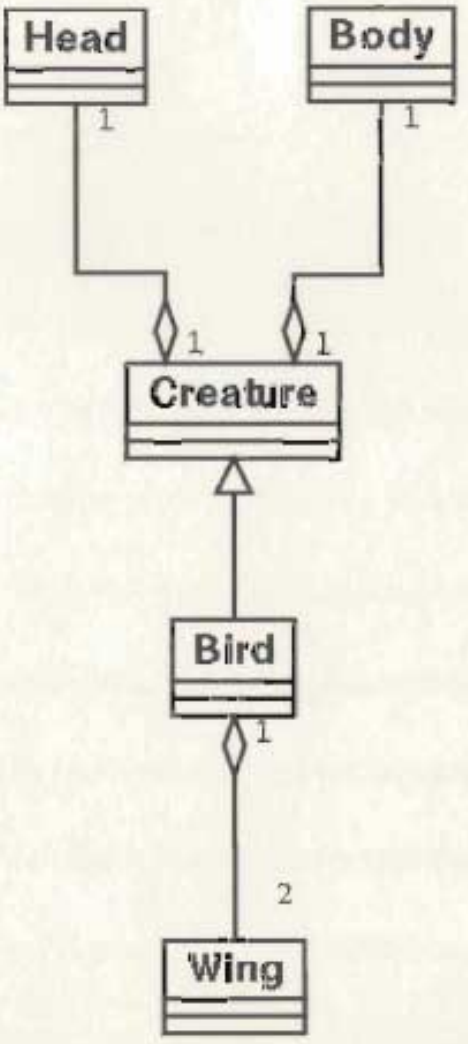

Figure 4.14 Correct model of Rule 14

A 'Creature' is an aggregate of 'Head', 'Body' and so on. In Figure 4.13, if no 'Wing' (multiplicity of '0') participates in the aggregation, it is fine. When a 'Creature' has two 'Wings' (multiplicity of '2'), it acquires an emergent behavior of 'Fly.' However, not all 'Creature' objects possess the behavior of 'Fly.' This conflicts with the definition of class, which is a group of objects with the same properties and behaviors. Thus, Figure 4.13,

which violates this rule, is an incorrect modeling. According to this rule, we model the objects, that acquire emergent property of 'Fly' by virtue of 'Wing' aggregation, as a specialized 'Bird' class in Figure 4.14.

The implementation of this rule also applies to both UML specification and CASE tools. In UML specification, multiplicity of '0..*' (* = 1, 2, ...N) for aggregation ends should be proscribed. That is, users should separate '0..*' into '0' and '1..*.' However, "A multiplicity of 0..0 is meaningless as it would indicate that no instances can occur" (OMG, 2003, p3-75). Thus, the multiplicity of '0' should not be modeled for aggregation ends. Remember that multiplicity of '0' should not be modeled for attributes (Rule 11) and multiplicity of '0' should not be modeled for association ends (Rule 12). We propose that multiplicity of '0' should not be modeled at all in UML specification. For the same reason, the option of multiplicity of '0' should be disabled in CASE tools. However, if a CASE tool permits users to input aggregation ends multiplicity, an examination is still needed. The examination gets aggregation ends and checks whether the multiplicity begins with '0.' If this is the case, the program shows a violation warning. This is expressed in the following:

```
Get aggregations;
Get aggregationEnds;
Get multiplicity;
If multiplicity.startsWith("0")
    Show error;
Else exit;
```

Rule 15 Object creation occurs when an entity acquires a property so that it becomes a member of a different class.

According to class definition, "A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics" (OMG, 2003, p2-26). If an entity acquires a property other than the 'n' properties of the class it currently belongs to, it can no longer belong to this class. It does not only possess the 'n' attributes possessed by other objects in that class, but also posses one (the 'n+1') more attribute than other objects. In this case, the entity becomes a member of a different class whose members all possess these 'n+1' attributes. In the 'Creature—Bird' example, a 'Creature' acquires a property of 'Wing' so that it becomes a member of the 'Bird' class.

This rule specifies the use of object creation to model the above process. However, "Object creation and destruction have no direct equivalent in the BWW-ontology" (Evermann, 2003, p61). In addition, we notice that the above process is like the following description: "Changes in natural kind of a thing correspond to changes in class membership of an object" (Evermann, 2003, p59). This description is actually the proposed re-classification. (Please refer Corollary 11 for details of re-classification.) Based on the above thinking, we would rather employ the re-classification mechanism than object creation to model the process that entities acquire properties so that it becomes a member of a different class.

This rule will change the UML specification. Implementation in CASE tool will only

be realized after this is done in UML specification.

Corollary 13 Object destruction occurs when an entity loses a property that is necessary for membership in a particular class.

This corollary describes the reverse process of Rule 15. An entity of 'n' properties loses a property and becomes to possess "n-1" properties. This entity no longer belongs to that 'n' property class, because this entity no longer share all the 'n' properties with other objects in that class. For example, when the 'Employee' loses the property of 'job', it no longer belongs to the 'Employee' class. This is because 'job' is necessary for membership of the 'Employee' class.

This corollary specifies the use of object destruction to model the above process. Based on the same thinking in Rule 15, we would rather employ the re-classification mechanism than object destruction to model the process that entities lose properties. These properties are necessary for membership of a particular class.

This rule will change the UML specification. Implementation in CASE tools will only be realized after this is done in UML specification.

Rule 16 Attributes with class scope should instead be modeled as attributes of an aggregate representing the objects of the class.

In UML classes, there are two kinds of attributes: instance-scope attribute and

class-scope attribute. The instance-scope attribute is an attribute of the instance objects of a class. A class is a set of objects with the same attributes and behaviors. Objects are instances of a class. So, every object of a class possesses the instance-scope attributes. Attributes we normally model are the instance-scope attributes. However, not all attributes can be possessed by each object. Some attributes are only possessed by the whole group of objects, namely the class. These attributes are class-scope attributes. This rule specifies that no class-scope attribute should be modeled. This is because "An object as a class instance possesses all methods and attributes defined for its class" (Evermann, 2003, p52); however, class-scope attributes are not possessed by a class instance, but by the set (aggregate) of all class instances. Let us look at an example. In the modeling of a bookstore, the designer models a 'Book' class with attributes: 'BookName', 'NumberOfBooks' and so on. This is an incorrect model. The 'BookName' is an instance-scope attribute because every book instance possesses a name. The 'NumberOfBooks' is a class-scope attribute because each book instance cannot possess the book number but only the book class can. According to this rule, the 'NumberOfBooks' should not be modeled as an attribute of the 'Book' class. Instead, we should create an aggregation relationship in this case. The 'Book' is the part class and 'Inventory' is the whole class. The 'NumberOfBooks' should be modeled as an attribute of the aggregated class 'Inventory.' This is because on the book inventory, every book title possesses the attribute of book number and book number is the instance-scope attribute of

the inventory class.

For the implementation, the program gets each attribute from each class to see whether any attribute is class-scope. If an attribute is found class-scope, it shows a violation warning. This is expressed in the following:

```
Get classes;
Get attributes;
While attributes.hasNext()
    Get attributeScope;
    If attributeScope=classScope
        Show error;
    Else exit;
```

Rule 17 If a class that is specialized is declared as abstract, the specialization must be declared to be 'complete'.

In UML, specializations inherit all attributes and behaviors from their generalization. Generalization is the super-class of all their specialized sub-classes. If all objects in generalized super-class are members of the set of specialized sub-classes, this is called "complete". Another concept we need to introduce is that a class can be declared "abstract". This means, there is no actual instance for the abstract class. A generalized super-class is a class, so it can be declared abstract, which means there is no instance for the super-class. That implies an abstract super-class has attributes and behaviors and there is no class object to possess them. In addition, "There are no properties without things possessing them and there are no laws without things adhering to them" (Evermann, 2003,

p65). Thus, the abstract super-class has to be specialized by its sub-classes so that objects of the specialized class can possess those attributes and behaviors. Since the super-class is abstract and cannot have any object, the specialization has to be complete. For example, 'GraduateStudent' and 'UndergradStudent' are two specialized sub-classes of 'Student' class. If the super-class 'Student' is declared as abstract, it means, there is no object for it. The actual student has to be member of 'GraduateStudent' or 'UnderagradStudent.' Since 'Student' is abstract, this specialization must be complete. That is, a student must be either a gradstudent or undergradstudent. Note that specializations can also be declared abstract. In this case, their sub-specializations should be complete.

To implement this rule, we need to first check all generalized super-class. Then these super-classes are checked to see whether they are abstract or not. For those super-classes, which are declared abstract, we finally check the specialization constraints to see whether it is declared complete. If the program finds a specialization not complete, it shows a violation warning. This is expressed in the following:

```
Get generalizations;
Get parentEnd;
If parentEnd==abstract
    Get constraints;
    If !constraint==complete
        Show error;
Exit;
```

However, the notion of abstract class in UML refelects implementation thinking, not conceptual thinking. We propose that this notion be removed from UML for conceptual

modeling.

Rule 18 A class that is not specialized cannot be declared abstract.

Having "There are no properties without things possessing them and there are no laws without things adhering them" (Evermann, 2003, p65), we can deduce this rule in two ways. First, a class is not specialized implies that this class has no sub-class. That means that all objects are instances of the class itself. Also beacuse there must be some objects to possess the class properties and behaviors, this class cannot have no instance and cannot be declared abstract. Second, a class that is declared abstract implies that this class has no object. Thus, the abstract class has to be realized by some specialized sub-classes. In addition, the specialization has to be declared complete according to Rule 17. For example, a 'Student' class without any specialized sub-class cannot be declared abstract.

To implement this rule, we need to check whether a class is abstract or participates in generalization as a parent. If it is abstract but does not satisfy the latter condition, the program shows a violation warning. This is expressed in the following:

```
Get classes;
If class==abstract
    Get associationEnds;
    While associationEnds.hasnext()
        If associationEnd==generalizeParent
        Exit;
    Show error;
```

Rule 19 A specialized class must define more attributes, more operations or participate in more associations than the general class.

According to UML specification, "Generalization is the taxonomic relationship between a more general element (the parent) and a more specific element (the child) that is fully consistent with the first element and that adds additional information" (OMG, 2003, 3-86). That is, the features of a specialized sub-class consist of two parts. One is the features inherited from its super classes, another is the additional features only possessed by itself. If the specialized class does not define more attributes, more operations or participate in more associations than the general class, the specialized class is absolutely the same as the general class. Thus, a specialized class must define more features than the general class. For example, a specialized sub-class 'Student' defines more attribute: 'studentNo,' more operation: 'study,' and participates in more association: 'School enroll student' than the general super-class 'Person.'

The implementation approach of this rule is to compare specialized sub-class with general super-class to see whether the sub-class possess more features. First, we get generalizations as well as both "parentEnds" and "childEnds." Second, the program gets all parentEnd features, such as attributes, methods and associations and puts them into a vector. Then, each attribute, method or association of the childEnd will be matched with the vector. If any feature is not found in the vector, the system exits. Otherwise, it shows a violation warning. This is expressed in the following:

```
Get associations;
If association==generalization
    Get parentEnd;
    Get attributes;
    Get operations;
    Get associationEnds;
    Add to vector;
    Get childEnds;
    Get attributes;
    While attributes.hasNext()
        Match with vector;
        If not find in vector
            Exit;
    Get operations;
    While operations.hasNext()
        Match with vector;
        If not find in vector
            Exit;
    Get associationEnds;
    While associationEnds.hasNext()
        Match with vector;
        If not find in vector
            Exit;
    Show error;
Else exit;
```

Noticing that the inherited attributes, operations or associations from the super-class should not be shown in the sub-classes, we only need to check whether a sub-class possesses any attribute, operation or participates in any association except the specialization itself. This is because the super-class should not possess any of the above features. If the program cannot find any feature of attribute, operation or participating in association for a sub-class, it shows a violation warning. This is expressed in the following:

```
Get associations;
If association==generalization
    Get childEnds;
    Get attributes;
    While attributes.hasNext()
        Exit;
    Get operations;
    While operations.hasNext()
        Exit;
    Get associationEnds;
    While associationEnds.hasNext()
        Exit;
    Show error;
```

Rule 20 Every ordinary association must be an association class.

In Rule 3, we discussed that mutual properties of classes participating an association should be modeled as attributes of association classes. For classes that have no mutual property or their mutual properties are out of our modeling scope, we only model an association to connect these classes. In UML, there are two functions of associations. One use of association is message passing. It is stated that "The link is used for transportation of the stimuli" (OMG, 2003, p3-130), "A link is an instance of an association" (OMG, 2003, g9), and "A message is a specification of a stimulus" (OMG, 2003, p3-111). However, message passing is not consistent with our ontology because "Since message passing is a design related concept and there exist no equivalent in the BWW-ontology, we propose that associations are ontologically excessive. Hence, they should not be employed for conceptual modeling" (Evermann, 2003, p67). Another situation of modeling

associations is that there is a mutual property of participated classes. However, if there are more than one mutual property, an association class is employed. That is, "the same ontological concept, mutual property, would be mapped to two different UML constructs, attributes and associations" (Evermann, 2003, p67). To avoid ambiguities, this rule specifies that association classes should take the place of associations. Please note that associations we discuss here only refer to ordinary associations, excluding aggregations.

To implement this rule, we can change it to other words. That is, all associations should be modeled together with association class. The approach is to check whether there is any association without association class connecting to it. If the program finds an independent association not connecting to association class, it shows a violation warning. Because this rule only applies to ordinary associations, our program does not check generalizations and aggregations. This is expressed in the following:

```
Get associations;
If generalization||aggregation
    Exit;
Get associationClass;
If associationClass==null
    Show error;
Else exit;
```

Chapter 5 Change Rules

In the last chapter, we discussed the static structure of things and their related concepts. According to ontology, every thing can change. In this chapter we will discuss the approach of implementing rules related to change within things.

Rule 21 A UML-state represents a specific assignment of values to the attributes [of ordinary classes] *and attribute of association classes of the objects for which the state is defined.*

According to BWW ontology, a thing can be represented by state functions, whose values are determined by properties of the thing (Bunge, 1977, p126). That is, states of things associate with properties of things. We have discussed before, that each property has its value. The different property value sets correspond to different states. In Evermann and Wand's research, UML-states and UML-state transitions are mapped with BWW-states and BWW-state transitions, respectively. Also, "There exist no states which are independent of attributes because properties express all the characteristics of a thing" (Evermann, 2003, p71). In addition, properties consist of intrinsic properties and mutual properties, which are represented by attributes of ordinary and association classes

respectively. Thus, we get this rule. For example, a 'Person' object has an attribute of 'location.' Different values of 'location' specify different states of the 'Person.' A value of 'office' implies state of 'work'; a value of 'bus' implies state of 'go home'; and a value of 'home' implies state of 'rest.' (Note that 'work', 'go home', and 'home' are high-level composite states. Our example model only concerns this level.)

However, there is no mechanism in UML mapping with ontology in this case: "States in UML are independent of attributes or properties and UML provides no mechanism with which to specify any such connection" (Evermann, 2003, p72). Thus, this rule can be realized by enabling this feature in both UML specification and CASE tools. That is, to implement this rule some additional mechanism needs to be added to UML. CASE tools will enable some functions based on this mechanism. Because UML is ontologically deficient with respect to the rule, until such a connection is defined in UML, there is no way to implement it.

Corollary 14 A UML-transition must change the value of at least one attribute used to define the state space.

We have discussed in the last rule that different states of a thing are determined by different property value sets. This also means that the same property value sets correspond to the same state of a thing. In UML, "A transition is a directed relationship between a source state vertex and a target state vertex" (OMG, 2003, p2-149). A state of a thing

changed by transition indicates that property values are changed. For example, states of a 'Person' transiting from 'work' to 'go home' imply values of 'location' attribute changing from 'office' to 'bus.' However, we notice that other 'Person' attributes, such as 'height' remain invariant with this transition. There can be different modelings of a thing depends on the different purposes (Bunge, 1977, p119). In each modeling, the thing can have a state. Not all properties have to be used to define states in a model. For example, the attribute of 'location' is used to describe states of modeling for one purpose, while the attribute of 'height' is used to describe states of another modeling for another purpose. Values of 'location' only determine state space of {work, go home, home}; values of 'height' only determine state space of {child, youth, adult}. That is why this corollary only relates to the "attribute used to define the state space."

It is possible that the source and target are actually the same state. Same state means same attributes of this state space. If this is the case, the transition between the same states would not change any attribute value. This is inconsistent with this corollary. We revise this corollary by proposing a constraint to it. That is:

If source state and target state are not the same, a UML-transition must change the value of at least one attribute used to define the state space.

In the last rule, we have known that states and attributes of an object are independent (unrelated) in UML. Evermann proposes a meta-model in which UML-states should be associated with attributes. If this is adopted by UML, we can implement the corollary in

this way. Once the program finds a transition, it will check whether any attribute of that object changes. If the transition does not change any attribute, the program shows a violation warning. Otherwise, the program checks whether these changed attributes span the state space of that object. If not, it shows a violation warning. This is expressed in the following.

```
Get transition;
Get attributes;
If attribute change
    Get stateSpace;
    If stateSpace.contains(attribute)
        Exit;
Show error;
```

Rule 22 For every level of refinement of a state C, there must be an additional set of attributes in the class description or in participating association classes that change as the object transitions among the sub-states.

According to Evermann and Wand's research, "A composite state or submachine state may be refined as a state machine comprising sub-states and transitions among sub-states" (Evermann, 2003, p77). That is, designers can decompose a composite state into a set of sub-states. Whatever sub-state a thing is currently in, it remains in the same composite state of those sub-states. Thus, the attribute values determining the composite state also remain invariant. From Rule 21, we know that this set of sub-states is also defined by some attribute values. That is, a transition between the sub-states must change the value of at

least one attribute used to define the sub-state space (Corollary 14). What are these attributes used to define the sub-state? Because the sub-states are extended from a composite state, some attributes extended from the attributes defining composite state are used to define these sub-states: "Whenever we encounter sub-states, the set of attributes used to describe them must be extended from the set used to describe the super-state" (Evermann, 2003, p79). These extended-attributes are an additional set of attributes in the class description or in participating association classes that change as the object undergoes transition among the sub-states. For example, we refine the state 'rest' of a 'Person.' Sub-states of 'rest' may be 'eat', 'shower', and 'sleep.' The additional attributes describing these sub-states are extended from 'location.' Namely, an attribute of 'position' with values of {kitchen, bathroom, bedroom} describes the three sub-states respectively.

The implementation is similar to Corollary 14. The difference is this rule only checks composite super-states. The program gets all transitions among sub-states within a composite state. Because extended attributes are also attributes of the same object, they should be added to the same class / object. Thus the program checks whether any attribute of that object changes. If the transition does not change any attribute, it will show a violation warning. Otherwise, the program checks whether these changed attributes are in the state space of the composite super-state. If yes, it shows a violation warning. Since the top-level composite state represents direct attributes of a class and this rule discusses extended attributes, we do not check such top-level composite states. This rule also needs

to use the proposed meta-model discussed in Corollary 14. This is expressed in the following:

```
Get compositeState;
If compositeState==top
    Get sub-states;
    If sub-states.size>1
        Get transition;
        Get attributes;
        If attribute not change
            Get superStateSpace;
            If superStateSpace.contains(attribute)
                Show error;
Else exit;
```

Corollary 15 For all immediate sub-states of a super-state, the values assigned to attributes describing the super-state are invariant and are equal to those defining the super-state.

Since all immediate sub-states of a super-state share the same super-state, the super-state remains invariant when transitions happen among sub-states. That means the values assigned to attributes describing the super-state are invariant and are equal to those defining the super-state. It is conform with the conclusion we made in Rule 22.

The implementation of the last rule has made sure the transitions among sub-states would not change any attributes of their composite super-state. If users follow Rule 22, they will not violate this corollary. Therefore, there is no need to implement it.

Corollary 16 Concurrent sub-states require mutually disjunct sets of additional attributes in the class description or in participating association classes.

We have known from Rule 22 that for sub-states there must be an additional set of attributes in the class description or in the participating association classes that change as the object experiences transition among these sub-states. Now, we will talk about concurrent sub-states. In UML, the concurrent sub-state is defined as "a sub-state that can be held simultaneously with other sub-states contained in the same composite state" (OMG, 2003, glo-5). That is, a thing is in 'N' sub-states at the same time. Since different attribute values determine different states, an attribute can only possess one value for one state. Otherwise, it specifies other states. That means it requires 'N' attributes to specify 'N' values to describe the 'N' sub-states. For example, there can be several concurrent sub-states describing a 'Person' in a state of 'rest'. He can 'eat' and 'think' at the same time. However, state 'eat' is determined by the 'position' attribute with the value of 'kitchen.' There needs to be another attribute to describe the 'think' state. An attribute of 'plan' with the value of 'go to cinema' can be the solution.

To implement this corollary, the program first gets a sub-state. If it has concurrent states, both of their related attributes will be examined. If attributes related with these concurrent sub-states are the same, the program shows a violation warning. This rule also needs to use the proposed meta-model discussed in Corollary 14 (Please refer to discussion of Corollary14 for details). This is expressed in the following:

```
Get subStates;
If subState==concurrent
Get ssAttributes;
While attributes.hasNext()
    Get concurrentState;
    Get csAttribute;
    If ssAttribute!==csAttribute
        Exit;
Show error;
```

Rule 23 Guard conditions on transitions from the same state to nonconcurrent sub-states must be mutually disjunct.

It is stated that "Ontologically, a thing can be in only one current state in any given model" (Evermann, 2003, p83). When we map this into UML, some constraints are required to ensure that an object will not go into two states at the same time. In UML, transition is "A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied" (OMG, 2003, glo16). States change from one to another through transitions. That is, there can be at most one transition fired from the same state. Also, transitions can be enabled or disabled via guard conditions: "Guard condition is a condition that must be satisfied in order to enable an associated transition to fire" (OMG, 2003, glo8). To avoid the transition of objects from one state to more than one state, we need to prohibit more than one guard condition being satisfied. That is, guard conditions on transitions from the same state must be mutually disjunctive and not the

same. However, in the discussion of Corollary 16 we gave an example of a 'Person' in the 'eat' and 'think' states at the same time. That example does not violate this ontological mapping. When we began discussing this rule we specified that a thing can be in only one current state in any given model. A thing can be in 'N' states concurrently in 'N' different models. The purpose of the 'eat' state is to capture the physical status of what a 'Person' is doing. The 'think' state is for purpose of modeling the mental activity of that 'Person.' They are two concurrent sub-states in two given models. Thus, this rule only applies to nonconcurrent cases.

To implement this rule, we need to check with state chart diagrams. First, the program checks whether a state has more than one outgoing transition. If this is the case, it then checks whether these transition targets are concurrent sub-states. For those which are not concurrent, we check whether the guard conditions of these transitions are the same. If they are the same, a violation warning is shown. This is expressed in the following:

```
Get states;
Get transitions;
While transition.hasNext()
    If transition== outgoing
        Get target;
If targets==concurrent
    Exit;
Get guards;
While guards.hasNext()
    If vector.contains(guard)
        Show error;
    Else add to vector;
```

There is a limitation of this implementation approach. In some cases where more than

one guard condition is satisfied, the examination will not show a violation warning. For example, guard1 is 'X>3' and guard2 is 'X>5.' If in some cases 'X=6', both guard1 and guard2 are true and the object goes into two states. That is because, if there is overlap of two guards, they are not disjunct. In this case, the program cannot accurately tell whether two guards are strictly disjunctive. The reason is that the firing of transitions depends on the boolean values of guard conditions. However, different guard expressions may result in a same boolean value. There is no general method for the program to tell whether two different expressions always result in the same boolean value. However, for some specific domains, we can find solutions. In the above example, if 'X' is an integer and has a range of '-50…+50', we can let the program test each value of 'X.' The program will easily find that guard1 and guard2 are not mutually disjunctive.

Rule 24 Action states are super-states of a set of sub-states. The object transitions among these while in the action state. State charts must reflect this fact.

In UML, there is a notion of action states: "An action state is a simple state with an entry action whose only exit transition is triggered by the implicit event of completing the execution of the entry action" (OMG, 2003, p2-172). That means, in the whole process (from entry to exit) of an action, an object is always in one state. In the discussion of Rule 21, we know that a state represents specific attribute values. That is, in the whole process of an action, attribute values determining the action state remain invariant. In UML, "An

action typically results in a change in the state of the system, and can be realized by sending a message to an object or modifying a link or a value of an attribute" (OMG, 2003, g2). Thus, Evermann and Wand map a UML-action to the ontological concept of a BWW-state transition (Evermann, 2003, p120, We will discuss this in detail in Corollary 31.). In addition, "BWW-state transitions are mapped to UML-state transitions" (Evermann, 2003, p72). Futhermore, because of Corollary 14 (A UML-transition must change the value of at least one attribute used to define the state space), we can conclude that an action changes attribute values. However, this contradicts with the UML notion of action state described above.

This contradiction can be explained by the fact that action states are in fact not simple states, but composite states. When an action happens, namely a transition happens, it changes the attribute values and consequently changes the object states. However, the action does not change the composite super-state, but those sub-states. The unchanged state is actually a super-state. Let us see the example of a 'Person' in the action state of 'go home.' In the action from the person leaving office to arriving home, s/he is continually in the super-state 'go home.' However, its sub-states are changing. The person experiences sub-states of 'in elevator', 'on bus', 'at door' and so on. According to the above analysis, action states should be specializations of composite states. However, in UML action states are specializations of simple states. Thus, this rule indicates such a deficiency of UML. If it is adopted, the UML specification should be changed to require analysts to model a

group of sub-states for each action state. As a result, CASE tools will enforce analysts to follow this rule, which can use the following approach: to check whether every action state in activity diagrams is a composite state in a state chart diagrams. If there is no such mapping for any action state, the program shows a violation warning. This is expressed in the following:

```
Get actionState;
Get asName;
Get states;
Get sNames;
If sNames.contains(asName)
    If state==composite
        Exit;
Show error;
```

Corollary 17 States must not be associated with any actions. Sub-states corresponding to different models should be used instead.

The last rule already requires modeling a group of sub-states for each action state. That means the actions of action states can be represented by the whole series of transitions among those sub-states. Thus, "there appears no need to assign such an action" (Evermann, 2003, p84). That is, all actions associated with states can be modeled by sub-states instead. States should not associate with any actions. This is Corollary 17.

This corollary proposes a change in UML specification. If adopted, CASE tools will disable the feature of associating actions with states.

Corollary 18 All states in an activity diagram must be states of the same object.

In activity diagrams, there are states and state transitions. In addition, "ontologically, states are defined for a given thing and state transitions are defined only between states of the same thing" (Evermann, 2003, p86). Thus, all states in activity diagrams must represent the same object.

In UML, "an activity graph is a special case of a state machine" (OMG, 2003, p2-172). In addition, UML specifies that a state machine is a behavior that specifies the sequences of states of an object (OMG, 2003, g14). That is, an activity diagram only represents one object. Consequently, all states in an activity diagram must be states of the same object. That means that this corollary cannot to be violated. Thus, there is no need to implement it.

Corollary 19 If the partitions of an activity diagram represent different objects, they must be part of a composite, which is shown in the class diagram.

According to Evermann and Wand, "partitions (swimlanes) are employed in UML activity diagrams to group states or action states together" (Evermann, 2003, p86). States in different partitions of activity diagrams can describe different objects. In addition, "an activity graph is a special case of a state machine that is used to model processes involving one or more classifiers" (OMG, 2003, g2). However, according to Corollary 18, all states in an activity diagram must be states of the same object. This seems to be a contradiction. However, the same object in Corollary 18 is the single composite object of those different

objects in this corollary (Corollary 20). That is, all different objects represented by states in an activity diagram must be part of a composite object. Thus, this composite object should be modeled in a class diagram. For example, in an activity diagram a swimlane divides states into two partitions of 'make motor' and 'make body.' These two groups of states represent two objects of 'Motor Workshop' and 'Body Workshop' respectively. These two objects are actually parts of the composite object 'Car Factory.' The composite relationship: {Car Factory: Motor Workshop, Body Workshop, ...} should be modeled in a class diagram.

In UML, activity diagrams use swimlanes for partitions: "A swimlane maps into a Partition of the states in the ActivityGraph" (OMG, 2003, p3-162). In activity diagrams, the program can check state partitions divided by swimlanes to see what objects the partitions describe. If these partitions describe more than one object, the program checks in class diagrams whether these objects participate an aggregate association as parts. If no such composition is found, the program shows a violation warning. This is expressed in the following:

```
Get swimlanes;
Get partitions[i];
Get objects;
If objects[j]!==objects[k]
    Get aggregates;
    Get partClasses;
    If partClasses.contains(objects[j])&&partClasses.contains(objects[k])
        Exit;
Show error;
```

Rule 25 The quantitative object behaviour (for each model) is entirely describable by top-level state chart (SC0)

In UML, object behaviours are modeled by operations: "An operation is a service that can be requested from an object to effect behaviour" (OMG, 2003, p2-44). In addition, "operations are related to ontological notion of change and ontologically, all (quantitative) changes of a thing are describable by a series of state transitions among stable states" (Evermann, 2003, p87). That is, behaviours correspond with state transitions through operations and changes. For an object, the class diagram models its behaviours while the state chart diagram models its states and state transitions. In state chart diagrams, the top-level state chart is the composite super-state describing the whole object. Thus, all behaviours of an object can be described by state transitions in top-level state charts.

Since object behaviours are modeled by operations in UML, our program checks class operations in class diagrams for this rule. Then it searches top-level state charts to see whether there are transitions corresponding with every operation. If the program cannot find any transition for an operation, it shows a violation warning. This is expressed in the following:

```
Get transitions;
If transition==top;
    Add to vector;
Get class;
Get operation;
```

If ! vector.contains(operation)
Show error;
Exit;

Rule 26 All UML-transitions in SC0 must correspond to an operation of the object which SC0 is associated with.

By the last rule, we know that every operation of an object can be described by transitions in the top-level state chart (SC0) of that object. That is, these SC0 transitions correspond to the operation they describe. Note that several transitions correspond to one operation, because there may be several state charts modeling one object for different purposes. For example, two SC0s of a 'Person' object model for purposes of capturing his/her physical and mental activities. In the physical model, states change from 'sleep' to 'shower' by the transition 'get up.' In the mental model, states change from 'dream' to 'think' by the transition 'get up' as well. The transitions 'get up' in these two models correspond to the operation 'get up' of the 'Person' object.

To implement this rule, we check each transition in SC0 to see whether it has corresponding operations in class diagrams. This can be done by first getting all operations of a class and adding them to a vector. Then the program gets each transition in SC0 of this class. After this, it checks whether the transition is included in the vector. If the transition is excluded, it shows a violation warning. This is expressed in the following:

Get class;
Get operations;

```
Add operations to opVector;
Get stateMachine;
Get transition;
Get sourceState;
Get targetState;
If sourceState.container==top&&targetState.container==top
    If opVector.contains(transition)
        Exit;
Show error;
```

Corollary 20 Every object must have at least one operation.

It is stated that "In our ontology everything must be able to change" (Evermann, 2003, p89). Also, we map BWW-thing with UML-object (Rule 1) and map BWW-change with UML-operation (Rule 25). Thus, every object must possess at least one operation to ensure it is able to change.

To implement this corollary, the program only needs to check each class to see whether it possesses operation. If the program finds any class has no operation, it shows a violation warning. This is expressed in the following:

```
Get class;
Get operation;
If operation==null
    Show error;
Exit;
```

Corollary 21 States in SC0 are stable.

"In UML, an object remains in a particular state until an operation invoked by some other object" (Evermann, 2003, p90). That is, if there is no other object participating, an

object will remain in a stable state. Also, states in SC0 only describe a whole object. That means that in a SC0 of an object, all states are stable.

The implementation approach uses trigger of transitions to identify the stability of states. In state chart diagrams, transitions can have triggers by defining an event. That is, if a state transition has a trigger event, only the event can cause the transition. If a state transition has no trigger event, it can spontaneously happen. This is consistent with BWW-ontology, which identifies stable and unstable states by judging whether state transitions are externally induced or spontaneous (Evermann, 2003, p90). Thus, UML specification should force analysts to model a trigger for transitions in SC0. Also, CASE tools should enable this feature.

Corollary 22 All UML-transitions in SC0 must be associated with a UMLevent.

Since states in SC0 are stable (last corollary), they must not have any spontaneous out transitions. In other words, all transitions in SC0 must have a trigger event. Actually, we already use this corollary in the implementation approach of the last corollary.

This is a corollary and already covered by Corollary 21, there is no need to implement it.

Rule 27 An object must exhibit additional operations expressing qualitative changes, if a super- or sub-class is defined and instances can undergo changes of class to the super- or

sub-class.

We have discussed object generalization / specialization; in this rule, we will talk about operations applying to super / sub classes. According to Rule 21, a state corresponds to a certain property value set. In addition, generalization / specialization through losing / acquiring additional properties is a qualitative change (Bunge, 1977, p220); different property sets identify different things (Bunge, 1977, p88). When an object undergoes qualitative change it acquires / loses properties and is specialized / generalized into the sub / super class. Since "the acquisition or loss of behaviour is generally concurrent with loss or acquisition of properties that change in that behaviour" (Evermann, 2003, p33), an object is specialized / generalized into the sub / super class through acquisition / loss of additional properties and behaviours concurrently. Thus, we get this rule.

The implementation approach of this rule is to compare specialized sub-classes with general super-classes to see whether the sub-classes possess more operations. First, the program gets generalizations as well as both parent and child ends. Second, it gets all parentEnd operations and puts them into a vector. Then, each operation of the childEnd will be matched with the vector. If the program finds any operation not in the vector, the system exits. Otherwise, it shows a violation warning. This is expressed in the following:

```
Get associations;
If association==generalization
    Get parentEnd;
    Get operations;
    Add to vector;
```

```
Get childEnds;
Get operations;
While operations.hasNext()
    Match with vector;
    If find not in vector
        Exit;
Show error;
Else exit;
```

Notice that the inherited operations from the super-class should not be shown in the sub-classes; any operation in the sub-class is supposed to be additional than its super-class. Thus, we only need to check whether a sub-class possesses any operation. If we cannot find any operation for a sub-class, it shows a violation warning. This is expressed in the following:

```
Get associations;
If association==generalization
    Get childEnds;
    Get operations;
    While operations.hasNext()
        Exit;
    Show error;
```

Rule 28 Methods may be described by state charts other than top-level state charts.

We have mapped operations with state transitions. In UML, a method is "the implementation of an operation. It specifies the algorithm or procedure associated with an operation" (OMG, 2003, g9). In ontology, a lawful transformation can be thought of as transition laws (Evermann, 2003, p34). Thus, methods are mapped with lawful transformations (Evermann, 2003, p94). A lawful transformation is defined as a "path" in

state space between an initial and a final state. That means that we can use a state chart to model lawful transformations, namely methods. However, the top-level state chart models behaviour from a class-scope. This top-level SC0 cannot model methods. In other words, state transitions in the SC0 represent operations of a class, while other state charts represent methods implementing these operations.

Since UML allows state charts to describe methods (OMG, 2003, p3-136), this rule is consistent with UML. We need to proscribe methods described by the SC0. This can be done by checking state charts describing methods. If a state chart is the top-level SC0, the program shows a violation warning. This is expressed in the following:

```
Get class;
Get SC0;
Get method;
Get stateChart;
If stateChart==SC0
    Show error;
Exit;
```

Corollary 23 A state chart describing a method must begin and end with those states in SC0 which the operation that the method implements is a realization of.

The last rule specifies that methods can be described by state charts. In addition, methods implement operations. State charts describing methods actually describe implementations of operations. Rule 25 specifies that every operation can be represented as a state transition in a top-level state chart (SC0). That is, state charts describing methods

associate with transitions in SC0. Thus, these state charts must begin with source states and end with target states of those transitions in SC0.

The implementation approach for this corollary is to check whether the initial and final states of state charts describing methods equal the source and target states of transitions describing operations, respectively. First, the program gets the SC0 of a class. Then it gets transitions in this SC0 corresponding to operations of the class. For each transition, its source and final states will be retrieved. Then the program checks state chart describing the method implementing the operation, which corresponds to this transition. If the initial or final state of the method and the source or target state of the transition are not the same respectively, it shows a violation warning. Please note that not every operation has to be implemented by a method; for example, an abstract operation has no method (OMG, 2003, p3-46). That is, the program also needs to make clear whether an operation has a method. This is expressed in the following:

```
Get class;
Get sc0;
Get sc0Transitions;
While sc0Transitions.hasNext()
    Get transitionName;
    Get sourceState;
    Get targetState;
    Get methods;
    While methods.hasNext()
        Get methodName;
        If methodName==transitionName
            Get methodStateChart;
            Get initialState;
```

Get finalState;
If !(initialState==sourceState)||!(finalState==targetName)
Show error;
Exit;

Corollary 24 State transitions out of the initial state of a method realizing an operation must be associated with the same event that is associated with the transition in SC0 which represents that operation.

Since a state chart describing a method begins with the source state of transition describing an operation in SC0, the initial state of the method is the same as the source state of the operation. The source state transits to the target state via the transition describing the operation and this transition is triggered by a trigger event. Since the state chart describes the method of the operation, the initial state of the method should be triggered by the same event, which triggers the transition describing the operation. Note that 'associated' in this corollary is not UML-associations, but means **'relate with'**.

The implementation approach for this corollary is to check whether the trigger event of transition in SC0 representing an operation equals the trigger event of transition out of the initial state of method implementing the operation.

First, the program gets the SC0 of a class. Then, the program gets transitions in this SC0 corresponding to operations of the class. For each transition, the program gets its trigger event. Then it checks the state chart describing the method implementing the operation, which corresponds to this transition. It gets trigger events of transitions out of

the initial state of the method. If these two trigger events are not same, the program shows a violation warning. This is expressed in the following:

```
Get class;
Get sc0;
Get sc0Transitions;
While sc0Transitions.hasNext()
    Get transitionName;
    Get trigger;
    Get event1;
    Get eventName1;
    Get methods;
        Get methodName;
        If methodName==transitionName
            Get methodStateChart;
            Get initialState;
            Get outTransition;
            Get trigger;
            Get event2;
            Get eventName2;
            If !(eventname1==eventName2)
                Show error;
Exit;
```

Corollary 25 A state chart either expresses the external behaviour of an object (SC0), a method, a signal reception or is a composite state contained in another state machine.

According to the UML specification, "A statechart diagram can be used to describe the behavior of instances of a model element such as an object or an interaction" (OMG, 2003, p3-315). If the state chart is a top-level SC0, it expresses the external behaviour of an object (Rule 26). Other state charts may express methods (Rule 28). According to UML, receptions designate signals and are the summary of expected behaviours (OMG, 2003,

p2-102). Thus, state charts can also express signal receptions. (We will discuss signal receptions in Rule 30.) However, there is a kind of state charts not expressing these behavioural model elements. These are composite states contained in other state machines, which express behaviours of instances of model elements.

The implementation approach is to check whether all state charts express one of the four situations in this corollary. The program gets a state chart to see whether it is a composite state other than SC0. If it is not a composite state, the program checks the model element that owns the state chart. If the owner model element is not an object, a method or a signal reception, it shows a violation warning. This is expressed in the following:

```
Get stateChart;
If !(stateChart.isComposite)
    While state.hasContainer
        Get container;
    Get container.owner();
    If !(owner==object)||!(owner==method) ||!(owner==signalReception)
        Show error;
Exit;
```

Rule 29 An operation is not directly specified by state machines. Instead, the methods that implement an operation are specified by state machines.

In Corollary 25, we know that state machines represent behavioural features of model elements. These model elements include objects, methods and signal receptions. Rule 25 specifies that operations are described as state transitions in top-level state charts (SC0s). That is, operations are not behavioural features of model elements and cannot be specified

by state machines. Thus, it is not operations specified by state machines directly, but methods specified by state machines. In addition, these methods implement those operations.

The implementation approach of Corollary 25 will shows a violation warning if a state machine specifies a model element other than an object, a method or a signal reception. That is, if an operation is specified by state machine, the program for Corollary 25 will detect the violation. There is no need to implement this corollary.

Corollary 26 A state machine that specifies the behaviour of a class or a method is not contained in other state machines.

State machines are used to represent behavioural features of model elements. Classes and methods are these model elements and can be specified by state machines. In Corollary 25, we have discussed that some state machines can be contained in other state machines. Can a state machine that specifies the behavior of a class or a method be contained in other state machines? The answer is "No". State machines contained in other state machines actually do not model behavioural features of model elements. They only describe some states within the global behavioural features modeling of model elements. Since classes and methods are model elements, they cannot be described by some states within other state machines.

The implementation approach is to check whether a state machine attaching to a class

or a method is top-level. The program first gets a class or method. Then it gets state machines attaching to this class or method. Finally, it checks the containers of these state machines. If any container is not top-level, the program shows a violation warning. This is expressed in the following:

```
Get class/method;
Get stateMachine;
Get smContainer;
If !(smContainer==top)
    Show error;
Else exit;
```

Corollary 27 The method corresponding to a state chart must modify the attribute values of the object corresponding to the values defined for the initial and final state of the method.

Since operations are described as transitions by top-level state charts and methods implement operations, methods actually describe transitions in SC0. Corollary 14 specifies that every transition must change the value of at least one attribute used to define the state space. That is, methods must change some attribute values. What are these attributes used to define the state space in this case? Corollary 23 tells us that the initial and final states of state charts describing methods equal the source and target states of transitions describing operations, respectively. According to Rule 21, there must be a set of object attributes, whose values correspond to these initial / source and final / target states. This set of object attributes is the answer to our question above. Thus, we get this corollary.

Since this corollary is a consequence of other rules and corollaries, if modellers strictly follow other rules and corollaries, this corollary will be satisfied. Thus, there is no need to implement it.

Rule30, Rule31, Corollary28:

State machines describe behavioural features of model elements. We have discussed behavioural features of operations and methods. Now, we discuss another behavioural feature, receptions: "A reception is a declaration stating that a classifier is prepared to react to the receipt of a signal" (OMG, 2003, p2-102). That is, receptions capture the process of receiving signals of classifiers. Actually, this process can be realized by operations. Operations are modeled by transitions in state charts. For each transition, if there is a signal, the trigger of the signal event can be modeled. That means that classifiers receiving signals can be represented via operations by the trigger of signal events of transitions in state charts. Thus, receptions in UML are redundant. We propose a rule in the following.

Rule-New 03[1] Receptions should not be modeled.

The implementation of the new rule is to check whether there is any reception declared. Since all receptions attach to signals, the program gets signals first. Then it checks whether a signal has any reception. If the program finds one, it shows a violation warning. This is expressed in the following:

[1] For consistency between the thesis and the program code, we use 'Rule-New 03' for this new rule and 'Rule-New 02' for the next new rule.

Get signal;
Get receptions;
If !receptions.isEmpty
    Show error;
Exit;

However, even though we propose receptions in UML to be redundant, it does not conflict with UML or ontology. Analysts may still want to use it. If this is the case, there are some rules which must be followed to ensure consistency. We still discuss and implement these rules (Rule 30, Rule 31 and Corollary 28) in the following. Please note that even though we implement both Rule-New 03 and {Rule 30, Rule 31, Corollary 28}, they can not be satisfied concurrently.

Rule 30 An operation must be associated with the declaration of signal reception.

States and transitions in BWW-ontology describe all behavioural features in the world (Evermann, 2003, p99). In UML, state charts, operations and methods are behavioural features and have been mapped with BWW-states and transitions. There is another behavioural feature in UML, reception: "Reception is a child of BehaviouralFeature" (OMG, 2003, p2-102). That is, UML-receptions should also map with BWW-states and transitions. In addition, these UML concepts map with the same BWW concept, and they should be consistent with each other. Previous rules and corollaries have specified operations and methods to be associated. This rule specifies that signal reception should also be associated with them.

Because receptions are a sub-class of behavioural features, which associate with signals, we use the implementation approach to check whether an operation associates with signals and whether the signals possess receptions. The program gets classes with their operations. For each operation, it gets signals. If the signal is not null, the program gets its receptions. If no reception is detected, it shows a violation warning. This is expressed in the following:

```
Get class;
Get operation;
Get signals;
If !(signals==null)
    Get receptions;
    if !(receptions==null)
        Exit;
Show error;
```

Rule 31 The event associated with an operation must be identical to the event associated with the signal associated with the reception.

In the last rule, we have discussed that operations, methods and signal receptions in UML map with the same ontology behaviour, namely state and transitions. That means, they should be consistent with each other. Thus, the event triggers of an operation should equal the event trigger transition out of initial state of the reception associated with the signal.

The implementation approach is to compare trigger events associated with operations and transitions out of the initial states of receptions to see whether they are identical. The

program first obtains classes as well as its operations. Second, it gets SC0 describing the operations. For each operation / transition, it gets the trigger event. Then the program gets signal receptions associating with this operation. In state charts describing these signal receptions, trigger events of transitions out of the initial states will be compared with the event above. If these two events are not identical, the program shows a violation warning. This is expressed in the following:

```
Get class;
Get operation;
Get SC0;
Get transition1;
Get event1;
Get signal;
Get reception;
Get stateChart;
Get initialState;
Get transition2;
Get event2;
If !(event1==event2)
    Show error;
Exit;
```

Corollary 28 The state machines associated with a reception and with a method specifying the implementation of an operation which is in turn associated with that reception, must possess the same initial and final states.

We have discussed in Rule 30 and Rule 31 that, operations, methods and receptions should be consistent with each other. That is, these behavioural features should begin and end in consistent states. Previous rules and corollaries have specified this consistency

between operations and methods. This corollary specifies this consistency between methods and receptions.

The implementation approach is to compare initial and final states of state machines of receptions and methods to see whether they are identical respectively. The program first gets an operation. Second it gets reception and method associating with the operation. Then, it gets two state machines associating with them respectively. Finally, the program checks whether the initial states in these two state machines are identical, as well as final states. If they are not the same, it shows a violation warning. This is expressed in the following:

```
Get class;
Get operation;
Get reception;
Get stateMachine1;
Get initialState1;
Get finalState1;
Get method;
Get stateMachine2;
Get initianState2;
Get finalState2;
If (!initialState1==initialState2)||(!finalState1==finalState2)
    Show error;
Exit;
```

Rule 32 Acquisition (loss) of independent properties leads to expansion (contraction) of the thing's top-level state space SC0 by an orthogonal region.

According to previous rules and corollaries (Rule 14, 15, Corollary 11, 12,13),

specialized sub-classes possess properties inherited from their super-classes and acquire additional properties. How does an analyst model this in state chart diagrams? First, we need to discuss what is acquisition of independent properties. In the acquired additional properties in specialized classes, some are independent from other properties of their super-classes, while some are dependent. Dependent additional properties only exist when some super-class properties have some certain values. For example, in a 'Person' specialized by 'SinglePerson' and 'MarriedPerson' association, 'MarriedPerson' acquires the additional property of 'nameOfSpouse' while 'SinglePerson' does not possess this additional property. The property 'nameOfSpouse' only exists when the value of the 'Person's' property 'marrigeStatus' is 'married.' In this case, the property 'nameOfSpouse' is a dependent property. Independent additional properties exist all the time and are irrelevant with properties of their super-classes. For example, in a 'Person' specialized by 'Employee' association, 'Employee' acquires the additional property of 'skill.' In this case, an 'Employee' has the property 'skill' regardless of whether, for example, the value of that 'Employee's sex' is 'male' or 'female.' Now we discuss how to model state charts for specialized classes by acquisition of independent properties and acquisition of dependent properties, respectively. Since independent properties of sub-classes and properties of super-classes are irrelevant, their values may change concurrently. For example, 'sex' changes from 'male' to 'female' while 'skill' changes from 'driving' to 'programming.' Also, according to Rule 21, values of these independent

properties define states of objects they describe, in this example, the 'employee.' Since these irrelevant property values change concurrently, the structure of an orthogonal region is adopted. That is, the states and transitions in SC0 should be inherited and the independent properties should be modeled as concurrent states and transitions with those in SC0.

The implementation approach is as follows: First, we search for specialized sub-classes and get their attributes. Second, we need to identify whether these attributes are independent or not with attributes of their super-classes. However, the program cannot know whether they are independent or dependent. It will ask users for the answer. When users indicate 'independent', the program continues; otherwise it exits. Next, it checks state charts for the specialized class. According to the rule, these state charts must have an orthogonal region. Thus, for those state charts, which are not concurrent, it shows a violation warning. This is expressed in the following:

```
Get genaralization;
Get superClass;
Get subClass;
Get attributes;
Query user;
If attributes==independent
        Get stateMachine;
        Get top;
        If !(top==concurrent)
            Show error;
Exit;
```

Now, we talk about modeling of state charts for specialized classes by acquisition of dependent properties. Additional properties of specialized classes only exist when properties of super-classes have certain values. According to Rule 21, these certain values define a certain state. That is, change of additional property values of specialized classes only happens within the certain state. Thus, this certain state should be modeled as the composite super-state of states and transitions corresponding to the additional properties. However, Evermann's research does not formalize this as a rule. Here we propose a rule to fill in this blank:

Rule-New 02 Acquisition (loss) of dependent properties should be modeled as a sub-machine of the related state of things modeled by top-level state SC0.

Figure 5.1 and Figure 5.2 describes this rule.

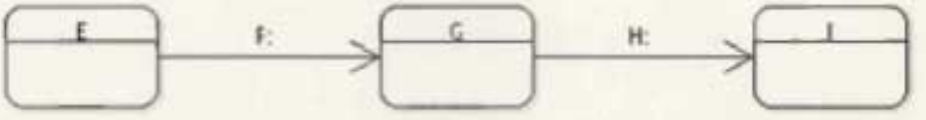

Figure 5.1 State chart before class is specialized

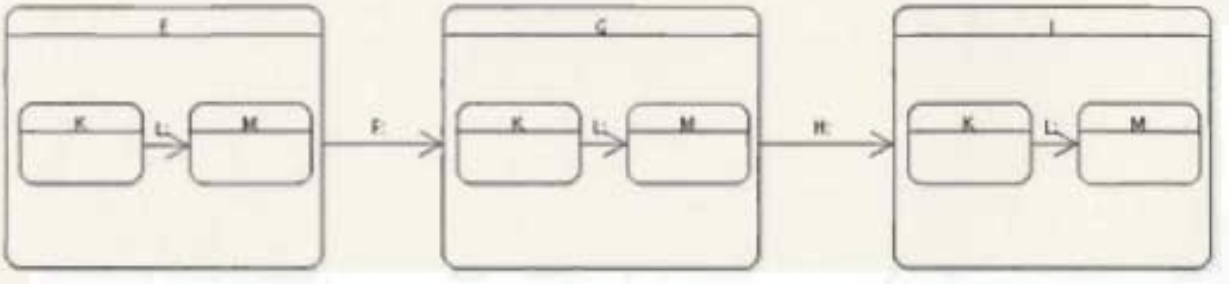

Figure 5.2 State chart after class is specialized

The first part of the implementation approach is the same as Rule 32. First, we search for specialized sub-classes and get their attributes. Second, we need to identify if these attributes are dependent or not with attributes of their super-classes. Since the program cannot know whether they are independent or dependent, it asks users for the answer. When the user indicates dependent, the program continues; otherwise it exits. Then, it checks state charts for the specialized class. According to the rule, to model additional features of the sub-class, all states in SC0 of the sub-class must be composite states. In this way, the additional features can be modeled within each of these composite states. If the above are not satisfied, the program shows a violation warning. This is expressed in the following:

```
Get generalization;
Get superClass;
Get subClass;
Get attributes;
Query user;
If attributes==nonIndependent
    Get stateMachine;
    Get transitions;
    While transitions.hasNext()
        Get sourceState;
        Get targetState;
        Get stateContainer;
        If (sourceStateContainer==top)&&(targetStateContainer==top)
                If !(sourceState.isComposite)||!(targetState.isComposite)
                    Show error;
Exit;
```

Corollary 29 Every object must be capable of at least one state transition or be able to

undergo change of class to a super- or sub-class.

The Bunge's ontology specifies that all things are changeable (Bunge, 1977, p219). Mapping this to UML, we get the conclusion that every object must be capable of change. There are two types of change, namely quantitative change and qualitative change (Evermann, 2003, p33). Quantitative change is the change of values of properties. Qualitative change is the change of acquiring or losing properties. That is, attributes of objects must be capable of either changing their values or being acquired or lost. In UML, operations change the values of attributes and all operations can be described by state transitions in top-level state charts (Rule 25). In addition, every object must possess at least one operation (Corollary 20). Thus, objects that have undergone quantitative change must be capable of at least one state transition. When objects change by acquiring or losing properties, they should be specialized as sub-class or generalized as super-class, respectively (Rule 12, 14, 15, 27, Corollary 11, 12,13)

This corollary comes from previous rules and corollaries. It specifies the modeling of two types of change. Modeling of quantitative change was already discussed and implemented by Rule 25 and Corollary 20. Modeling of qualitative change was already discussed and implemented by Rule 12, 14, 15, 27 and Corollary 11, 12, 13. Thus, there is no need to implement it.

Chapter 6 Interaction Rules

The last chapter discussed change within things. In this chapter, we will discuss the implementation approach of rules related to interactions between things. Interactions are expressed by message-passing: "An interaction contains a set of partially ordered messages, each specifying one communication" (OMG, 2003, p2-128). Messages may be considered as ontological things. For example, when a 'Manager' interacts with an 'Employee' by sending an email including tasks. This email is the message passing from 'Manager' to 'Employee.' However, this is only a special case and does not apply to all interactions. For example, when a 'Driver' interacts with a 'Car' by turning the steering wheel, the 'Driver' does not send any message. In addition, "a message is a specification of stimulus" (OMG, 2003, p3-111). Thus, "Stimuli are not things in the world. They are abstract concepts that serve as descriptions, illustrations, abstractions or representations of interaction" (Evermann, 2003, p118). That is, both stimuli and messages have no equivalent in ontology and are ontologically excessive (Evermann, 2003, p120). We propose the following rule:

Rule-New04 Neither stimuli, nor messages should be modeled.

The implementation is to check whether there is any stimuli or message modeled. If the program finds one, it shows a warning. This is expressed in the following:

```
Get stimuli;
Get message;
If !(stimuli==null)||!(message==null)
    Show error;
Exit;
```

However, even through we propose that messages are ontologically unnecessary in UML, they do not conflict with ontology or other concepts in UML. Analysts may still want to use them. If so, there are some rules that must be followed to insure consistency. We still discuss and implement these rules and corollaries in the following. Please note that, Rule-New 04 and the following rules / corollaries cannot be satisfied concurrently.

Rule 33 For every class of objects between which message passing is declared, there exists an association class or the two classes are parts of the same aggregate.

According to the above discussion, interactions are expressed by message-passing: "A message defines a particular communication between instances that is specified in an interaction" (OMG, 2003, p2-119). That is, when message-passing exists, there is an interaction. Since interactions happen between objects, we need to know how interactions affect participating objects. According to Bunge, "Two different things X and Y interact iff each acts upon the other" (Bunge, 1977, p259). That means both objects participating an

interaction undergo changes. Since "change may be quantitative, in which case the values of one or more properties is changed" (Evermann, 2003, p33), property values of both objects should change. Also because every property must lawfully relate to other properties (Bunge, 1977, p77), the changes of both objects must obey laws. That is, an interaction must lawfully change properties of both participating objects. However, a law is any restriction on the possible values of a thing (Bunge, 1977, p129). A law only constrains property values of a single object (Evermann, 2003, p32). How can an interaction change property values while satisfying laws in both objects? The answer is, the interaction must change the mutual properties of participating objects. We have mapped mutual properties to association class before (Rule 3). Thus, for every class of objects between which message-passing is declared, there exists an association class. For example, a 'Company' interacts with an 'Employee' by promotion. This promotion interaction changes the value of mutual property 'salary' of both 'Company' and 'Employee.'

Now, consider the latter part of this rule. One interpretation is that if two interacting objects are parts of an aggregation, there does not have to exist an association class. However, we disagree with Evermann's conclusion, which motivates from "If the two things are parts of the same composite, changes among them may happen by virtue of emergent properties" (Evermann, 2003, p113). The motivation conflicts with ontology that "emergent properties are not possessed by any of the parts of a composite" (Evermann,

2003, p32), while mutual properties are properties possessed by all participating objects. That is, while change of mutual properties affects all participating objects, change of emergent properties do not have to affect all objects as parts. For example, a 'CPU' and a 'HardDisk' are two parts of composition 'Computer.' When the property 'type' of 'CPU' changes from 'Celeron-1.2G' to 'Pentium4-2.4G', the emergent property 'computationSpeed' changes from 'medium' to 'fast.' However, no property of the 'HardDisk' changes. Thus, we propose to delete the latter conclusion of this rule.

The implementation approach only needs to check whether there is an association between interacting objects. This is because we already enforce an association class for every association in Rule 20. The program first gets interaction and its message. Then the sender and receiver objects will be retrieved. Next, it will check objects corresponding to the sender and receiver in class diagrams. If there is an association between them, the program exits. Otherwise, it shows a violation warning. This is expressed in the following:

```
Get interaction;
Get message;
Get sender;
Get receiver;
Get class;
If class==sender
    sendClass=class;
If class==receiver
    receClass=class;
Get sendClass;
Get association;
While association.hasNext()
    Get assoOtherEnd;
```

```
Get class;
If class==receClass
    Exit;
Show error;
```

Rule 34 Every object must be the receiver and sender of some message.

In ontology, "every thing acts on, and is acted upon by, other things" (Bunge, 1977, p259). Previously, BWW-things were mapped to UML-objects (Rule 1) and interactions were expressed by message-passing (Rule 33), thus, every object must send and receive some message. For example, a human resource manager interacts with a job applicant. The 'Applicant' object sends the message of 'applyJob' and receives the message of 'accept / decline.' The 'Manager' object sends the message of 'accept / decline' and receives the message of 'applyJob.' However, according to Evermann's research, this rule does not apply to every object participating in an aggregation as a part (Evermann, 2003, p114). That is, not every part object has to send and receive messages. As long as some part objects send messages and some part objects receive messages, the whole object sends and receives message. There is one thing needs to be noted here, which is not explained by Evermann. Sending and receiving messages discussed above for part objects should not include messages between these part objects. Even though one part object sends / receives messages to / from other part objects, from the scope of the object as a whole, it does not send / receive messages. For example, a whole object 'Person' consists of part objects 'Brain' and 'Eye.' When 'Brain' sends message 'open' to 'Eye', this message is sent

inside the 'Person.' From the scope of the 'Person', there is no message sent or received. That is, if class 'A' is an aggregate of classes 'B' and 'C', in addition messages only be sent and received between 'B' and 'C', this still violates this Rule. This is because from the scope of 'A', it does not send or receive messages.

The implementation approach is to check whether every object is the sender or receiver of any message. First, the program gets messages in collaboration diagrams. For each message, it gets the sender and receiver. Then, it gets base classes of sender and receiver and adds them into "_sender" and "_receiver" vectors, respectively. After this, all classes and their associationEnds will be retrieved. If there is an aggregation, it gets the association and the part class. These part classes will be added into the "_part" vector. Finally, it gets an object (class). (Please note, this step is inside a loop. An object is retrieved in each loop. Eventually, all objects will be retrieved.) For each class not in the "_part" vector, if it can be found in both "_sender" and "_receiver" vectors, the program will exit. Otherwise, it shows a violation warning. Since this rule does not apply to every object participating in an aggregation as a part, we do not check part objects. This is expressed in the following:

Get interaction;
Get message;
Get sender;
Get base;
Add to _sender;
Get receiver;
Get base;

```
Add to _receiver;
Get class;
Get assoEnd;
If (asoEnd==aggregate)
    Get otherAssoEnd;
    Get type;
    Add to _part;
If (!_sender.contains(class)||(!_receiver.contains(class)))&&!_part.contains(class)
    Show error;
```

Corollary 30 An association class cannot be sender or receiver of a message.

The motivation of Rule 34 is that every thing acts on and is acted upon by other things in ontology. From Rule 1, we know that BWW-things map with UML-objects. In addition, from Corollary 2 we know that association classes cannot represent objects. That is, association classes are not ordinary classes, which must be the receiver and sender. Can association classes send or receive messages? From Corollary 5 and Rule 4, we know that mutual properties cannot act on or be acted upon. Thus, association classes cannot be sender or receiver.

Since this corollary can be fully deduced from previous rules and corollaries, strictly following them should satisfy this corollary.

However, this is not true in our CASE tool. If the user models an association class as a message sender or receiver, no violation warning will be shown. This is because, this corollary applies to collaboration diagrams while programs implementing those rules and corollaries only check class diagrams. Because association classes are not classes of

objects, but classes of mutual properties, association classes in class diagrams should not correspond to objects in other diagrams, such as collaboration diagrams and sequence diagrams. We propose a rule to fill this gap:

Rule-New05 Association classes in class diagrams should not be instantiated by link-objects modeled in other UML diagrams.

According to the previous rules and corollaries, association classes are mutual properties and objects are substantial things. Properties will not equal things. That is, the implementation of this rule is covered by previous rules and corollaries. Thus, there is no need to implement it.

Rule 35 A constraint relates attributes of a single class or attributes of association classes the class participates in.

UML has a concept of constraint, which is "a semantic relationship among model elements that specifies conditions and propositions that must be maintained as true; otherwise, the system described by the model is invalid" (OMG, 2003, p3-26). As we can see, constraints are restrictions, which guarantee a model's validity. This conforms to the concept of law in ontology. In ontology, a law is any restriction on the possible values of a thing (Bunge, 1977, p129). Thus, UML-constraints are mapped with BWW-law. Ontology also specifies that a law applies to a thing. That is, BWW-law cannot relate the properties

of different things (Evermann, 2003, p32). So, UML-constraints should also apply to a single class. Note that association classes are groups of mutual properties of participating classes. Constraints relating attributes of association classes do not conflict the above discussion. For example, class 'A' interacts with class 'B' via association class 'C.' A constraint cannot relate attributes of 'A' to attributes of 'B', but only relates attributes of 'A' or that of 'C.'

The implementation approach is to check constraints attached to class attributes to see whether they extend to different classes. The program first gets constraints of an attribute in a class. Then, it reads these constraints. If any constraint relates to attributes of other classes, then it checks whether these other classes are association classes, which this class participates in. If any is not an association class, the program shows a violation warning. This is expressed in the following:

```
Get classA;
Get attribute;
Get constraint;
Get associatedAttributes;
If !(associatedAttribute==null)
    Get assoAttOwner;
    If !(assoAttrOwner==classA);
        Show error;
Exit;
```

According to UML, constraints may use uncertain languages: "One predefined language for writing constraints is OCL; otherwise, the constraint may be written in natural language" (OMG, 2003, p3-27). That is, the computer may not be able to read

constraints. One possible, but not accurate, solution is to search the whole constraint text string. If any string equals an attribute string in the whole class diagram, we consider it specifying that attribute. It is obvious that this solution is not accurate.

Rule 36 For every attribute there exists a constraint which relates this attribute to some other attribute.

In ontology, "every substantial property is lawfully related to other substantial property" (Bunge, 1977, p78). That is, every UML-attribute must relate to other attributes. In the last rule, we map Ontology-law with UML-constraint. That means that there exists a constraint of every attribute to relate it to other attributes. Note that other attributes here refer to attributes of the same class (See Rule 35 for details).

The implementation approach is to check every attribute to see whether there exists its constraint specifying the relation among it and other attributes. The program first gets constraints of an attribute in a class. Then, it reads these constraints. If no constraint relates to other attributes, it shows a violation warning. When there are constraints relating to other attributes, the program exits. It does not continue to check whether these attributes belong to the same class, because this has been done in Rule 35. This is expressed in the following:

Get class;
Get attribute;
Get constraint;

Get associatedAttributes;
If associatedAttribute==null
    Show error;
Exit;

Also, this implementation has the limitation as we discussed in the last rule.

Corollary 31 A UML-state transition associated with an action must modify an association class attribute's value.

In this corollary, we discuss actions: "The action that causes a stimulus to be sent according to the message" (OMG, 2003, p2-119). That is, an action causes message-sending. According to the discussion of Rule 33, message-passing expressing interaction changes attribute values of an association class. Since the object whose state transition associated with the action sends the message, the transition must modify an association class attribute's value. This association class is the one which represents the mutual properties of this object and the object receiving the message. Evermann also proposes that this corollary only applies to transitions in method descriptions. The motivation is that in SC0s, "operations may leave the object in the same state or in a state in which at least the mutual or emergent properties possess the same values" (Evermann, 2003, p120). When an object is still in the same state after a transition, it does not change. If this is the case, then according to "a UML-action is similarly interpreted as change in one object that brings about a change in another object" (Evermann, 2003, p120), there is no action accompanying this transition at all. Thus, this corollary should not only restrict

to state machine of methods, but also applies to the top-level state chart (SC0).

UML does not require modeling different attribute values for different object states (before and after transitions) (OMG, 2003, p3-42). Thus, our program cannot tell whether attributes values change. The ideal solution is to add this corollary in UML, including specifying such a mechanism. CASE tools will enable this feature according to the UML specification.

Corollary 32 For every interaction between UML-objects, there must exist a corresponding UML-state transition in both interacting UML-objects.

From the discussion of Rule 33 and Corollary 31, we know that an interaction between two objects change values of mutual properties of these objects. That is, property values of both objects are modified. From Rule 21 and Corollary 14, different property value sets determine different states of a thing. That is, both object states are changed via change of mutual property values by interaction. Thus, there exist corresponding state transitions in both interacting objects.

The ideal implementation approach is to adopt this corollary in UML. In this way, all CASE tools will enable the feature of associating interactions to state transitions of participating objects and force modellers to do that.

Corollary 33 A state transition associated with an event must modify an association class

attribute's value.

According to the UML specification, "Event instances are generated as a result of some action either within the system or in the environment surrounding the system" (OMG, 2003, p2-155). Evermann proposes that events signify reception of the stimulus (Evermann, 2003, p121). For example, "a call event represents the reception of a request to synchronously invoke a specific operation" (OMG, 2003, p2-142). Since "a stimulus is created and sent by an action, receipt of a stimulus is an event" (Evermann, 2003, p117), action and event are a pair of concepts and should have similar interpretations. According to the discussion of Rule 33, message-passing expressing interaction changes attribute values of an association class. However, it is the sender object which changes the mutual property values of itself and the receiver object, and thus makes the receiver object transit to another state. The receiver object does not actually actively modify association class attributes values. We change the wording of this corollary into the following:

A state transition associated with an event must be associated with the change of the attribute's value of an association class. This association class is the one which represents the mutual properties of this object and the object sending the message. Evermann proposes that this corollary also only applies to transitions in method descriptions. Because of the same thinking in Corollary 31, we propose that this corollary should not only be restricted to state machine of methods, but also applies to top-level state chart **(SC0).**

UML does not supply a mechanism of modeling different attribute values for before and after transitions (OMG, 2003, p3-42). Thus, the program cannot tell whether attribute values change. The ideal solution is to add this corollary in UML, including specifying such a mechanism. CASE tools will enable this feature according to the UML specification.

Corollary 34 A signal event may only be associated with a transition in a top-level state chart and the initial transition of a method implementing this.

"A signal is a specification of an asynchronous stimulus communicated between instances" (OMG, 2003, p2-102). As we have discussed at the beginning of this chapter, stimuli are ontologically excessive, the signal is a redundant concept and must conform to others. Evermann proposes an additional association in the UML meta-model that every signal must associate with an operation (Evermann, 2003, p123). Rule 25 specifies that every operation maps with a transition in the top-level state chart (SC0). That is, signals can only correspond into SC0. In addition, "a signal event represents the reception of a particular (asynchronous) signal" (OMG, 2003, p 2-146), and every signal event associates with a signal (OMG, 2003, p2-142). We get that every signal event must be represented in SC0. Since events trigger transitions in state charts, a signal event may only be associated with a transition in a top-level state chart. From Corollary 24, we can know that the signal event also associates with the initial transition of a method implementing the operation.

The implementation approach is to check whether there is any signal event modeled in state charts other than SC0. The program first gets a transition and its trigger event. If a trigger event is a signal event, then it gets the state chart which possesses this transition. If the owner of the state chart is a class, it checks whether the transition's container is top. If it is top, the program exits; otherwise it shows a violation warning. Since the latter part of this corollary comes from Corollary 24, which we have already implemented, there is no need to implement it. This is expressed in the following:

```
Get transition;
Get triggerEvent;
If !(triggerEvent==signalEvent)
    Exit;
Get stateMachine;
Get smOwner;
If smOwner==class
    Get sourceState;
    Get targetState;
    If (sourceState.isTop)&&(targetState.isTop)
        Exit;
Show error;
```

Corollary 35 A call event may only be associated with a transition in a top-level state chart or the initial transition of a method implementing this.

In UML, "A call event represents the reception of a request to synchronously invoke a specific operation. (Note that a call event instance is distinct from the call action that caused it.)" (OMG, 2003, p2-142). A call event receives the stimuli sent by a call action. According to the UML meta-model, every call event must associate with an operation

(OMG, 2003, p2-142). Rule 25 specifies that every operation maps to a transition in the top-level state chart (SC0). That is, every call event must be represented in SC0. Since events trigger transitions in state charts, a call event may only be associated with a transition in a top-level state chart. From Corollary 24, we can know that the call event also associates with the initial transition of a method implementing the operation.

The implementation approach is to check whether there is any call event modeled in state charts other than SC0. The program first gets a transition and its trigger event. If a trigger event is a call event, then it gets the state chart, which possesses this transition. If the owner of the state chart is a class, it checks whether the transitions container is top. If it is top, the program exits; otherwise it shows a violation warning. This is expressed in the following:

```
Get transition;
Get triggerEvent;
If !(triggerEvent==callEvent)
    Exit;
Get stateMachine;
Get smOwner;
If smOwner==class
    Get sourceState;
    Get targetState;
    If (sourceState.isTop)&&(targetState.isTop)
        Exit;
Show error;
```

Corollary 36 Synchronous communication of objects implies transition to a state which cannot be left except through a state transition associated with the return signal.

Interactions between objects can be synchronous or asynchronous. We discuss synchronous communication in this corollary and leave asynchronous to the next corollary. In UML, "A synchronous invocation is the transmission of a request from a requestor to a target object in which the requestor waits for a reply from the invoked execution. The invoked execution may supply return values, but even if there are no return values, the requestor waits for a reply indicating that the invoked execution has completed" (OMG, 2003, p2-313). In addition, "while the execution of a procedure invoked by a synchronous invocation is progressing, the requesting action execution is 'blocked.' This does not require any special mechanisms; the synchronous action execution is simply in the 'executing' state until the invoked procedure execution terminates and returns" (OMG, 2003, p2-315). Thus, synchronous communication of objects implies a transition to a state which cannot be left except through a state transition associated with the return signal.

This corollary is fully deduced from UML specification. Actually, it is a rewording of UML. Thus, it is fully covered by UML and all CASE tools implementing UML specification should follow this corollary. There is no need to implement it.

Corollary 37 Asynchronous communication of objects with expected response implies the existence of at least one state transition caused by the object acted upon, signifying the return interaction after the state transition signifying the original communication.

"An asynchronous invocation is the transmission of a request from a requestor to a

target object in which the requestor continues execution immediately, without waiting for a reply" (OMG, 2003, p2-312). The motivation of this corollary is that even the acting object does not wait for a reply, the acted object may still return a reply, which is considered the "expected response" (Evermann, 2003, p127). However, UML specifies that "it is permissible to asynchronously invoke a request to a procedure that eventually issues a reply; the reply message is simply discarded" (OMG, 2003, p2-312). People may argue that if the reply message actually sends some useful information, how can it just be discarded? The answer is that "if the invocation is repliable, a subsequent reply by the invoked execution is transmitted to the requesting object as an asynchronous request" (OMG, 2003, p2-322). That is, the response actually invokes another new communication. This is because "the target object might later communicate to the requestor, but any such communication must be explicitly programmed and it is not part of the asynchronous invocation action itself" (OMG, 2003, p2-312). As we can see, there is no "expected response" for asynchronous communication. The motivation of this corollary is incorrect. We propose not to adopt it.

Corollary 38 The final state transitions of any method implementing an operation that may be invoked through a call action must cause a return action.

There are several types of action, for example, signal action and call action. Among these actions, call actions are synchronous and require response: "The (call) action

execution waits until the effect invoked by the request completes and returns to the caller. When the execution of a procedure is complete, its result values are returned to the calling execution" (OMG, 2003, p2-324). That is, there exist return values for a call action. When a method implementing an operation is invoked by a call action, the method must return values at the end of its procedure. This is because "during the execution of a synchronous invocation, the invoked procedure execution must have sufficient information to be able to awaken the invoking action execution when execution of the invoked procedure is complete" (OMG, 2003, p2-320). The final state transition is the last behaviour of a method procedure. Thus, it causes a return action to return values.

The implementation approach searches collaboration diagrams for call actions. For an interaction, whose message includes a call action, the program gets the object and method being called. Then it checks state chart diagram attaching to this method. The final transition and its action will be retrieved. If the action is a return action, the program exits; otherwise, it shows a violation warning. This is expressed in the following:

```
Get collaborationDiagram;
Get interaction;
Get message;
Get callAction;
If callAction==null
    Exit;
Get method1;
Get receiver;
Get classDiagram;
Get classes;
While classes.hasNext()
```

```
If class==receiver
    Get method2s;
    While method2s.hasNext()
        If method2==method1
            Get stateChart;
            Get transition;
            Get target;
            If target==finalState
                Get action;
                If action==returnAction
                    Exit;
Show error;
```

Corollary 39 For the state machine of a method to contain a state transition whose effect is a return action, there must exist a corresponding state transition in a state machine of some other object whose effect is a corresponding call action.

As we discussed in the last corollary, an interaction with a call action works like this: Method A implementing operation B of object C calls method D implementing operation E of object F. When the call happens, object C waits for return from object F, while F begins running method D. When D completes, its return action returns values to method A and A continues. Corollary 38 specifies that there must be a corresponding return action in the called object for every call action in the caller object. In reverse, since methods with return actions are invoked by call actions, there must be a corresponding call action in the caller object for every return action in the called object. According to Corollary 32, there is a corresponding transition in both caller and called objects. The transition of the called object causes a return action, which is specified by Corollary 38. The transition of the

caller object causes a call action. Through the last corollary and this one, three diagrams are associated to ensure consistency. These diagrams are state chart diagrams of both caller object and called object, and collaboration diagram specifying the interaction between them.

The implementation approach is to check whether there is a transition with a call action corresponding to every transition with a return action in caller class and called class, respectively. The program first gets a class method and its state machine. Then it checks the transition leading to the final state. If there is no return action attaching with that transition, the program exits; otherwise, it reads the return action and gets the caller class and method. Then it searches state machine attaching to the caller method. If there is a transition with call action, which invokes the called method, the program exits; otherwise, it shows a violation warning. This is expressed in the following:

```
Get class1;
Get method1;
Get stateMachine1;
Get transition;
Get target;
If target==finalState
    Get action;
    If !(action==returnAction)
        Exit;
    Get class2;
    Get method2;
    Get stateMachine2;
    Get transitions;
    While transitions.hasNext()
        Get callAction;
```

```
Read callAction;
If (class==class1)&&(method==method1)
    Exit;
Show error;
```

Chapter 7 Implementation in ArgoUML

The previous three chapters discussed Evermann's rules and provided a general implementation approach. However, they are not CASE tool specific. That is, all those approaches can work for any UML CASE tools. This chapter actually discusses the implementation of the proposed approach in a specific tool: ArgoUML. The approaches presented in earlier chapters are based on assumption of a fully functional UML CASE tool. However, ArgoUML is an open source project, and is still being developed. There are several UML functions that do not work properly in ArgoUML as well as some bugs. That means that not all of the approaches discussed in previous chapters can actually be implemented in the current ArgoUML. In this situation, we give the alternative solutions for those rules which cannot be actually implemented in the current ArgoUML. This is expressed in the following structure. First, we explain how our implementation works in ArgoUML and related issues. Then, we introduce the communication structure between our system and users. Finally, we give the alternative approaches for rules that cannot be actually implemented in the current ArgoUML.

7.1 How ArgoUML criticizes

The critiquing system of ArgoUML has the following features: Critics for rules are agents that continually analyze the design. The system can present feedback as soon as violations arise. In addition, feedback is retracted as soon as violations are fixed. These features are realized through four components: Criticism control mechanisms, Communication Structure management, Inference engine, and Knowledge base.

7.1.1 Criticism Control Mechanism

The criticism control mechanisms controls critique execution to keep feedback relevant and timely to the modeler's activities. It processes all critiques in each task and disables selected critiques for fixed time of periods. The following is a piece of code implementing these functions and the screenshot of the 'Criticism Control Center.' (For explanation of specific code please refer to www.argouml.org.)

Process all critics in each task:

```
Agency.register(crR20, classCls);
Agency.register(crR20, assocCls);
Agency.register(crR21, stateVertexCls);
Agency.register(crR22, transitionCls);
```

Disables selected critics for fixed time periods:

```
public void snooze() {
if (_snoozeAgain.after(getNow())) _interval = nextInterval(_interval);
else _interval = _initialIntervalMS;
long now = (getNow()).getTime();
_snoozeUntil.setTime(now + _interval);
_snoozeAgain.setTime(now + _interval + _initialIntervalMS);
cat.info("Setting snooze order to: " + _snoozeUntil.toString());}
```

public void unsnooze() {_snoozeUntil = new Date(0); }

Screenshot:

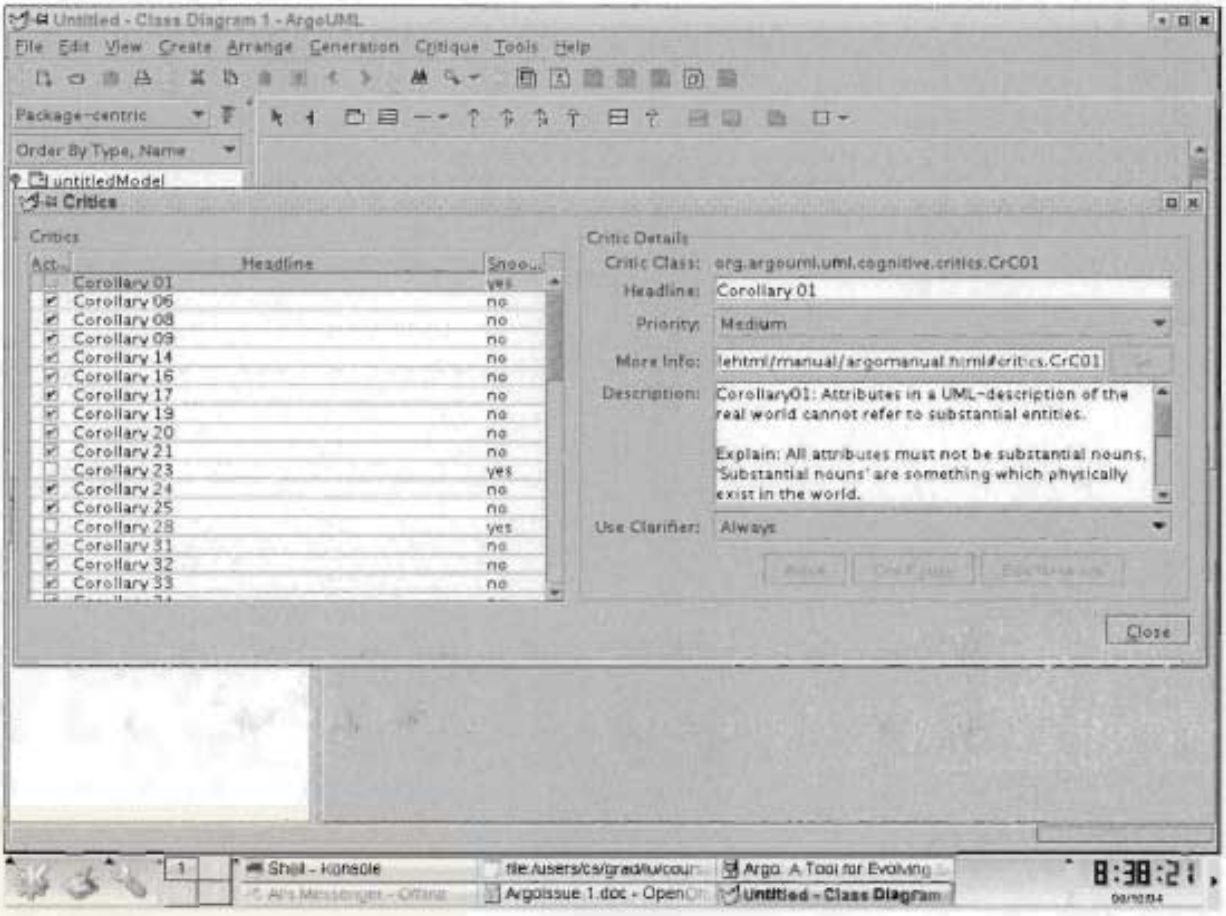

Figure 7.1 Criticism Control Center

7.1.2 Communication Mechanisms

Communication mechanisms present violations to users. It is also responsible for the interaction between the machine and users. There are four structures for the system to communicate with users. We will explain these in detail in Section 7.3 (Please refer to Table 8.1 Categories of feasible (to implement) rules for details). Following is a piece of

code and the screenshots of the 'High-Priority' communication mechanism. (Please refer to discussion for R18 for code comment.)

The 'High-Priority' communication mechanism:

```
//if already checked
if (_checked.contains(dm)){
    switchOn = 1;
    return PROBLEM_FOUND;}
else{
    //user select
    try{
        //show error
        Object[] options = { "Put into Reminder", "Explain" };
        int in = JOptionPane.showOptionDialog(null, "Class ' "+nameStr+"' is not
        specialized and cannot be declared abstract.", "Violate Rule 18",
        JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE,null,
        options, options[0]);
        if (in == 1){ JOptionPane.showMessageDialog(null, "Explain: Only a
        super-class being specializeded can be declared as abstract.\n \nExample: None",
        "Rule18: A class that is not specialized cannot be declared abstract.",
        JOptionPane.INFORMATION_MESSAGE);}}
    catch(Exception ex){ System.out.println(ex); }
    //add this class to checked vector
    _checked.addElement(dm);
    switchOn = 1;
    return PROBLEM_FOUND;}
```

Screenshot:

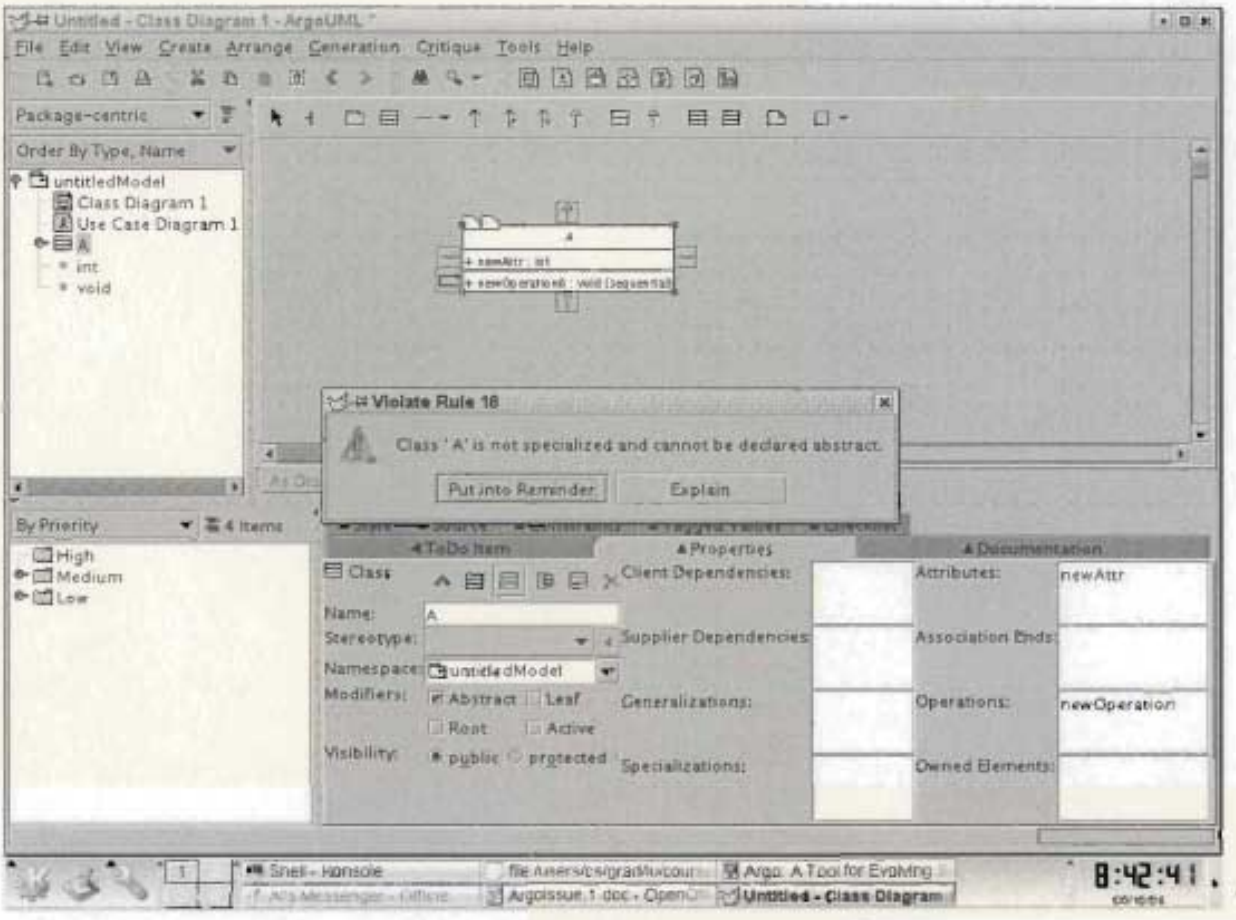

Figure 7.2 Show Violation of Rule 18

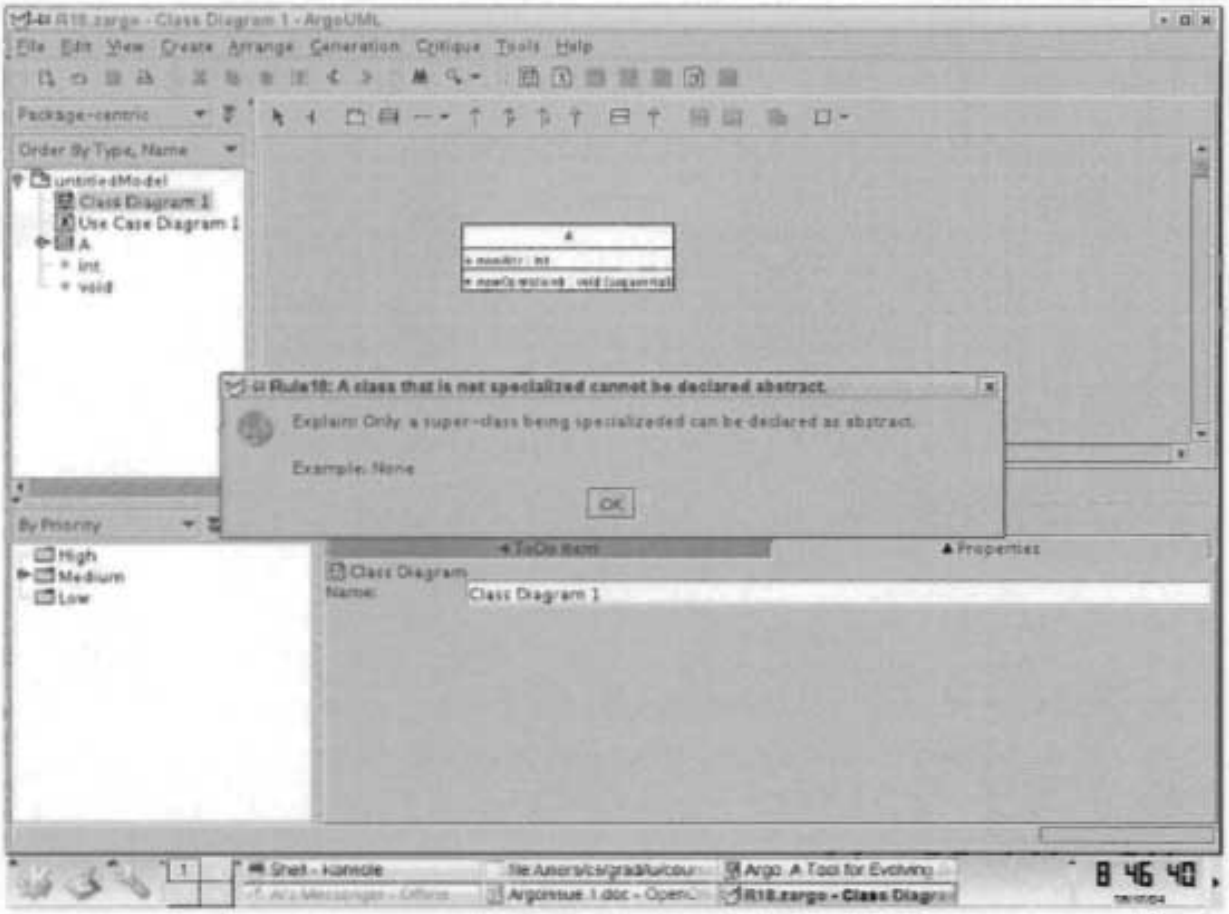

Figure 7.3 Explain Violation of Rule 18

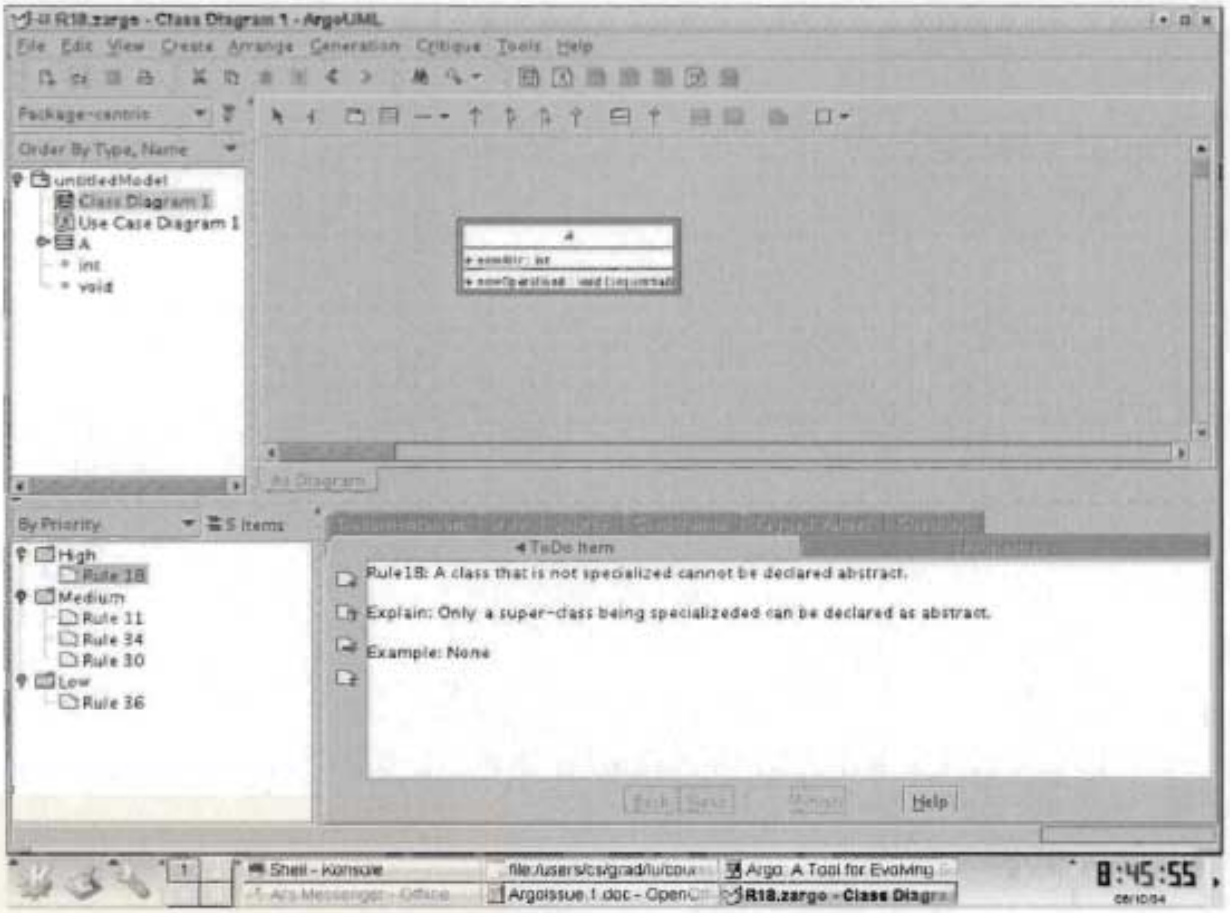

Figure 7.4 Violation in Reminder List

7.1.3 Inference Engine

The inference engine is the control mechanism that applies knowledge of rules presented in the knowledge base to the specific diagram as input data to arrive at a conclusion of firing a violation or not. This is also a key component of an expert system. Following is a piece of code of the inference engine.

Inference engine:

> */** Examine the given Object and Designer and, if appropriate, produce one or more violations and add them to the offending modeling material's and the modeler's ToDoList. By default this is basically a simple if-statement that reites on predicate() to determine if there is some appropriate feedback, and toDoItem() to produce the ToDoItem.*/*

```
public void critique(Object dm, Designer dsgr) {
    if (predicate(dm, dsgr)) {
        _numCriticsFired++;
        ToDoItem item = toDoItem(dm, dsgr);
        postItem(item, dm, dsgr);}}
```

7.1.4 Knowledge Base

The knowledge base is the implementation set of all ontological rules and corollaries and is the core part of the system. Following is a piece of code of the knowledge base and the screenshot of the knowledge base list.

Knowledge base:

```
// File: CrR18.java
// Classes: CrR18
// Author: Shan, lu@cs.mun.ca
// $Id: CrR18.java,v 0.1 2004/4/22 15:43
package org.argouml.uml.cognitive.critics;
import java.util.*;
...
import java.util.regex.Pattern;
public class CrR18 extends CrUML {
int switchOn = 1;
public static Vector _checked = new Vector();
public CrR18()    {
setHeadline("Rule 18");
setPriority(ToDoItem.HIGH_PRIORITY);
addSupportedDecision(CrUML.decNAMING);
setKnowledgeTypes(Critic.KT_SYNTAX);}
public boolean predicate2(Object dm, Designer dsgr){
if (switchOn == 0)
return NO_PROBLEM;
//Pause the paralle critic for this rule
switchOn = 0;
...
//'High-Priority' communication structure
```

```
...
switchOn = 1;
 return NO_PROBLEM;}
} /* end class CrR18 */
```

Screenshot:

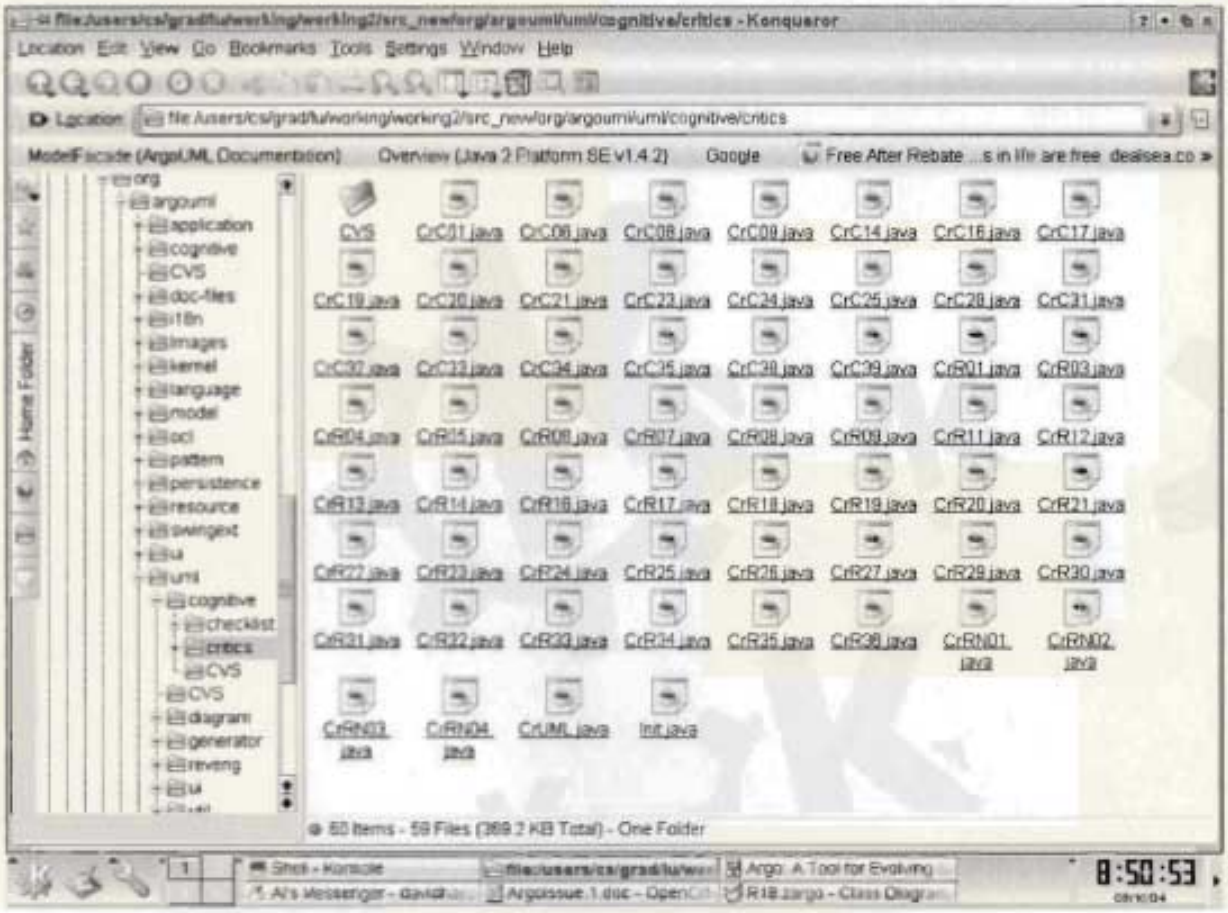

Figure 7.5 Knowledge Base List

The above four components work in the following process: When the ArgoUML starts, the criticism control mechanisms activate and initialize rules and corollaries. When user develops UML diagrams, the inference engine monitors users activities in real-time. It checks diagrams according to the knowledge base. Once a violation is matched with rules / corollaries in the knowledge base, the communication mechanisms present feedback or

interact with the modeler.

7.2 Problems in ArgoUML

ArgoUML is not a complete business product, but an open source project for research purposes. It is still being developed and not all functions have been realized. Some realized components currently do not work properly. The following are problems known in ArgoUML currently.

Known Incompatibilities	
Here is a list of features, which are not implemented yet and can be seen as a violation of the UML specification (1.3)	
All Diagrams	
1	*Qualifiers are missing.*
2	*Optional directional triangle missing on association names.*
3	*List of provided multiplicities is not complete.*
Class Diagram	
4	*Association classes are missing. Though ArgoUML can read association classes from XMI files, it cannot display them correctly. That is due to a flaw in GEF, which needs a major rework.*
5	*Parameterized classes are missing.*
6	*Notes to any model element not implemented.*
7	*Cannot add a method to an operation.*
8	*Cannot show collaborations on a class diagram.*

	Object Diagrams
9	*Currently not implemented, use a Deploymentdiagram.*
10	*Attributes cannot have attribute values.*
	Activity Diagrams
11	*Swimlanes missing.*
12	*Signal receipt/sending missing.*
	Sequence Diagrams
13	*Missing completely from ArgoUML 0.14. (They were present in 0.10.1)*
	Statechart Diagrams
14	*Regions are drawn and handled as ordinary composite states.*
15	*There is no distinction between kinds of Actions.*
16	*Distinction between kinds of Events is only partly implemented.*
17	*A "junction" is only partially supported*

Please note: *This list is not complete!*
(Tigris, 2004, http://argouml.tigris.org/documentation/umlsupport)

Table 7.1 Incompatibilities of Current ArgoUML

These incompatibilities will affect us when our implementation relates to these components. To solve this problem, we try to use alternative solutions, instead of just skipping rules or corollaries. We will discuss the specific issues in related rules or corollaries in Section 7.4.

7.3 Four Communication Mechanisms

There are four structures for the communication between critiques and users.

The first structure is for the "MustBe" rules. "MustBe" rules are rules which specify that something must be or must not be in a particular way; for example, Rule 9 states that object ID's must not be modeled as attributes. For this type of rule, the violation warning must be shown immediately to prevent further problems. That is, as soon as the rule is violated, the system should alert the user. For some violations, for example violating Rule 1 (details can be found in Rule 1 of Section 6.4), which can be easily corrected by the user, we popup a dialogue informing the user of the violation and reset the related space so that the user can re-input correct information. We call this 'Critique' communication structure.

However, not all violations are easy to correct. We cannot simply delete the violation component. For example, when a diagram violates Rule 3 (details can be found in Rule 3 of Section 6.4), we cannot delete the whole association class. Since this kind of violations are difficult to correct, a user may need a long time fixing it or even do not want to fix it immediately. In this case, the system continues popup dialogues, showing something wrong here and something wrong there and the user may become annoyed. To prevent this, we adopt the 'High-Priority' communication structure. Once the input is considered as a violation, the system will popup a dialogue to warn the user. However, we do not clear anything in diagrams. Instead, we keep the input there and give the option for the user to correct it later or even not to change it at all (Users can also dismiss violations and snooze

critiques. We will explain these structures at the end of this section). To prevent users from forgetting it later and also to avoid continually popping-up violation dialogues, we put this violation into the high-priority reminder list. That means the popup dialogue will only be shown once for each instance of a rule violation. The reminder will continue to examine the diagram in real time. However, all this is done in the background. If the user does not change the input, the violation message will be shown in the reminder list all the time. It does not affect the user's job. The user only needs to check the list at any time, s/he will be aware of what violation s/he has not corrected yet.

The 'Medium-Priority' communication structure is for "MustHave" rules. "MustHave" rules specify that something must be or must not be included in a diagram; for example, Corollary 7 (details can be found in Corollary 7 of Section 6.4), which states that an association class must possess at least one attribute. For this kind of rule, it is hard to say whether the user forgets having something, or not input it yet. On the other hand, the examination is in real time. For rules of the form "A must have B", as soon as A is created, the system will find there is no B in A before a user can input B. In this case, we should not let the system popup a dialogue saying. A must have B. Instead, we put all these kinds of violations into the medium-priority reminder list directly. The system does not popup any dialogue at all. Using this structure, we can keep track of the violation without annoying users.

The 'Low-Priority' communication structure is not for violation warnings, but

consists of reminders. The newly proposed structures by Evermann do not exist in current UML. Also, some structures in UML have not been implemented or are not working properly in ArgoUML. There is no way for the program to detect violations for this kind of rule. In this case, the program will only popup reminders to users. Then it will put the reminder into the low-priority reminder list. The user can dismiss the reminder at any time.

The system may make mistake of some complex rules. For example, attributes 'name' in 'Customer' and 'BookStore' classes are considered the violation of Rule 3 (details can be found in Rule 3 of Section 6.4). In this case, ArgoUML also offers mechanisms of dismissing violations and snoozing critiques. If the system shows an unexpected critique, the user can simply click the "Dismiss" button and the critique will disappear. If the user wants to disable a rule critique temporarily, he / she can simply click the "Snooze" button and the critique will be deactivated temporarily. If the user wants to disable a rule critique forever, he / she can simply deselect the corresponding critique in the setting menu and the critique will no longer be activated. Since this is a critique expert system, we also have a mechanism of inquiry users for required information. When our program cannot get enough information it inquiries users. For example in Rule 5 (details can be found in Rule 5 of Chapter 3), the program inquires users whether each attribute is created by the interaction creating the association class.

We will explain these four communication structures through rule examples in the discussion below. Specifically, we will explain the 'Critique' structure in Rule 1, the

'High-Priority' structure in Rule 3, the 'Medium-Priority' structure in Rule 7, and the 'Low-Priority' structure in Rule 21.

7.4 Alternative solution

This section will discuss alternative approaches for rules that cannot be implemented in the current ArgoUML. We still discuss this by order of Evermann's rules. For those rules which can be implemented by the previous approaches, we do not discuss them again here.

Rule 1 Only substantial entities in the world are modeled as object.

In the implementation, we have two text files, "subnouns" and "nonsubnouns" to save the substantial nouns and non-substantial nouns, respectively. We have a class "CrR1" (CrR* is the implementation java code, please refer to Rule 1 in Chapter 4 for details) to match the object name in the two text files. For the object, the "CrR1" not only examines the object name, but also examines the class name. Because a class is a group of objects, they normally have the exact same name. For the GUI, if a name is found in the "nonsubnouns", the system will popup a message window, indicating the error. The user can choose "Ok" to reinput a name or choose "Explain" to read the explanation and examples of this rule. If a name cannot be found in either "subnouns" or "nonsubnouns", the system will popup a message dialogue, asking the user whether the name is substantial or not. The user can choose "Yes" or "No" to save the name in "subnouns" or

"nonsubnouns", respectively, or choose "Explain" to see the explanation and examples of this rule. In the query window, we also give users an "others" option, in case the user input a string which is neither substantial noun nor nonsubstantial noun. This is because we use the 'Critique' communication structure for this rule. When a non-substantial noun is input, the system will popup a violation warning as well as clearing string of the non-substantial noun in the diagram and waits for the new input string. That is, the user cannot continue until he / she inputs a substantial noun. However, users may not agree with the violation warning. In this case, he / she can choose the "Other" option to continue his / her job.

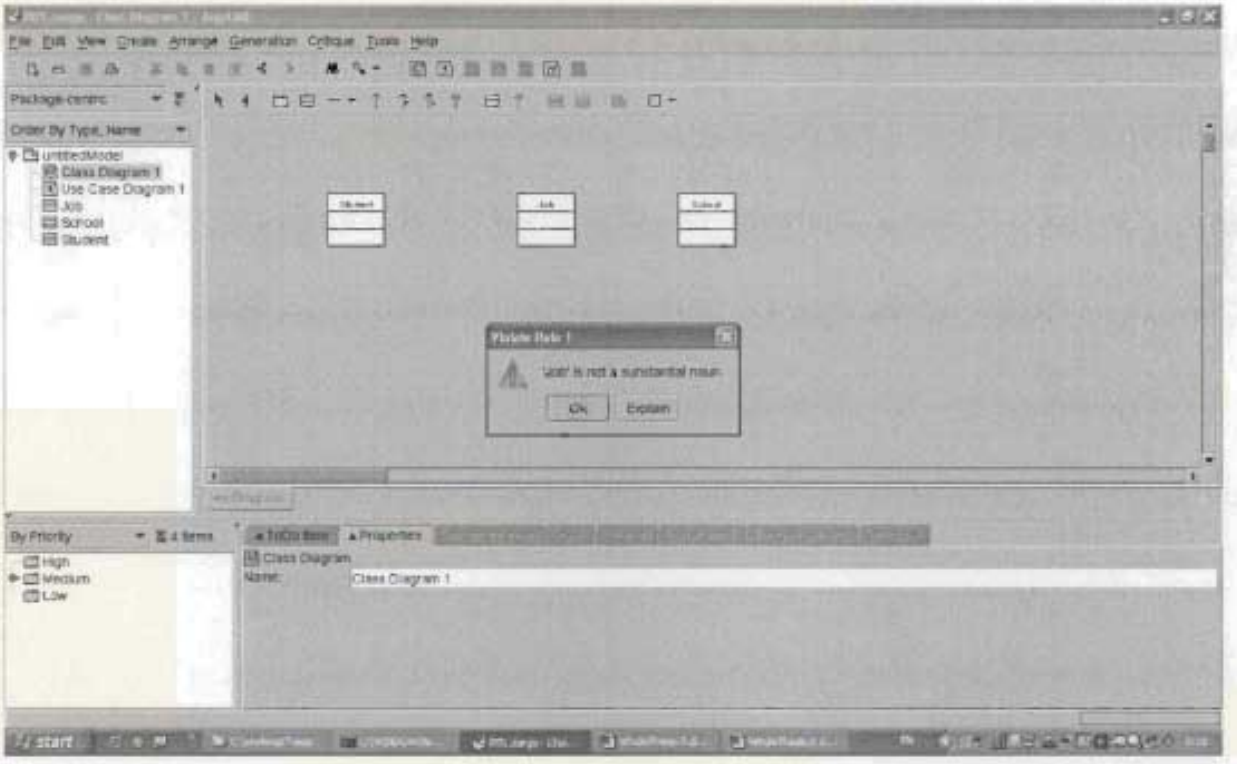

Figure 7.6 Violation of Rule 1

Rule 2 Ontological properties of things must be modeled as UML-attributes.

The approach for this rule in Chapter 3 has already been shown realistic for Rule 1; however, we meet some difficulties in this rule. The program needs to get all the text strings from all diagrams, for example, class name, class attribute, class operation, association, and even strings in sequence diagram, collaboration diagram, and state machine. However, ArgoUML currently does not support returning all text from all diagrams. Actually, ArgoUML is using the class library of the NUSoft Company, which does not release the source code. The other problem is that this implementation will keep searching all strings in all diagrams and matching with databases. This will cost lots of CPU time, which is considered inefficient. According to the above reason, we decide to implement corollaries of this rule. This does not mean that Rule 2 is not realistic. There is no principle problem here. It is neither Rule 2 nor its implementation approach is infeasible, but the development environment has a problem currently.

Corollary 1 Attributes in an UML-description of the real world cannot refer to substantial entities.

There is a "CrC1" class to fulfill this corollary. When the input string is found in the nonsubnouns file, the program returns no problem. When the input string is matched with a string in the subnouns file, the program will show a violation warning and explain this corollary to users. Also, if an object cannot be found in our database, the program will interact with the user by asking whether that object is substantial or not. The program will

remember the new substantial or nonsubstantial noun in its database. Thus, the system has the ability to learn by remembering newly added nouns. The 'Critique' communication structure is used for this corollary.

Rule 3 Sets of mutual properties must be represented as attributes of association classes.

This rule adopts the 'High-Priority' communication structure. First, this rule is a "must be" rule. The system should inform users as soon as the violation appears. Thus, we design this examination to popup a dialogue when it matches the same attribute in two associated classes. However, this violation cannot be corrected immediately. The user needs to create an association and put this attribute in that association class. The user needs at least several minutes to do this. Our examination system is in real time and it will fire another examination thread for this rule within several seconds. It is very possible that the user has not enough time to correct it before the next thread is fired. Since the correction is complex, another possibility is that the user does not want to correct this right away and will correct it later. In these two cases, the next thread will still find the violation and popup a dialogue. That is, a dialogue will popup every few seconds, which is unacceptable. On the other hand, the system may even make mistakes here. Consider the customer and bookstore example. Both customer and bookstore may have their intrinsic attribute "name", which refers to customer name and bookstore name, respectively. These "name" attributes are not a mutual attribute between customer and bookstore and thus should not

be modeled in an association class. However, this situation still satisfies the violation of this rule in the program. Because of the above situations, the popup dialogue will only be shown once. After that, if the violation is still considered by the program, the information will be shown in the reminder list. If the violation is correct, the user will not forget it. If the violation is incorrect, the user can just leave it there or simply dismiss the violation, or even snooze the examination for this rule. Now, how about if a violation of another association appears? To prevent the new violation being thrown into the reminder list directly, we create a vector to record the associations that have already violated this rule and still popped up dialogues for new violations. In this way, we guarantee that each violation will popup the dialogue once and only once.

Corollary 2 An association class cannot represent substantial entities or composites of substantial entities.

This is a "must be" corollary and we can use the 'Critique' communication structure here. Because this is a corollary of Rule 3 and we have already implemented Rule 3, there is no need to implement the corollary. We just proved the corollary and gave the implementation approach in Chapter 3.

Corollary 5 An association class cannot possess methods or operations.

This corollary is part of Rule 4, we will implement it in Rule 4.

Corollary 6 An association class cannot be associated with a state machine.

If this corollary is finally adopted by UML specification, all CASE tools will disable this feature and there is no need to write a program to examine this. Currently, ArgoUML allows this feature. So, we use an alternative way to implement this corollary. The implementation of this corollary relates to two diagrams. First, the examination needs to get all association class names in a class diagram. Then, the state machine names need to be retrieved from a statechart diagram. These names will be matched. If an association class name and a state machine name are found to be the same, the program will fire the violation warning. This is expressed in the following:

```
Get associationClass;
Get acNames;
While acNames.haveNext()
    Get stateMachine;
    Get smNames;
    If acName = anName
        Show error;
    Else exit;
```

However, the association class and state machine are in different diagrams. It is possible that the user occasionally uses the same names. If the program warns the user just by matching a same class name and statemachine name, it may not be accurate. In the detailed implementation, we decide to use another more accurate approach. We realized that a statemachine is associated with a class. That is, there is no independent statemachine that exists. So, we can get all elements owned by an association class. If any element is a

statemachine, then the program shows a violation warning. This is expressed in the following:

```
Get associationClass;
Get ownedElements;
While element.isStatemachine()
    Show error;
Exit;
```

For communication structure, this is a "must be" corollary and we use the 'High-Priority' structure for this corollary.

A problem here is that the ArgoUML currently does not support association classes. That is, one cannot model an association class. We adopt the following solution for this problem.

We use ordinary classes to represent association classes. To distinguish them from those "real" ordinary classes, users must add a stereotype of "associationClass" of the classes. We do not use regular association to connect association classes to ordinary classes. Instead, we use the solid line in the "Select Tool" menu, which currently has no meaning in ArgoUML. The reason will be explained in Corollary 8.

Corollary 7 An association class must possess at least one attribute.

From the above discussion in chapter3, we can see that this corollary is covered in Rule 3. Thus, there is no need to implement it.

Corollary 8 An association class must not be associated with another class.

The examination only needs to check association classes to see whether they own any association ends. If the program finds any, it shows a violation warning. This is expressed in the following:

```
Get associationClasses;
Get associationEnds;
While associationEnds.haveNext()
    Show error;
Exit;
```

This is a "must be" corollary and we use the 'High-Priority' communication structure here.

However, the current ArgoUML does not support association classes and we use the stereotype to identify association classes. If we use associations to connect association classes and ordinary classes, we could not use the above approach. In this case, association classes have to own association ends. The program cannot tell which association is a violation. Thus, we use the solid line in the "Select Tool" menu to connect ordinary classes and association classes. Note that even though we can see the connection between association classes and ordinary classes on the diagram, there is actually not any relationship between these classes. ArgoUML does not give any role to symbols in the "Select Tool" menu.

Corollary 9 An association class must not participate in generalization relationships.

The examination can simply check all association classes to see whether there is any generalization. If the program finds any, it shows a violation warning. This is expressed in the following:

```
Get associationClasses;
Get generalization;
While generalization.hasNext()
    Show error;
Exit;
```

We can also use another approach. First to get all generalization relationships, then the program checks both sides of these relations. If any side is an association class, it shows a violation warning. This is expressed in the following:

```
Get generalization;
Get ends;
If end=associationClass
    Show error;
Exit;
```

We use the latter approach for this corollary. This is a "must be" corollary and we use the 'High-Priority' communication structure.

Rule 4 If mutual properties can change quantitatively, methods and operations that change the values of attributes of the association class must be modeled for one or more of the classes participating in the association, objects of which can effect the change, not for the associations class.

This rule is a "must be" type and we use the 'High-Priority' communication structure here.

Rule 5 An association class represents a set of mutual properties arising out of the same interaction.

The program for this rule can only work by interacting with users. It separates a complex problem to several simple questions. The user only needs to answer these questions and the program will examine diagrams based on the user's answer. We use the 'High-Priority' communication structure.

Rule 6 A composition relation must not be modelled.

This is an alternative implementation. The examination just checks each association end of each class. If any composition association is found, it shows a violation warning. This is expressed in the following:

```
Get classes;
Get associationEnds;
If associationEnds=composition
    Show error;
Else exit;
```

For the communication structure, we use the 'High-Priority' one.

Rule 7 Every UML-aggregate must possess at least one attribute which is not an attribute of its parts or participate in an association.

This is a "must have" rule so we use the 'Medium-Priority' communication structure

here: Immediately after the user creates an aggregate class, the system will find that there is no additional attribute in the aggregate class than in its parts. The system fires a violation warning. However, the fact is that the user has no time to input an attribute yet. To prevent annoying users, the 'Medium-Priority' communication structure does not popup a window, but only show this violation in the reminderList. If the user models an additional attribute in a later time, the warning will disappear.

Rule 8 All UML-classes must possess at least one attribute or participate in an association.

This is a "must have" rule and we use the 'Medium-Priority' communication structure for it.

Rule 9 Object ID's must not be modelled as attributes.

This is a "must be" rule. However, the program needs to interact with users by asking them to confirm the examination. We use the 'High-Priority' communication structure.

Rule 10 The set of attribute values (representing mutual and intrinsic properties) must uniquely identify an object.

The ArgoUML environment does not support identification of different objects. That is, there is no object diagram and no place to save object attribute values. This is not a

problem of ArgoUML either, because UML1.5 specifies that "Tools need not support a separate format for object diagrams" (OMG, 2003, P3-35). Thus, we could not implement this rule in ArgoUML. However, we have shown that this rule is feasible to implement.

Rule 11 Every attribute has a value.

As we explained in Rule 10, ArgoUML does not support object diagrams; the examination can only check class diagrams. Also, since the ArgoUML already allows users to specify multiplicity of "0" and value of "null" for attributes, we use an alternative way to implement this rule. We get all attributes and check their values. The examination shows a violation warning until every attribute's initial value is specified. Also, the program checks attribute multiplicities. If any attribute multiplicities of "0" is found, it shows a violation warning. Currently, ArgoUML only has three options of attribute multiplicities: "1", "0..1", and "1..*". This is expressed in the following:

```
Get class;
Get attributes;
While attributes.hasNext()
    Get initial value;
    Get multiplicity;
    If initial value=null||multiplicity=0
        Show error;
    Else exit;
```

Since this is a must have rule, we use the 'Medium-Priority' communication structure.

Rule-New 01 Every UML attribute can only have a single value. An attribute of

multiplicity of "N" should be decomposed into "N" attributes of multiplicity of "1." An attribute multiplicity of "0" should be generalized as a super-class not possessing the attribute.

In the current stage, we use an examination to enforce this rule in the ArgoUML. First, the program gets attribute multiplicities. When finding any multiplicity not equal "1", it shows a violation warning. This is expressed in the following:

```
Get class;
Get attribute;
Get multiplicity;
If multiplicity==1
    Exit;
Show error;
```

We use the 'High-Priority' communication structure for this corollary.

Rule 12 Classes of objects that exhibit additional behavior, additional attributes or additional association classes with respect to other objects of the same class, must be modeled as specialized sub-classes.

The 'High-Priority' communication structure is used for this rule.

Rule 13 Every UML-aggregate object must consist of at least two parts.

This is a must have rule. Thus, we use the 'Medium-Priority' communication structure.

Rule 14 An instance of a class that by virtue of additional aggregation relationships acquires emergent properties or emergent behavior must be modeled as an instance of a specialized class which declares the corresponding attributes and operations.

The 'High-Priority' communication structure is adopted for this rule.

Rule 16 Attributes with class scope should instead be modeled as attributes of an aggregate representing the objects of the class.

This is a "must be" rule and we use the 'High-Priority' communication structure here. In AgroUML, a "static" box in an attribute property is used to indicate whether an attribute is instance-scope or not. Thus, we check whether the "static" box is selected for instance-scope attribute.

Rule 17 If a class that is specialized is declared as abstract, the specialization must be declared to be 'complete'.

The ArgoUML currently does not support declaring "complete" for generalizations. We use the "stereotype" to declare complete. ArgoUML allows users to model several specializations for a single super-class. The stereotype is defined for each specialization independently. Thus, if all super-class objects are members of one of the sub-classes, each specialization should be defined as complete. If a specialization is defined as complete, it does not mean that this single specialization is complete; but that all specializations of the

super-class together are complete. The 'High-Priority' communication structure is used here.

Rule 18 A class that is not specialized cannot be declared abstract.

The 'High-Priority' communication structure is adopted for this rule.

Rule 19 A specialized class must define more attributes, more operations or participate in more associations than the general class.

The 'Medium-Priority' communication structure is used for this rule.

Rule 20 Every ordinary association must be an association class.

As mentioned before, ArgoUML currently does not support modeling of association classes. In the implementation of previous rules, we use the stereotype of "associationClass" to identify association classes. However, even the structure looks fine, the solid line connecting association and association class does not include any information. Our alternative solution for association class cannot return an association and association class pair. That means that the system could not know whether an association class connects with an association. We have to force users to specify the same name for both the association and the association class. In this way, our program will consider an association and association class connected if they have the same name. The alternative

approach is expressed in the following:

```
Get associations;
If generalization||aggregation
    Exit;
Get assoName;
Get associationClass;
Get assoClassName;
If assoClassName==assoName
    Exit;
Show error;
```

According to ArgoUML structure, only one model element can be passed to one examination thread. Because association and class are different types of model elements, we cannot get both association and association class in one examination process. Thus, we have to depend on the parallel examination process. In process threads, which get an association class element, the class name is added into a vector. In process threads, which get an association element, the association name will be matched in the vector. In the latter type of threads, if an association name cannot be found in the vector, the violation warning will be fired. The ideal working processes are expressed in the following: Firstly, the program gets all association class elements and adds them to the vector. Then, it begins to get associations and match their names in the vector. That is, to finish all the second type of threads before the first type of threads begin. However, because the parallel threads get model elements randomly, we cannot guarantee the ideal situation happening. If the computer running this program is too slow, it may show incorrect violation warnings. All these problems are because ArgoUML currently does not support association class. In a

fully functional UML CASE tool, the approach will work. This is a must have rule and we use the 'Medium-Priority' communication structure for it.

Rule 21 A UML-state represents a specific assignment of values to the attributes [of ordinary classes] *and attributes of association classes of the objects for which the state is defined.*

Because there is no mechanism connecting states and attribute values in current UML, We use an alternative implementation for this corollary in ArgoUML. The program gets all states and state machines they belong to. Then it gets the model element owning the state machine and reminds users to follow this rule. If the user does not want to keep this reminder, it can be dismissed and will not appear in the reminder list again. However, because state machines are not allowed for signals (we will discuss this in Corollary 25), the program does not check signal state machines. Actually, the program cannot detect violations, but only remind users for each state. Thus, the 'Low-Priority' communication structure of reminder is adopted here. This is expressed in the following:

```
Get states;
Get stateMachine;
Get smOwner;
If !(smOwner==signal)
    Show reminder;
    Put to reminderList;
Exit;
```

Corollary 14 A UML-transition must change the value of at least one attribute used to define the state space.

Even though Evermann proposes a meta-model of associating states with attributes, this is not in the current UML specification. Thus, the program will not know whether a transition actually changes attributes with that state space. We implement an alternative way to enforce this corollary in ArgoUML. The program gets all transitions and reminds modellers to enforce this corollary. However, the modeller can dismiss this reminder and it will not show in the reminder list. The 'Low-Priority' communication structure is adopted here. This is expressed in the following:

Get transitions;
Show reminder;
Put to reminder;
Exit;

Rule 22 For every level of refinement of a state C, there must be an additional set of attributes in the class description or in participating association classes that change as the object transitions among the sub-states.

Based on the same reasoning as in the last corollary, we implement an alternative way to remind users following this rule in ArgoUML. The program gets transitions and their source states and target states. Then it checks whether the source state and target state belong to a same composite state. If this is the case, the program shows the reminder to users. The 'Low-Priority' communication structure is adopted here. This is expressed in

the following:

```
Get transition;
Get sourceState;
Get targetState;
If sourceState.container==targetState.container;
    If sourceState.container==top
        Exit;
    Show reminder;
    Put to reminder;
Exit;
```

Corollary 16 Concurrent sub-states require mutually disjunct sets of additional attributes in the class description or in participating association classes.

Based on the same reasoning as in Corollary 14, we implement an alternative way to remind users following this corollary in ArgoUML. In addition, ArgoUML does not strictly follow the notation of concurrent sub-states (region) in UML specification. It only supplies a "concurrent" option for a whole composite state. We could not set or get concurrent information for each state. Thus, we deem all sub-states concurrent if the whole composite state is set to concurrent and vice versa. The program first gets a sub-state. If the composite super-state is not declared concurrent, then it exits; otherwise, the program reminds the user. The 'Low-Priority' communication structure is adopted here. This is expressed in the following:

```
Get sub-state;
Get super-state;
If !(super-state==concurrent)
    Exit;
```

Show reminder;
Query user;
If answer=='Y'
    Put to reminder;
Else exit;

Rule 23 Guard conditions on transitions from the same state to nonconcurrent sub-states must be mutually disjunct.

Based on the same reasoning as in Corollary 16, we deem all sub-states concurrent if the whole composite state is set to concurrent and vice versa. The 'High-Priority' communication structure is used here.

Rule 24 Action states are super-states of a set of sub-states. The object transitions among these while in the action state. State charts must reflect this fact.

This is a must have rule and we use the 'Medium-Priority' communication structure for it.

Corollary 17 States must not be associated with any actions. Sub-states corresponding to different models should be used instead.

Currently, ArgoUML allows users to associate actions with states. We use an alternative way to prohibit users from doing so. If any state action is created, the program shows a violation warning. This is expressed in the following:

```
Get states;
Get entryAction;
Get exitAction;
If entryAction==null && exitAction==null
    Exit;
Show error;
```

The 'High-Priority' communication structure is used here.

Corollary 19 If the partitions of an activity diagram represent different objects, they must be part of a composite, which is shown in the class diagram.

ArgoUML currently does not support swimlanes. Our program cannot know what objects the states describe. Thus, our program can only show a reminder for this corollary for all states in activity diagrams. We use the 'Low-Priority' communication structure here. This is expressed in the following:

```
Get actionState;
Show reminder;
Add to reminder list;
Exit;
```

Rule 25 The quantitative object behaviour (for each model) is entirely describable by top-level state chart (SC0)

This is a must have rule and we adopt the 'Medium-Priority' communication structure for it.

Rule 26 All UML-transitions in SC0 must correspond to an operation of the object, which

SC0 is associated with.

This is a must have rule and the 'Medium-Priority' communication structure is adopted.

Corollary 20 Every object must have at least one operation.

This is a must have corollary and we use the 'Medium-Priority' communication structure for it.

Corollary 21 States in SC0 are stable.

The ideal solution is to add this corollary in the UML specification. Currently, we use an alternative solution in ArgoUML for this corollary. To define a state as stable or unstable, our program first gets all transitions with this state as the source state in SC0. Then each of these transitions is checked to see whether it has a trigger event. If any transition has no trigger, it can happen spontaneously. Thus, this state is unstable and the program shows a violation warning. This is expressed in the following:

```
Get state;
Get container;
If container==top
    Get outTransitions;
    While outTransitions.hasNext()
        Get trigger;
        If trigger==null
            Show error;
Exit;
```

Since the implementation shows violation warning unless all out-transitions have trigger, this is a must have corollary. The 'Medium-Priority' communication structure is used.

Rule 27 An object must exhibit additional operations expressing qualitative changes, if a super- or sub-class is defined and instances can undergo changes of class to the super- or sub-class.

This is a must have rule. We use the 'Medium-Priority' communication structure for it.

Rule 28 Methods may be described by state charts other than top-level state charts.

The ArgoUML currently has the problem of modeling methods. Users cannot attach a note to operations. We alternatively use the stereotype of <<method>> to model methods of operations. That is, if the stereotype of an operation is <<method>>, this operation is actually a method. This method implements the operation with the same name. However, operations cannot associate with state charts, which are already associated with classes in ArgoUML. In other words, a class and its operation cannot associate with a same state chart. That is, using our alternative way of modeling methods, it is impossible to describe methods using SC0. This rule is always true in ArgoUML. Thus, there is no need to implement it.

Corollary 23 A state chart describing a method must begin and end with those states in SC0 which the operation that the method implements is a realization of.

Since we use <<method>> stereotypes to identify methods, the program needs to check an operation stereotype before getting a method. This is expressed in the following:

```
Get class;
Get sc0;
Get sc0Transitions;
While sc0Transitions.hasNext()
    Get transitionName;
    Get sourceState;
    Get targetState;
    Get operations;
    While operations.hasNext()
        Get stereotype;
        If stereotype==method
            Get operationName;
            If operationName==transitionName
                Get operationStateChart;
                Get initialState;
                Get finalState;
                If !(initialState==sourceState)||!(finalState==targetName)
                    Show error;
Exit;
```

This is a must be corollary and the 'High-Priority' communication structure is adopted.

Corollary 24 State transitions out of the initial state of a method realizing an operation must be associated with the same event that is associated with the transition in SC0, which

represents that operation.

Based on the same reasoning as in Corollary 23, the program needs to check an operation stereotype before getting a method. This is expressed in the following:

```
Get class;
Get sc0;
Get sc0Transitions;
While sc0Transitions.hasNext()
    Get transitionName;
    Get trigger;
    Get event1;
    Get eventName1;
    Get operations;
    While operations.hasNext()
        Get stereotype;
        If stereotype==method
            Get operations;
            Get operationName;
        If operationName==transitionName
            Get operationStateChart;
            Get initialState;
            Get outTransition;
            Get trigger;
            Get event2;
            Get eventName2;
            If !(eventname1==eventName2)
                Show error;
Exit;
```

This is a must be corollary and we use the 'High-Priority' communication structure here.

Corollary 25 A state chart either expresses the external behaviour of an object (SC0), a method, a signal reception or is a composite state contained in another state machine.

Rule 29 gives the solution when state charts specify operations. Thus, in the implementation of this corollary in ArgoUML, if a model element is an operation, the program does not show a violation warning. We leave this to be implemented in Rule 29. In ArgoUML, state machines as composite states contained in another state machine are different from ordinary state machines. That is, when we get state machines, those state machines as composite states would not be retrieved. We can simplify our implementation by ignoring these state machines. In addition, ArgoUML currently has the problem of attaching state machines to receptions. That is, we only need to check classes, operations and methods. The program gets ordinary state charts and elements owning them. If these elements are not classes or methods (operations), it shows a violation warning. This is expressed in the following:

```
Get stateChart;
Get owner;
If !(owner==object)||!(owner==method)|| !(owner==operation)
        Show error;
Exit;
```

This is a must be corollary and the 'High-Priority' communication structure is adopted.

Rule 29 An operation is not directly specified by state machines. Instead, the methods that implement an operation are specified by state machines.

In the implementation of Corollary 25 in ArgoUML, we did not check operations and

left that job to be done by this rule. The program gets classes and their operations. Then it checks behavioural features of every operation. If any behaviour owned by an operation is a state machine, the program shows a violation warning as well as the solution. This is expressed in the following:

```
Get class;
Get operations;
While operations.hasContainer
    Get behaviours;
    While behaviours.hasNext()
        If behaviour==stateMachine
            Show error;
Exit;
```

This is a must be rule and the 'High-Priority' communication structure is adopted.

Corollary 26 A state machine that specifies the behaviour of a class or a method is not contained in other state machines.

UML does not specify that a CASE tool has to supply functions of attaching submachines to the behaviour of a class or a method. Thus, ArgoUML does not support modeling a state machine as a container for state machines of classes, operations and methods. That is, modellers using ArgoUML cannot model a class, operation or method state machine contained by another state machine. This corollary is always true in ArgoUML and there is no need to implement it.

Rule-New 03 Receptions should not be modeled.

This is a must be rule and the 'High-Priority' communication structure is used here.

Rule 30 An operation must be associated with the declaration of signal reception.

This is a must have rule and the 'Medium-Priority' communication structure is used for it.

Rule 31 The event associated with an operation must be identical to the event associated with the signal associated with the reception.

In Corollary 25, we have discussed that statemachine for receptions does not work in ArgoUML. Thus, the program is not able to compare events associated with operations and receptions. When there is an event associated with an operation, it shows a reminder to users and the 'Low-Priority' communication structure is used. This is expressed in the following:

Get class;
Get operation;
Get SC0;
Get transition;
Get event;
Get signal;
Get reception;
Show reminder;
Put into reminderList;
Exit;

Corollary 28 The state machines associated with a reception and with a method specifying

the implementation of an operation which is in turn associated with that reception, must possess the same initial and final states.

The ArgoUML currently throws exceptions when attaching statemachines to receptions. Thus, there is no way to get trigger events of receptions. We can only get the initial and final states of the method. When the program finds the corresponding reception, it reminds users to follow this corollary. The 'Low-Priority' communication structure is adopted here. This is expressed in the following:

Get class;
Get operation;
Get method;
Get stateMachine;
Get initialState;
Get finalState;
Get reception;
Show reminder;
Put to reminder;
Exit;

Rule 32 Acquisition (loss) of independent properties leads to expansion (contraction) of the thing's top-level state space SC0 by an orthogonal region.

This is a must be rule and we use the 'High-Priority' communication structure here.

Rule-New 02 Acquisition (loss) of non-independent properties should be modeled as a sub-machine of the related thing's state modeled by top-level state SC0.

This is a must be rule and the 'High-Priority' communication structure is adopted.

Rule-New 04 Neither stimuli, nor messages should be modeled.

Since stimuli cannot be modeled in ArgoUML currently, we only check messages. The 'Medium-Priority' communication structure for must have rules is used here. This is expressed in the following:

```
Get interaction;
Get message;
If !(message==null)
Get collaboration;
    Show error;
Exit;
```

Currently, the biggest deficiency in ArgoUML is that the sequence diagrams do not work at all: "Sequence diagrams missing completely from ArgoUML0.14" (Tigris, 2003). In addition, in UML "collaboration diagrams show the full context of an interaction" (OMG, 2003, p3-122). Thus, for Rules 33-36 and Corollaries 30-39, we only check collaboration diagrams.

Another thing we have to mention is the retrieving of collaboration diagrams does not work properly. When a diagram file is opened, collaboration diagrams in it will not be shown. However, all information, such as interactions and senders are correctly saved. We can go to the text field to read them. This deficiency is merely inconvenience when we see the demo or test our program.

Rule 33 For every class of objects between which message passing is declared, there exists an association class or the two classes are parts of the same aggregate.

This is a must have rule and we use the 'Medium-Priority' communication structure here.

Rule 34 Every object must be the receiver and sender of some message.

This is a must have rule and we the 'Medium-Priority' communication structure is used.

Rule 35 A constraint relates attributes of a single class or attributes of association classes the class participates in.

Constraints in ArgoUML do not follow the notation in UML and do not work properly. At least, one needs to have a field to input constrains. However, ArgoUML uses an very different way doing that and it's not working Thus, when there is an attribute, our implementation gets the owner class and association classes. Then, it shows this rule to remind modelers following this rule. If the user does not want to keep this reminder, it can be dismissed and will not appear in the reminder list again. The 'Low-Priority' communication structure is adopted here. This is expressed in the following:

Get attribute;
Get class;
Get associationClasses;

Show reminder;
Put to reminderList;
Exit;

Rule 36 For every attribute there exists a constraint which relates this attribute to some other attribute.

When there is an attribute, the program only shows this rule to remind modelers to follow it. If the user does not want to keep this reminder, it can be dismissed and will not appear in the reminder list again. Because ArgoUML does not follow the notation of constraints in UML, the 'Low-Priority' communication structure is used here. This is expressed in the following:

Get attribute;
Show reminder;
Put to reminderList;
Exit;

Corollary 31 A UML-state transition associated with an action must modify an association class attribute's value.

Ideally, this corollary can be implemented by adding a mechanism to UML. CASE tools will also enable the mechanism based on UML specification. In the current stage, we implement an alternative way to enforce this corollary in ArgoUML. The program first gets state transitions. Then it checks whether there are actions for a transition. If an action is not null, it checks what model element the state machine represents. Because of

Corollary 25, the program only reminds user to follow this corollary when the model element is a class or method. However, transitions of methods cannot be retrieved using the above method. We have to get a method and its state machine, then get transitions and actions. When such an action is found, the program shows a reminder to users. If the user does not want to keep this reminder, it can be dismissed and will not appear in the reminder list again. The 'Low-Priority' communication structure is adopted here. This is expressed in the following:

```
Get transition;
Get action;
If !(action==null)
    Get stateMachine;
    Get owner;
    If owner==class
        Show reminder;
        Put to reminderList;
Get method;
Get stateMachine;
Get transition;
Get action;
If !(action==null)
    Show reminder;
    Put to reminderList;
Exit;
```

Corollary 32 For every interaction between UML-objects, there must exist a corresponding UML-state transition in both interacting UML-objects.

This corollary can be implemented by adding a mechanism to UML. CASE tools will also enable the mechanism based on UML specification. Based on the same reasoning as

in Corollary 31, we implement an alternative way to enforce this corollary in ArgoUML. The program gets an interaction and its participating objects. Then it goes to class diagrams looking for these two classes. Next, it gets the top-level state chart (SC0) of both classes and reminds users to follow this corollary. If the user does not want to keep this reminder, it can be dismissed and will not appear in reminder list again. The 'Low-Priority' communication structure is adopted here. This is expressed in the following:

Get collaboration;
Get interactions;
Get sender;
Get senderBaseClass;
Get sbcStateMachine;
Get receiverBaseClass;
Get rbcStateMachine;
Show reminder;
Put to reminderList;
Exit;

Corollary 33 A state transition associated with an event must modify an association class attribute's value.

The ideal solution for this corollary is also to add a mechanism to UML specification. In current stage, we implement an alternative way to enforce it in ArgoUML. The program first gets state transitions. Then it checks whether there is trigger events for a transition. If the event is not null, it checks what model element the state machine represents. Because of Corollary 25, the program only reminds users to follow this corollary when the model

element is a class or method. However, transitions of methods cannot be retrieved in the above manner. We have to get a method and its state machine, then get transitions and events. When such an event is found, the program shows reminder to users. If the user does not want to keep this reminder, it can be dismissed and will not appear in the reminder list again. The 'Low-Priority' communication structure is adopted here. This is expressed in the following:

```
Get transition;
Get event;
If !(event==null)
    Get stateMachine;
    Get owner;
    If owner==class
        Show reminder;
        Put to reminderList;
Get method;
Get stateMachine;
Get transition;
Get event;
If !(event==null)
    Show reminder;
    Put to reminderList;
Exit;
```

Corollary 34 A signal event may only be associated with a transition in a top-level state chart and the initial transition of a method implementing this.

In Corollary 34, we discussed that some transitions in state machines other than class state machines, specifically in ArgoUML: method state machines, could not be retrieved by the ideal solution. Thus, for these transitions, the program gets a method and its state

machine, then gets transitions and events. If the event is a signal event, it shows a violation warning. The 'High-Priority' communication structure is adopted here. This is expressed in the following:

```
Get transition;
Get event;
If event==signalEvent
    Get stateMachine;
    Get owner;
    If owner==class
        Get sourceState;
        Get targetState;
        If !(sourceState.isTop)||!(targetState.isTop)
            Show error;
Get method;
Get stateMachine;
Get transition;
Get event;
If event==signalEvent
    Show error;
Exit;
```

Corollary 35 A call event may only be associated with a transition in a top-level state chart or the initial transition of a method implementing this.

Based on the same reasoning as in Corollary 34, for those corresponding transitions, the program gets a method and its state machine, then it gets transitions and events. If the event is a call event, it shows a violation warning. The 'High-Priority' communication structure is adopted here. This is expressed in the following:

```
Get transition;
Get event;
```

```
If event==callEvent
    Get stateMachine;
    Get owner;
    If owner==class
        Get sourceState;
        Get targetState;
        If !(sourceState.isTop)||!(targetState.isTop)
            Show error;
Get method;
Get stateMachine;
Get transition;
Get event;
If event==callEvent
    Show error;
Exit;
```

Corollary 38 The final state transitions of any method implementing an operation that may be invoked through a call action must cause a return action.

Currently, the action expression cannot be retrieved in ArgoUML. We specify it through an action name. That is, an action name field is filled by an action expression. The 'Medium-Priority' communication structure for must have rules is used here.

Corollary 39 For the state machine of a method to contain a state transition whose effect is a return action, there must exist a corresponding state transition in a state machine of some other object whose effect is a corresponding call action.

Since the action expression does not work in ArgoUML, we cannot retrieve the acting method. Thus, when the program finds a final transition with a return action representing a

method, it shows a reminder. The 'Low-Priority' communication structure is used here.

This is expressed in the following:

```
Get class;
Get method;
Get stateMachine;
Get transition;
Get target;
If target==finalState
    Get action;
    If action==returnAction
        Remind user;
        Put into reminderList;
Exit;
```

Chapter 8 Conclusion

UML has become very popular in software engineering and is used for at least two purposes: OO software design and conceptual modeling of a domain for which a system is to be required (communication and documentation). However, UML's origins in software engineering may limit its appropriateness for conceptual modeling. Thus, some research has attempted to develop rules that make UML better suited for domain modeling. Notably, Evermann and Wand's research specifically interests us. They point out that conceptual modeling involves representing aspects of the real world and that ontology is a branch of philosophy dealing with the nature and structure of the real world, making it appropriate as a basis for developing rules about what a conceptual modeling language should do. From these assumptions, they developed a set of ontological rules that place constraints on the construction of UML diagrams to ensure that they properly represent underlying ontological assumptions. On the other hand, there are no existing UML-based CASE tools that enforce such rules. This research has proposed implementation approaches for such functionality in UML-based CASE tools. In addition, we have implemented them in a specific tool, namely ArgoUML. The main contribution of this research is having developed software based on Evermann and Wand's ontological rules to

guide the modeling process. We have demonstrated the feasibility of implementing these rules. Specifically, the system has built-in 'intelligence' to help people using UML to do conceptual modeling. The software has the ability to realize two functions: first, to examine UML diagrams and detect violations, and second, to explain the nature of violations. There are in total 36 rules and 39 corollaries and the following result is obtained.

We have given the general implementation approach to non-specific CASE tools. Approaches of 35 rules and 26 corollaries are given. Within them, 3 rules and 11 corollaries are realized by enabling, disabling or modifying mechanisms (features) in UML specification; 32 rules and 15 corollaries are realized by the critiquing expert systems. We have given all algorithms to them. We did not give approaches for 1 rule and 13 corollaries. This is because 1 rule and 8 corollaries are covered by others; 2 corollaries are actually solution of others; 2 corollaries are dropped by our research; 1 corollary is always true and thus redundant.

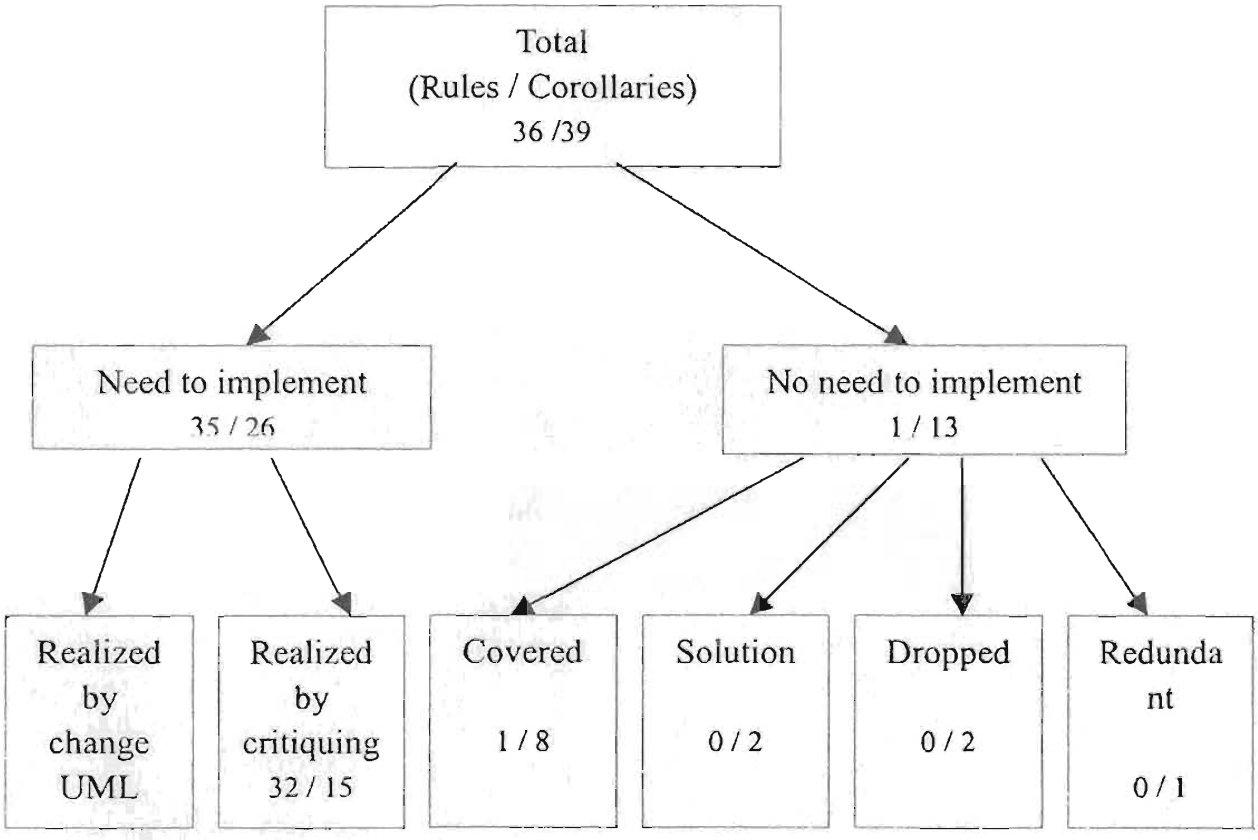

Figure 8.1 Implementation result for general CASE tools

Of the rules and corollaries that can be feasibly implemented, our analysis shows that these can be divided into four categories, depending on their importance and the ease with which they can be detected. The categories are listed in Table 8.1:

No.	Rule Type	Communication Structure	Explain
1	Must Be; Simple to correct	CRITICAL	The user is prevented from proceeding in the construction of a diagram until the violation is corrected.
2	Must Be; Difficult to correct	HIGH-Priority	The user is warned once of the violation, and the violation is added to a reminder list with high priority.

3	Must Have	MEDIUM-Priority	There is no popup indicating the violation. Instead, the potential violation is added to a reminder list with medium priority.
4	Relates to a non-existing UML component	LOW-Priority	There is popup indicating the reminder. Then a low priority reminder is added to the reminder list. The reminder can be discarded by users.

Table 8.1 Categories of feasible (to implement) rules

Of the rules and corollaries that are no need to be implemented, our analysis shows that these can be also divided into four categories. The categories are listed in Table 8.2:

No.	Rule Types	Explain
1	Covered	These rules are covered or logically implied when others are followed.
2	Solution	These rules propose a kind of solution to violations of others.
3	Dropped	These rules appear to be inconsistent with Bunge's ontology or UML.
4	Redundant	These rules are impossible to violate.

Table 8.2 Categories of no need (to implement) rules

This research also includes a Proof-of-Concept implementation. We have implemented Evermann and Wand's ontological rules in the specific UML-base CASE tool: ArgoUML. 32 rules and 20 corollaries are implemented. Within them, 27 rules and 12 corollaries are fully implemented (Critiques for these rules and corollaries can accurately detect violations and represent to users. When the user solves the problem and the diagram no longer violates the rule / corollary, the system will automatically dismiss the violation warning.); 5 rules and 8 corollaries are implemented by giving reminders. We did not

implement 4 rules and 19 corollaries. This is because 1 rule and 4 corollaries relate to newly proposed mechanism, which is not supported by current UML; 10 corollaries are covered by others; 2 corollaries are actually solution of others; 2 corollaries are dropped by our research; 1 rule and 1 corollary are always true and thus redundant; the required functions for 2 rules do not work in ArgoUML.

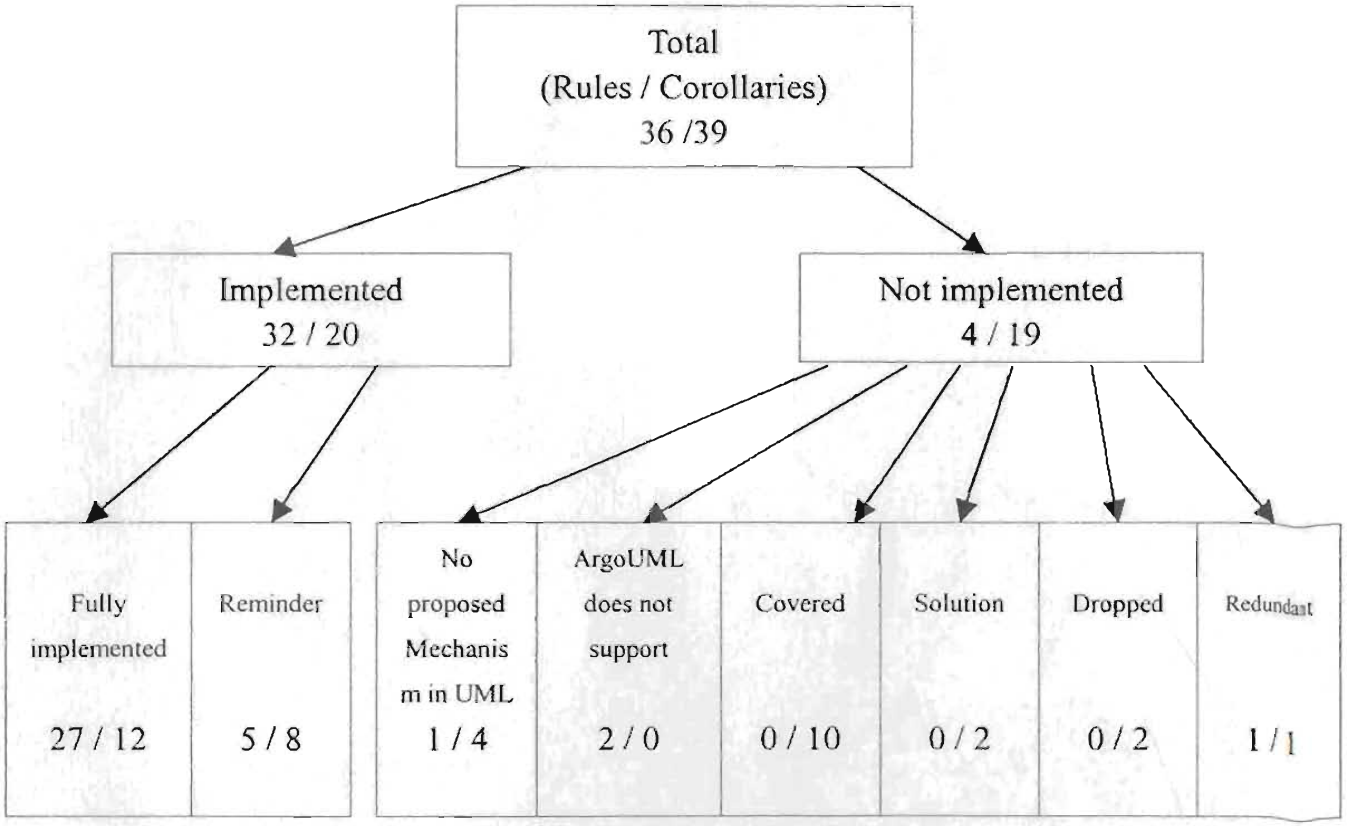

Figure 8.2 Implementation result for ArgoUML

To support the analysis and implementation of the ontological rules, we have also used several other approaches: inquiry for required information, use of a dictionary, and alternatives to proposed changes of UML.

As additional contribution of this research, we explained and evaluated Evermann and

Wand's ontological rules. As a result, we dropped 2 corollaries and proposed 4 new rules and also implemented them.

References:

1. Bodart, Francois and Ron, Weber. (1996). Optional Properties Versus Subtyping in Conceptual Modeling: A Theory and Empirical Test. Proceedings of the International Conference on Information Systems. P. 450.
2. Booch, Grady. (1994). Object Oriented Analysis and Design with Applications (2nd Edition). Redwood City, CA: Benjamin/Cummings.
3. Burton-Jones, A. and Meso, P. (2002). How Good Are These UML Diagrams? An Empirical Test of The Wand and Weber Good Decomposion Model. 2002-Twenty-Third International Conference on Information Systems.
4. Bunge, Mario Augusto. (1977). Ontology : The Furniture of the World: Volume 3 Treatise On Basic Philosophy. Dordrecht, Holland: D. Reidel Publishing Company.
5. Bunge, Mario Augusto. (1979). Ontology : A World of Systems: Volume 4 Treatise On Basic Philosophy. Dordrecht, Holland: D. Reidel Publishing Company.
6. CollabNet, Inc. 2003. ArgoUML User Manual. Retrieve September 2003 from http://argouml.tigris.org/documentation/defaulthtml/manual/.
7. CollabNet, Inc. 2003. ArgoUML Known Incompatiable. Retrieve September 2003 from http://argouml.tigris.org/documentation/umlsupport.

8. Evermann, Joerg and Yair, Wand. (2001). An Ontological Examination of Object Interaction. Proceedings of the Workshop on Information Technologies WITS 2001, New Orleans, LA.

9. Evermann, Joerg and Yair Wand. (2001a). Towards Ontologically based Semantics for UML Constructs. Proceedings of the 20th International Conference on Conceptual Modeling ER'2001, Yokohama, Japan. Berlin: Springer Verlag.

10. Evermann, Joerg. (2003). Using Design Languages For Conceptual Modeling: The UML CASE. Doctor thesis draft.

11. Gemino, A. (1999). Empirical Comparisons of Systems Analysis Modeling Techniques. Ph.D. thesis, University of British Columbia, Canada.

12. Guizzardi Giancarlo, Heinrich Herre and Gerd Wagner, (2002[1]). Towards Ontological Foundations for UML Conceptual Models.

13. Guizzardi Giancarlo, Heinrich Herre and Gerd Wagner, (2002[2]). On the general Ontological Foundation of Conceptual Modeling.

14. Green, Peter and Rosemann Michael. (2002). Developing a meta model for the Bunge-Wand-Weber ontological constructs. Information Systems, (27), P. 75-91.

15. Jacobson, I. Christerson, M. Jonsson, P. Overgaard, G. (1992). Object-oriented Software Engineering—A Use Case Driven Approach. Addison-Wesley, Reading/MA.

16. Jason E. Robbins, (1998). Design Critiquing Systems. Tech Report UCI-98-41.

17. Klemke E.D. (Edited by). (1968). Essays on Frege. Urbana: University of Illinois.

18. Martinus Nijhoff. (1977). On the content and object of presentations. A psychological investigation. Translated and with an introduction by Reinhardt Grossmann.

19. Object Management Group (OMG). (2003). OMG Unified Modeling Language Specification. Version 1.5. Retrieve at June, 2003 from http://www.uml.org.

20. Opdahl, A.L. and B, Henderson-Sellers. (1999). Evaluating and Improving OO Modeling Languages Using the BWW-Model. Proceedings of the Information Systems Foundation Workshop. Ontology, Semiotics and Practice.

21. Opdahl, A.L. and B. Henderson-Sellers and F. Barbier. (1999). An Ontological Evaluation of the OML Metamodel. Information System Concepts: An Integrated Discipline. Dordrecht, Holland: Kluwer.

22. Opdahl, A. L. and B. Henderson-Sellers. (2001). Grounding the OML Metamodel in Ontology. The Journal of Systems and Software, (57), P.119-143.

23. Opdahl, A. L. and B. Henderson-Sellers. (2002). Ontological Evaluation of the UML Using the Bunge–Wand–Weber Model. Software System Model (2002) 1: P.43–67 / Digital Object Identifier (DOI) 10.1007/s10270-002-0003-9.

24. Paul, Kegan. and Routledge. (1874). Psychology from an Empirical Standpoint.

25. Rumbaugh, J. Blaha, M. Premerlani, W. Eddy, F. Lorensen, W. (1991). Object-oriented Modeling and Design. Prentice Hall Englewood Cliffs / NJ.

26. Wand, Yair. (1989). A Proposal for a Formal Model of Objects. Object-Oriented

Concepts, Languages, Applications and Databases. Boston, MA.: Addison-Wesley, P. 537-559.

27. Wand, Yair and Weber, Ron. (1989). An Ontological Evaluation of Systems Analysis and Design Methods. Information System Concepts: An In-Depth Analysis. North-Holland: Elsevier Science Publishers B.V.

28. Wand, Yair and Weber, Ron. (1990). Mario Bunge's Ontology as a Formal Foundation for Information Systems Concepts. Studies on Mario Bunge's Treatise. Rodopi, Atlanta.

29. Wand, Yair and Weber, Ron. (1991). A unified model of software and data decomposition. Proceedings of the Twelfth International Conference on Information Systems, P. 101-110.

30. Wand, Yair and Weber, Ron. (1993). On the Ontological Expressiveness of Information Systems Analysis and Design Grammars, Journal of Information Systems. P.217-237.

31. Wand, Yair and Weber, Ron. (1995). Towards a theory of Deep Structure of Information Systems. Journal of Information Systems, (5), P. 203-223.

32. Wand, Yair and Veda Storey and Weber, Ron. (1999). An Ontological Analysis of the Relationship Construct in Conceptual Modeling. ACM Transactions on Database Systems, (24) 4, P. 494-528.

33. Weber, R. and Zhang, Y. (1996). An Analytical Evaluation of NIAM's Grammar for Conceptual Schema Diagrams. Information Systems Journal, 6(2), P. 147-170.

Appendix A

Implementation Manual

In order to run the ArgoUML including our implemenation of Evermann and Wand's rules, you have to follow two steps. Also, you can use the 'demo' step to experience and test the implementation. The configuration is for Microsoft Windows XP and Windows2000 operation systems. Because our programs (Rule 1 and Corollary 1) need access to several data files, which must be able to be found in the correct path, those programs have been configed for paths on Windows. However, we still cover Unix in case users have no access to WinXP or Win2000. Note that Rule 1 and Corollary 1 will not work properly on Unix.

1. Setup

1.1 Copy files to your computer

Please copy the 'LS' directory including all files and folders from our compact disk to the root directory to your hard drive: "C:\".

1.2 Set JAVA_HOME for ArgoUML running environment

By setting JAVA_HOME to different values you can at different times compile and

run ArgoUML with different versions of JDK and java.

In windows, set JAVA_HOME=\where\you\have\installed\jdk.

In Unix, set JAVA_HOME=/where/you/have/installed/jdk, export JAVA_HOME. This is for sh-style shells like sh, ksh, zsh and bash. If you use csh-style shells like csh and tcsh you will instead have to write setenv JAVA_HOME /where/you/have/installed/jdk.

2. Compile and run

For convinence, there are two scripts (one for Windows and one for Unix) that are called build.bat and build.sh respectively.

2.1 Compiling and running for Windows

1. Change the current directory to the directory of 'src_new':

chdir C:\ls\working\src_new

2.Clear the already existing class files using:

build clean

3.Compile and run ArgoUML using:

build run

If you do this from Cygwin you work just like for Unix.

2.2 Compiling and running for Unix

1. Change the current directory to the directory of 'src_new':

cd /your/copy/of/working/src_new

2. Clear the already existing class files using:

./build.sh clean

3. Compile and run ArgoUML using

./build.sh run

Since there are 1322 files to compile, it may take a little while. For example, a machine with a CPU of Celeron-D 2.8G spends 1 minute 31 seconds on compling and starting this version of ArgoUML.

Please refer to ArgoUML User Manual (Tigris, 2004) for operation guide.

3. Demo

You can use this step to experience and test our implementation. You just simply click on the "Open Project" in the "File" menu. Then choose the corresponding '.zargo' files. For example, 'R33-W.zargo' is the example of the 'Wrong' case of Rule 33 and 'R33-R.zargo' is the example of the 'Right' case of Rule 33. Note that, to see the demo, the corresponding rule / corollary critic must be activated in the "Browse Critics" of the "Critique" menu. For a non-confusing demo for each rule / corollary, we suggest users to

ONLY activate the corresponding critic for that rule / corollary.

Appendix B

Implementation Source Code and Demo Data

We have included all source code of the implementation in a compact disk. There is a '*.java' file for each implemented rule or corollary. The file names correspond to rule / corollary numbers. For example, 'CrR01.java' is the source code for implementation of Rule 1; 'CrC01.java' is the source code for implementation of Corollary 1. All the source code files are located in the following folder: "CD-Drive:\ls\working\src_new\org\argouml\uml\cognitive\critics". The CD also includes demo data for our implementation. There is one or two '*.zargo' file(s) for each implemented rule or corollary. The file names also correspond to rule / corollary numbers. For example, 'R33-W.zargo' is the example of the wrong case of Rule 33, 'R33-R.zargo' is the example of the right case of Rule 33. For the simple to correct violation rules / corollaries, we only give one demo file of the wrong case (some demo files include both right and wrong cases) for each of them. For example, 'R01.zargo' is both the right and wrong case of Rule 1.

Please refer the attached compact disk for those files.

Appendix C

List of Evermann's Ontological Rules and Corollaries

Rule 1 Only substantial entities in the world are modelled as objects.

Rule 2 Ontological properties of things must be modeled as UML-attributes.

Corollary 1 Attributes in a UML-description of the real world cannot refer to substantial entities.

Rule 3 Sets of mutual properties must be represented as attributes of association classes.

Corollary 2 An association class cannot represent substantial entities or composites of substantial entities.

Corollary 3 If an association class of an n-ary association is intended to represent substantial things, the association should instead be modelled as one with arity (n+1).

Corollary 4 An association class representing a composite must instead be modelled as a composite with attributes representing emergent intrinsic properties.

Corollary 5 An association class cannot possess methods or operations.

Corollary 6 An association class cannot be associated with a state machine.

Corollary 7 An association class must possess at least one attribute.

Corollary 8 An association class must not be associated with another class.

Corollary 9 An association class must not participate in generalization relationships.

Rule 4 If mutual properties can change quantitatively, methods and operations that change the values of attributes of the association class must be modelled for one or more of the classes participating in the association, objects of which can effect the change, not for the associations class.

Rule 5 An association class represents a set of mutual properties arising out of the same interaction.

Rule 6 A composition relation must not be modelled.

Rule 7 Every UML-aggregate must possess at least one attribute which is not an attribute of its parts or participate in an association.

Rule 8 All UML-classes must possess at least one attribute or participate in an association.

Rule 9 Object ID's must not be modelled as attributes.

Rule 10 The set of attribute values (representing mutual and intrinsic properties) must uniquely identify an object.

Rule 11 Every attribute has a value.

Corollary 10 Attribute multiplicities greater than one imply that the order of the different individual attribute value components is semantically irrelevant.

Rule 12 Classes of objects that exhibit additional behaviour, additional attributes or additional association classes with respect to other objects of the same class, must be modelled as specialized sub-classes.

Corollary 11 An object acquiring additional behaviour or properties must be destroyed as

instance of the general class and created as instance of the specialized class that is modelled with the relevant operations or association classes.

Corollary 12 Re-classification occurs only within a generalization / specialization hierarchy.

Rule 13 Every UML-aggregate object must consist of at least two parts.

Rule 14 An instance of a class that by virtue of additional aggregation relationships acquires emergent properties or emergent behaviour must be modeled as an instance of a specialized class which declares the corresponding attributes and operations.

Rule 15 Object creation occurs when an entity acquires a property so that it becomes a member of a different class.

Corollary 13 Object destruction occurs when an entity loses a property that is necessary for membership in a particular class.

Rule 16 Attributes with class scope should instead be modelled as attributes of an aggregate representing the objects of the class.

Rule 17 If a class that is specialized is declared as abstract, the specialization must be declared to be 'complete'.

Rule 18 A class that is not specialized cannot be declared abstract.

Rule 19 A specialized class must define more attributes, more operations or participate in more associations than the general class.

Rule 20 Every ordinary association must be an association class.

Rule 21 A UML-state represents a specific assignment of values to the attributes and attribute of association classes of the objects for which the state is defined.

Corollary 14 A UML-transition must change the value of at least one attribute used to define the state space.

Rule 22 For every level of refinement of a state C, there must be an additional set of attributes in the class description or in participating association classes that change as the object transitions among the sub-states.

Corollary 15 For all immediate substates of a super-state, the values assigned to attributes describing the super-state are invariant and are equal to those defining the super-state.

Corollary 16 Concurrent sub-states require mutually disjunct sets of additional attributes in the class description or in participating association classes.

Rule 23 Guard conditions on transitions from the same state to nonconcurrent sub-states must be mutually disjunct.

Rule 24 Action states are super-states of a set of sub-states. The object transitions among these while in the action state. State charts must reflect this fact.

Corollary 17 States must not be associated with any actions. Sub-states corresponding to different models should be used instead.

Corollary 18 All states in an activity diagram must be states of the same object.

Corollary 19 If the partitions of an activity diagram represent different objects, they must be part of a composite which is shown in the class diagram.

Rule 25 The quantitative object behaviour (for each model) is entirely describable by top-level state chart (SC0)

Rule 26 All UML-transitions in SC0 must correspond to an operation of the object which SC0 is associated with.

Corollary 20 Every object must have at least one operation.

Corollary 21 States in SC0 are stable.

Corollary 22 All UML-transitions in SC0 must be associated with a UML event.

Rule 27 An object must exhibit additional operations expressing qualitative changes, if a super- or sub-class is defined and instances can undergo changes of class to the super- or sub-class.

Rule 28 Methods may be described by state charts other than top-level state charts.

Corollary 23 A state chart describing a method must begin and end with those states in SC0 which the operation that the method implements is a realization of.

Corollary 24 State transitions out of the first state of a method realizing an operation must be associated with the same event that is associated with the transition in SC0 which represents that operation.

Corollary 25 A state chart either expresses the external behaviour of an object (SC0), a method, a signal reception or is a composite state contained in another state machine.

Rule 29 An operation is not directly specified by state machines. Instead, the methods that implement an operation are specified by state machines.

Corollary 26 A state machine that specifies the behaviour of a class or a method is not contained in other state machines.

Corollary 27 The method corresponding to a state chart must modify the attribute values of the object corresponding to the values defined for the initial and final state of the method.

Rule 30 An operation must be associated with the declaration of signal reception.

Rule 31 The event associated with an operation must be identical to the event associated with the signal associated with the reception.

Corollary 28 The state machines associated with a reception and with a method specifying the implementation of an operation which is in turn associated with that reception, must possess the same initial and final states.

Rule 32 Acquisition (loss) of independent properties leads to expansion (contraction) of the things top-level state space SC0 by an orthogonal region.

Corollary 29 Every object must be capable of at least one state transition or be able to undergo change of class to a super- or sub-class.

Rule 33 For every class of objects between which message passing is declared, there exists an association class.

Rule 34 Every object must be the receiver and sender of some message.

Rule 35 For every attribute there exists a constraint which relates this attribute to some other attribute.

Corollary 30 An association class cannot be sender or receiver of a message.

Rule 36 A constraint relates attributes of a single class or attributes of association classes the class participates in.

Corollary 31 A UML-state transition associated with an action must modify an association class attribute's value.

Corollary 32 For every interaction between UML-objects, there must exist a corresponding UML-state transition in both interacting UML-objects.

Corollary 33 A state transition associated with an event must modify an association class attribute's value.

Corollary 34 A signal event may only be associated with a transition in a top-level state chart and the initial transition of a method implementing this.

Corollary 35 A call event may only be associated with a transition in a top-level state chart or the initial transition of a method implementing this.

Corollary 36 Synchronous communication of objects implies transition to a state which cannot be left except through a state transition associated with the return signal.

Corollary 37 Asynchronous communication of objects with expected response implies the existence of at least one state transition caused by the object acted upon, signifying the return interaction after the state transition signifying the original communication.

Corollary 38 The final state transitions of any method implementing an operation that may be invoked through a call action must cause a return action.

Corollary 39 For the state machine of a method to contain a state transition whose effect is

a return action, there must exist a corresponding state transition in a state machine of some other object whose effect is a corresponding call action.