VISUAL SPECIFICATION OF MODEL INTERPRETERS

By

Jianfeng Wang

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May 2000

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Gabor Karsai for his support and guidance in this research over the past years. I would also like to thank Dr. Michael Moore for all of his helps and supports. Other members of the Institute for Software Integrated System have also been supportive: Dr. Janos Sztipanovits, Dr. Greg Nordstrom, and Dr. Akos Ledeczi.

Thanks to my family (my Mom, Dad, Jianqing and Honghong), they has given me a great deal of support throughout all the last years. Without their support this would not have been possible. Thanks most of all to my wife, Xiaoting. She has always provided much encouragement and supports to help me go through these years.

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Complex computer-based systems (CBSs) are characterized by their tight integration of information processing and the physical environment of the systems. It is the correct and efficient representation and interaction of the software portion with its physical context that guarantee the correct functionality and efficient performance of the CBSs. Most software and hardware systems, or CBSs, are experiencing rapid and continuous evolution in their full lifecycle. Thus, it is extremely important and necessary for us to have a way to construct software and manage its associations with its physical context so software systems can be evolved and maintained easily when their physical context changes.

An emerging technology that accomplishes this and has been accepted and proven to be an efficient and effective method is Model Integrated Computing (MIC) [15]. MIC has introduced model-based computing techniques into software engineering. Model-based computing can facilitate management of complex software systems and enable easy system modification and generation. Graphical models are used to specify the system. Model-based analysis can be performed while the system is still in the design stages. By using model-based computing techniques, designers can create domain-specific models to present the software, its physical context, and their relationships. Then specific tools are used to analyze the models and to generate the application program based on the models. Once the environment or physical context of the software changes, the models can be recreated or modified in the DSME, and application programs can be

regenerated. This facilitates the evolution of the application, supports system maintenance, and therefore reduces the costs during the entire lifecycle of the system.

One approach to MIC is model-integrated program synthesis (MIPS). In MIPS, models are created that capture various aspects of a domain-specific system. Model interpreters are used to translate these models for use in the system's execution environment. When changes in the overall system require new application programs, the models are updated to reflect these changes, and the applications are regenerated automatically from the models [1].

In MIC, it is the model interpreter that translates the domain-specific models into application program. Currently model interpreters are written for every domain-specific modeling environment. The interpreter encapsulates the information available in the domain-specific modeling environment, and understands the models and their relationships. Then particular translations are made to transfer the information in the domain-specific modeling environment into outputs (the application program, configurations for the system or anything desired). Writing an interpreter is not a trivial task. The programmer needs to understand the models in the domain-specific modeling environment, every detail of the output, and the relationships between the domain-specific modeling environment and the output. Most of the time, the models in a domain-specific modeling environment are very complex and hard to understand for people who have little or no experience on writing model interpreters. While the inputs of the interpreters are known (models in the domain-specific modeling environment), the output of the interpreters may vary depending on the different systems and requirements. The model interpretation process is similar to the back-end of compilers in some sense. They

always need to traverse the model structure, capture the relationships between the models and the attributes of the models, and transform this information into the desired outputs (machine code, in the case of compilers). Although, it is not easy and obvious to know how this process can be implemented, it is possible and desired to have a way to specify the similar behaviors of the model interpreters and automate the repetitive programming tasks in writing large, uninteresting portions of the model interpreter.

The Multi-Graph Architecture (MGA), being developed at the Institute for Software Integrated Systems at Vanderbilt University, is a toolkit for creating domain-specific MIPS environments (DSME). MGA provides a means to quickly and accurately evolve domain-specific applications. It has been successfully used in several application areas.

The MGA includes a metamodeling environment, which allows modeling of the domain-specific modeling environment by creating a model that describes a particular domain-specific MIPS environment (DSME) [1]. This is referred to as a metamodel. A metamodel specifies both the syntactic and semantic behavior of a DSME, then is used to synthesize the DSME itself. This allows the entire design environment to be evolved when the system requirements change. Both the domain-specific application and the DSME can evolve easily and efficiently over the full lifecycle of the system. However, the metamodeling environment of the MGA does not provide a way to model domain-specific model interpreters. When the system requirements change, the DSME will be updated to reflect the changes of the requirements. The domain-specific model interpreters must be recreated or modified by hand. Updating the existing interpreters is expensive and difficult work. A way to provide the ability for MGA to create a set of

model interpretation specifications in the metamodeling environment, and automatically generate the model interpreters is important and necessary.

My thesis is to develop and create a visual specification method and environment for modeling MGA interpreters, and integrate it into the MGA metamodeling environment, then to synthesize the domain-specific model interpreters automatically. This will allow complete domain-specific modeling environments to evolve during the system full lifecycle, and therefore will reduce the cost and risk of large-scale computer based system development.

This thesis begins by briefly discussing model integrated computing, model integrated program synthesis, and the MultiGraph Archticture developed by ISIS at Vanderbilt University. Then, two possible approaches specifying the model interpreter in MGA are discussed. Finally, an approach specifying the domain-specific model interpreters is presented and integrated into the existing metamodeling environment of the MGA.

CHAPTER II

BACKGROUND

Model Integrated Computing

The basic ideas and concepts of Model Integrated Computing are introduced in this section.

Modeling Techniques

Historically, various methods and technologies have been attempted to minimize the impacts of increasing complexity of software systems and computer technology used in industry. Engineers must manage the complexity that results from evolving system requirements, system content, or system context. The potential cost of system modifications to evolve and maintain software systems can be extremely large, according to the scale of the system. Even a small change or modification in one area may cause unexpected impact on others. Also modifying existing system may involve unforeseen risks due to unrealized mistakes of the designer. To avoid this, systems must be designed to evolve, and to manage changes and complexity of the system in a way to minimize the risks and cost involved during the evolution of the system [1].

Domain specific modeling concepts have been introduced into software engineering to manage the complexity of evolving software systems. System requirements exist in a context (domain). In order to support the requirements, the requirements must be analyzed and understood, and appropriate knowledge about the requirements and the domain must be captured. Models are the complete abstractions of systems or contexts (domains) [2]. They capture the structural and behavioral features of

the system and their related context. The most important feature of a model is its ability to reduce the complexity of a design [1]. Modeling is the process of creating models. The modeling focuses on those things that are relevant (essential) while avoiding those things that are irrelevant (incidental) to understand the system. It captures the most important and essential information of the system.

Model Integrated Computing (MIC) is a methodology to allow systems to be designed to evolve. MIC uses domain-specific modeling concept to capture the essential information about the system and its context as models. These domain-specific models can then be analyzed and validated from different levels or views (structural or behavioral). Also various computational transformations of the models can be performed.

Modeling Integrated Program Synthesis (MIPS) is a specific approach to MIC. In MIPS, models are created for a particular domain. Then models are analyzed, validated and translated into either the application program, or the configuration for the run-time application environment. Once the system requirements or context change, the models can be updated and analyzed. The application program can be regenerated from the updated models. MIPS facilitates the management of complex software systems and enables easy system modification and generation, reduces the cost and risk of system modifications, and therefore allows safe and efficient application evolution.

In a MIPS environment, a paradigm is used to describe the structural and behavioral organization of the systems in an abstract form. Paradigms define the models available in a domain, the relationships between the models, and the attributes of the models. Once a paradigm is created for a domain, a model builder tool allows designers form a model diagram by creating models, which are usually specified graphically. Then

specific model interpreters need to be created for the paradigm to transform the diagrams into desired outputs.

MetaModeling

A model is an abstraction of a system. Models describe the elements, their attributes and relationships between the elements in a system. The metamodel is a model that defines a modeling language. It captures information used to describe or define domain-specific models. In order to define models, a modeling language or environment must be employed, which consists of a collection of concepts (semantics) with a notation (syntax) and rules governing the concepts and notation [2]. A metamodel formally defines the syntax and semantics of a particular domain-specific modeling language or environment.

Why do we need to study or use the metamodeling concept here? In MIPS, the paradigm is created and used to describe the elements and their relationships in a system. Models are created according to the paradigm. Then interpreters are developed to translate the models into executable systems. The paradigm, models, model builder, and model interpreter form a domain-specific modeling environment (DSME), which describes a particular solution to the requirement of a system. Once the system content or context changes, models in the DSME can be updated to reflect the changes and application program can be regenerated. However, during the lifecycle of the large-scale computer based system, requirement of the system may also need to be modified to meet some particular purposes. In this case, the changes of the system requirements will lead to the changes of the solution to the requirement, which mean changes in the DSME. The

paradigm must be modified to reflect the changes of the requirements, the models must be modified or recreated to describe the solution to the new requirement, and interpreters must be modified or recreated to incorporate the changes in the requirements and solutions. Here lies the DSME evolution problem. Building a DSME is not a trivial task. Modifying an existing DSME to meet the new requirement is even worse than creating a new one. It is obvious that if we could use MIPS to model and manage the design and evolution of application systems, why could not we use the MIPS to model and manage the design and evolution of the DSME. This is the purpose of the metamodeling environment. The metamodeling environment provides a paradigm that captures and describes the generic syntax and semantics used to create DSME. Metamodels are created to describe a particular domain-specific modeling environment, and metamodel interpreters are used to translate the metamodels into the paradigm of the particular DSME. Refer to [1] for more detail about the metamodeling concept.

MultiGraph Architecture (MGA)

MultiGraph Architecture (MGA), being developed at the Institute for Software Integrated System at Vanderbilt University, is a toolkit for creating domain-specific MIPS environments. The structure of MGA is illustrated in Figure 1.

In the MGA, the Metaprogramming Interface provides a way for designers to specify and model the model paradigm of an application domain. The modeling paradigm is the modeling language of the domain specifying the modeling objects and their relationships [3]. The syntax, semantics and graphical representation of the modeling paradigm are specified in the metamodeling environment. In the metamodeling

environment, Unified Modeling Language (UML) class diagrams are used to specify the syntax of the model paradigm. The Object Constraint Language (OCL) is used to specify the semantics of the paradigm. The syntax, semantics and graphical representation specification of the paradigm form the metamodels in a metamodeling environment. These specifications are automatically translated into the paradigm of the MIPS environment.

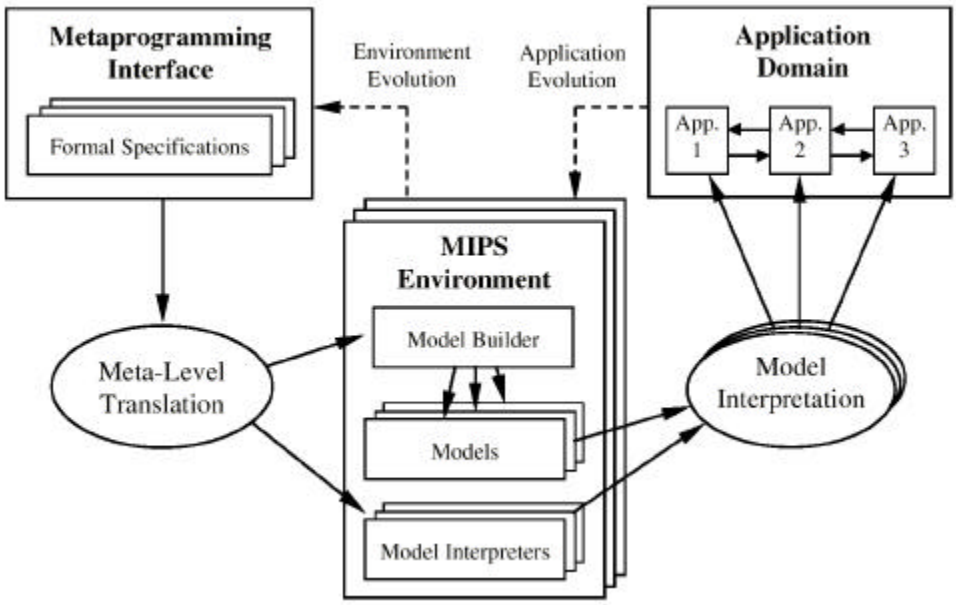

Figure 1: The Multigraph Architecture

A MIPS environment consists of a model builder, models, and model interpreters. The model builder is used to build models according to the model paradigm. The model interpreter translates the models built in the model builder into the applications, configurations of the analysis tools, or any desired outputs.

Modeling Concepts in the MGA

The following modeling concepts are used in the MGA to represent a generic and domain-independent approach to describe various modeling environments.

- Modeling paradigm
- Categories
- Atoms
- Models
- Attributes
- Connections
- Ports
- References
- Hierarchical containment
- Conditionals
- Aspects

The **modeling paradigm** describes the syntax and semantics of a particular domain. It defines what models or objects are presented in the domain, what are the attributes of these objects and how are they related. The **category** groups a set of related models which together perform a particular function of a system or construct a subset of a system.

The **atoms** and **models** are the objects available in a domain. An atom is the smallest elements in a domain, which can not contain other objects. A model is a compound object that can contain other models or atoms. For each kind of atom and model, a set of attributes can be defined and associated with them. An attribute is a

property of an object. The attribute values are user changeable. **Connections** present the simplest and most obvious relationships between objects. Connections can be created between atoms, atom references and ports. Connections cannot connect two models directly. They can only be connected through ports. Connections can be directed or undirected and can only connect two objects that are in the same parent model. **Ports** are parts of a model, through which the model can connect to other models or atoms in the same containment relationship (in the same parent model).

The connection concept can only express the relationship between two objects in the same hierarchy (connections can only be made between two objects that have the same parent model). In order to represent the relationships between objects in different hierarchies, or even in different model hierarchies (Categories), **references** are used in MGA. A **reference** works like a pointer in object-oriented programming languages. It provides a way to reach other objects out of the model scope. Like connections, references must be contained or defined in a model.

Connections and references can only represent the relationships between two objects. The **conditional** concept provides a way to express the relationships between two sets of objects. The first set of objects is called the conditional controllers, which consist of the same kind of domain objects (models, atoms or references). Usually the controller set has a single element. The second set is called the parts of the conditional. These can be of several different kinds of objects or connections [3].

As mentioned above, models can contain other models or atoms as parts. This **containment** relationship creates the hierarchical decomposition of the models. Any objects (models or atoms) can have at most one parent model.

Aspects provide a way to view the models from different points of view. Every model has a set of predefined aspects. Every part (other model, atom, connection or reference) of the model belongs to at least one aspect, and is visible or hidden in each aspect. The aspect concept provides primarily visibility control for the model. For a more detailed description of the MGA modeling concepts please refer to [6].

Graphical Modeling Editor

MGA provides a set of tools to perform and automate (1) building, checking, storing and generating model paradigms, (2) building, checking, storing models, (3) transforming the models into applications and/or inputs or configurations for system engineering analysis tools, and (4) integrating applications on heterogeneous parallel/distributed computing platforms [4,5].

At the core of the MGA MIPS environment is the Graphical Model Editor (GME) [6]. It provides a generic open structure to integrate all the MGA concepts and tools together. It provides interfaces to human modelers using a well-defined syntax (graphical, tabular, and textual). It can load and understand the model paradigms and provides an easy to use graphical interface to allow designer to manipulate the models. It is also integrated with the metamodeling layer and constraint manager to facilitate creating and maintaining the models. Models can be stored in or loaded from either the MS Repository or Object Oriented databases. GME uses the Component Object Model (COM) technology to achieve the component integration. It provides a COM interpreter interface that supports the accessing and modification of the models in the GME. The interpreters can be written in any language that supports COM. The GME COM interface allows the

interpreters to access all the details and concepts of the models in the GME, perform necessary syntax or semantics checking, and to modify the models. Upon the GME COM interpreter interface, a high-level C++ interpreter interface is implemented and provided in the GME. This high-level C++ interpreter interface wraps the COM interface and provides easier using functions to ease the programming task of the designer.

Model Interpreters

The main goal of this thesis is to study and create a generic approach specifying the behavior and structure of MGA model interpreters. This approach should allow designers to specify and model domain specific model interpreters in the metamodeling environment. The model interpreter specification together with the modeling paradigm specification constitutes the complete metamodeling specification for a DSME. In this way, the entire DSME can be specified, modeled, and automatically synthesized from the metamodeling environment, therefore allowing the evolution of the entire DSME.

A model interpreter acts like the back-end of a compiler in sense. It traverses the model structure (which can be viewed as a tree or graph) in specific order, retrieves information of the objects, performs actions during the traversing and visiting of the objects, and generates various desired outputs.

It is sometimes easy to describe the behaviors of a model interpreter, however it is highly non-obvious how to generically specify and implement them. There is no available method to specify the model interpreter in a high-level programming layer.

It is natural to think the behavior of a model interpreter as a series of actions taken during the traversing and visiting objects in a structure form, which is summarized as the following steps:

1) Traversing and visiting the objects in the model structure in a specific sequence.
2) Inspecting and capturing the relationships between the objects and attributes of the objects.
3) Performing relevant actions during visiting and traversing the object and translating the captured information into desired outputs.

In order to specify and implement these steps in a generic way, first we need to have a static and well-defined structure of the objects in the source domain so that the traversing and visiting procedure can be defined and associated to the structured form. Second, a concise and sophisticated way to specify all the possible and complex traversal sequences must be developed. Finally a way to merge or associate the actions with the traversing procedures to specify the order and duration of the actions must be developed. All these techniques must be simple enough to use, but sophisticated enough to specify all the possible situations that may happen during the interpretation procedure.

In the following sections, relevant techniques that may be useful in the high-level model interpreter specification and transformation are studied.

Compiler Techniques

The first and foremost application of graph traversal and actions is the intermediate code generator part of a compiler [7].

Compiler Background

A compiler is a program that reads a program written in one language – the source language – and translates it into an equivalent program in another language – the target language [15].

There are two stages to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent parts and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often, a special kind of tree called syntax tree is used, in which each node represents an operation, and the children of a node represent the arguments of the operation. Figure 2 shows the general phases in which a compiler operates. Each of these phases transforms the source program from one representation to another.

As translation progresses, the compiler's internal representation of the source program changes. A lexical analysis phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword, or a punctuation characters.

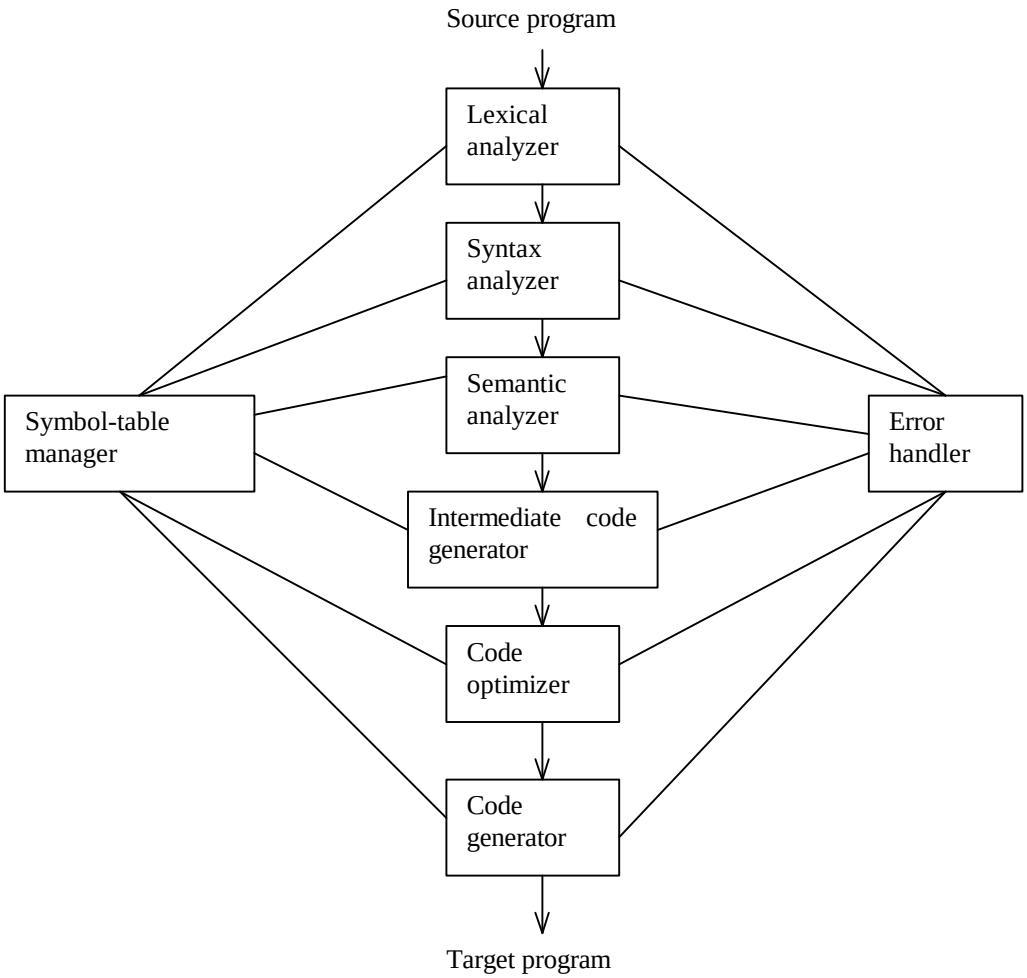

Figure 2: Phases of a Compiler

Syntax analysis (parsing) is kind of hierarchical analysis. It imposes a hierarchical structure (syntax tree) on the token stream. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize outputs. Usually, the grammatical phrases of the source program are represented by a syntax tree [7]. The syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the operands of an operator are the children of

the node for that operator. Figure 3 shows a syntax tree for the expression " position := initial + rate * 60".

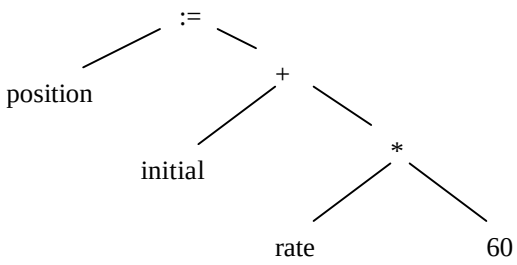

Figure 3: A Syntax Tree Example.

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.

After syntax and semantic analysis, some computers generate an explicit intermediate representation of the source program. We can think of this intermediate as a program for an abstract machine. The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result. The final phase of the compiler is the generation of target code. The intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

After building the syntax tree from the input text, and performing semantic analysis, compilers traverse the syntax tree and output the generated code. Compiler research literature provides a great source for efficient traversal and transformation algorithms. Also, the area of automatically generated compilers provides some interesting

technologies for the structured specification of graph traversals. Attribute grammar is the widely used one. In the following section I will give a brief overview of Attribute Grammars, and their possible application to model interpreter specification.

Attribute Grammars

Attribute Grammars were introduced by Knuth [8] more than thirty years ago, and since then they have been widely studied and have been proved to be a useful formalism for specifying the context-sensitive syntax and the semantics of programming languages, as well as for implementing editors, compilers and compiler-writing systems. An attribute grammar is a declarative specification that ties semantic specification to the syntactical rules of a programming language, and describes how attributes (variables) are computed for rules in a particular syntax [9].

An AG uses a context-free grammar (a notation for specifying the syntax of a language) to specify the syntactic structure of the input. It associates a set of attributes with each grammar symbol, and associates a set of semantic rules for computing values of the attributes associated with the symbols with each production. The grammar and the set of semantic rules constitute the AG.

An AG forms an extension of a context-free grammar framework in the sense that information is associated with programming language constructs by attaching attributes to the grammar symbols representing these constructs [9]. Each attribute has a (possibly infinite) set of possible values. Attribute values are defined by attribute evaluation rules (semantic rules) associated with the productions of the context-free grammar. These semantic rules specify how to compute the values of certain attribute occurrences as a

function of other attribute occurrences. There are two notations for associating semantic rules with productions, syntax-directed definitions, and translation schemes. Syntax-directed definitions are high-level specifications for translations. They hide many implementation details, and free the user from having to specify explicitly the order in which translation takes place. Translation schemes indicate the order in which semantic rules are to be evaluated, so they allow some implementation details to be shown.

Conceptually, with both syntax-directed definitions and translation schemes, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse–tree nodes (see Figure 3). Evaluation of the semantic rules may generate code, save information in a symbol table, issue error messages, or perform other activities. The translation of the token stream is the result of evaluating the semantic rules.

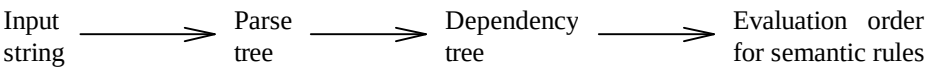

Figure 4: Conceptual View of Attribute Grammar

The attributes associated with a grammar symbol can represent anything we choose: a string, a number, a type, a memory location, or whatever. They are divided into two disjoint classes, the synthesized attributes and the inherited attributes. The attribute evaluation rules associated with a grammar production define the synthesized attributes attached to the grammar symbol on the left side and the inherited attributes attached to the grammar symbols on the right side of the production [9].

A context-free grammar assigns a tree structure to each sentence. One could think of the nodes for the grammar symbols in a parse tree as records with fields for holding

information, where the field names correspond to attributes. The values of the synthesized attributes at a parse tree node and the inherited attributes at its immediate descendants are defined by the attribute evaluation rules associated with the production applied at that node. The value of a synthesized attribute of the parent is computed from the values of attributes at its children and (possibly) other attributes of the parent itself. The values of an inherited attribute of a child are computed from the values of attributes at its parent and its siblings and (possibly) other attributes of the child itself.

Generally speaking, a synthesized attribute attached to a tree node contains information concerning the subtree at that node. Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears [9].

Semantic rules set up dependencies between attributes, which are represented by a graph. If an attribute *b* at a node in a parse tree depends on an attribute *c*, then the semantic rule for *b* at that node must be evaluated after the semantic rule that defines *c*. The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph. From the dependency graph, we can derive an evaluation order for the semantic rules. Evaluation of the semantic rules defines the values of the attributes at the nodes in the parse tree for the input string. A semantic rule may also have side effects, e.g. printing a value or updating a global variable. The evaluation order derived from the dependency graph is a topological sort of a directed acyclic graph. Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.

The translation specified by an attribute grammar can be summarized as follows: The underlying grammar is used to construct a parse tree for the input. The dependency graph is constructed by the interdependency of the attributes defined by semantic rules. From a topological sort of the dependency graph, we obtain an evaluation order for the semantic rules. Evaluation of the semantic rules in this order yields the translation of the input.

There are many well-studied algorithms and implementations to construct the translators, including constructing dependency graphs, finding topological sort of the graphs, and evaluating the semantic rules in order and generating the outputs. Here, I only focus on the specification approaches toward evaluation, and how they are related to the model interpreter specification.

Specifying Model Interpreters with AGs

Attribute Grammars specify a way to walk the tree and calculate the values of the attributes attached to the nodes of the tree [10]. In the MGA, a paradigm is used to define the structure of the modeled system. In general, a software structure can be viewed as a graph. The objects are the nodes in the graph. The edges between the nodes represent the relationships between the objects. The graph can be reduced to a tree, if we only view or study some specific relationships (containment, aggregation and inheritance) between the objects. In this way, the model interpreter can be viewed as a translator that traverses the tree in specific order, performs actions associated to the nodes in the tree, and generates the outputs. This is very similar to the way an Attribute Grammar works, so it is possible

for us to construct an Attribute Grammar that specifies the translation of a paradigm in the MGA.

Although the Attribute Grammar provides a very high-level specification formalism for the traversal of a tree through the use of dependency among attributes [10], unfortunately, Attribute Grammars have very serious practical limitations. First the construction of an Attribute Grammar is a non-trivial task. The attributes must be set up over the whole tree for a specific traversal order. Sometimes one has to introduce extra attributes just for forcing a particular kind of traversal [10]. If the software system is complex enough, assigning attributes to the tree to specify a behavior of the interpreter will be extremely painful. Some AG specifications may be as large as or even larger than a manually developed implementation for the same task. Second, in order to apply attribute grammars, the software system must be viewed as a tree. In this way, some relationships specified by the MGA may or will not be accessed easily. Also largeAGs lack a structure to improve comprehensibility and maintainability. Thus, while Attributed Grammars offer a very high-level formalism for the structured specification of traversals of graphs, their usability is limited.

Adaptive Programming

Introduction to Adaptive Programming

The adaptive programming concept was developed by Lieberherr [11] and others to support evolutionary development of object-oriented software. Object-oriented (OO) programming has been a great success since its appearance. Part of its success is because of the flexibility that object encapsulation provides. OO allows both the data and

behavior to be encapsulated in the class. This provides a more natural description of the real world and reduces the semantic gap between a program and the world it models. However OO doesn't provide comparable support for flexibility in object interrelationships. It has been noted that OO programs follow a pattern of collaborations where multiple objects of different classes cooperate to achieve a certain goal [10]. A complex behavior in OO program is implemented by a set of simpler behaviors distributed over a set of objects. In this case, the tight binding of data and function in OO becomes one of its disadvantages. This leads to programs that are hard to evolve and to maintain. It requires coding in the smallest details of the relationships among objects. Often, a great deal of code needs to be edited in the face of even a small change to the conceptual interdependence class structure [11].

Adaptive Programming is designed to allow the flexibility of the relationships between functions and data. That is, functions and data are loosely coupled through navigation specification. Adaptive means that the software heuristically changes itself to handle an interesting class of requirements changes related to changing the object structure [11]. It works by having the programmer program at a higher schematic level that abstracts away details such as navigation paths. These schematic patterns can be instantiated to a particular class graph to arrive at an executable program. It allows us to express the "intention" of a program without being sidetracked by the details of the objects structure [11]. Adaptive programming techniques provide a novel solution to the maintenance and evolution of OO software. It also provides valuable information and techniques for the model interpreter specification.

The main goal of the Adaptive Programming is to make the programs structure-shy by using only minimal information about the implementation-specific class structure when writing the behavior. Adaptive programs are written using two loosely coupled fragments: behavior and implementation class structures. It provides a high level description of class structures (Class Dictionary Graph) and a high level language for describing object behavior (propagation pattern, traversal strategies with visitor classes). An adaptive object-oriented software development tool called Demeter has been developed in Northwest University, which provides more valuable practical experiences.

Demeter

Demeter [11] is an object-oriented software development environment. It provides tools to help you design the two key components in any object-oriented system: object structure and object behavior. In Demeter, these two components are represented by the concepts of class dictionary and propagation pattern, respectively.

The class dictionary in Demeter defines the class structure of an application. It defines the classes of the application and the relationships between the classes. In particular, it describes inheritance (is-a) relationships and aggregation (part-of) relationships. Informally, class dictionary graphs express object-oriented class hierarchies as mathematical graph structures. In a class dictionary graph, concrete and abstract classes are represented as vertices, and part-of and inheritance relationships are expressed as edges. Demeter will automatically generate the C++, or Java classes definitions from the class dictionary. In Demeter, a class dictionary is defined by a high-level specification language. It also allows limited UML usage in the specification of the class dictionary.

A propagation pattern defines a specific behavior for a collection of classes. The behaviors of the application as a whole are described by a set of propagation patterns. The advantage of using propagation patterns as opposed to regular C++ code is their adaptiveness. Propagation patterns avoid hard coding the details of the class structure into the program, and therefore de-couple the class structure as much as possible from the program. Consequently, the program is less prone to change when the class structure is modified. In addition, programs written with propagation patterns are more concise since trivial structure traversal code does not need to be specified, and is again left to the code generator (savings are typically 50% or more) [12]. Demeter provides a high level language to specify the behavior of the application separately from the structure (propagation pattern), which is also very valuable to the study of model interpreter specification. In the following section, I will give a detail description of Demeter propagation pattern with the traversal strategies.

Propagation Pattern

Propagation patterns define the behaviors of the objects in terms of collaborating groups of classes. Every propagation pattern generally defines a function or method (called an operation in Demeter) of the application. So, for an application there may be numbers of propagation patterns defined. In general, a propagation pattern consists of 4 logical blocks:

1) Signature

A propagation pattern specifies a function or method of the system. The signature of the propagation pattern defines the method or function's signature for the set

of classes involved in this function, which includes name, parameters and return type of the function.

2) Traversal specification

A traversal is a navigation through a group of related objects with the purpose of accomplishing some task. Traversals define the set of classes involved in the operation in a succinct form, in terms of paths in a class dictionary graph [12]. From the implementation point of view, a path represents a sequence of object invocations, following the order of the edges in the subgraph defined by the traversal. Traversal defines a subgraph of the class dictionary graph that specifies the precise strategy to traverse the subgraph. The strategy is a very compact and high-level specification of the traversal: it simply refers to the classes involved, omitting all implementation details. For example: if class A is associated with class B, which is associated with class C, a traversal can simply specify " from A to C" without mention the intermediate class B, to specify traversal from A to C through B [10]. The paths to traverse the subgraph specified by the propagation pattern are computed based on the propagation patterns and class dictionary graph. The low-level code that traverses and navigates the subgraph is generated automatically by the tool. Traversal specification must include, at least, the source classes (*from* ClassSet), i.e. those classes for which the operation applies. They may include the target classes (*to* ClassSet). They may also restrict the paths by using the primitives: (*bypassing* EdgeSet), (*through* EdgeSet), and (*via* ClassSet).

3) Transportation specification

In many cases, it is necessary to pass information from one class to others along the traversal. Transportation specification declares the internal argument objects to be

carried along portions of the traversal. The transportations of objects are made along transportation traversals, which are subsets of the propagation pattern traversal. The carried objects can be (*in*), (*out*) or (* inout*). These qualifiers have the same meaning as in many interface definition languages: (*in*) objects can be read along the traversal and their modification will not be visible outside; (*out*) objects are to be written to somewhere along the traversal, so that they bring out information.

Carried objects can be initialized when they are declared. By default, they are initialized at the source class of the transportation directive. Transportation of objects is internally implemented by the Demeter tools as a signature extension, that is, the corresponding methods are augmented with more parameters. The variable names of the carried objects are available in all classes involved in the transportation path.

4) Code wrappers

Code wrappers are pieces of C++ or Java code attached to some classes, which are involved in the propagation pattern. The C++ or Java codes are the implementation of the operations or actions taken in the classes and edges along the traversal. The wrappers can be "*prefix*" or "*suffix*". The "*prefix*" codes are executed before the traversal enter the classes or edges, the "*suffix*" codes are executed after the traversal leaves the classes or edges. The codes in the wrappers are no concern of the Demeter. They are low-level implementation codes entered by the designer, and it is the designer's responsibility to guarantee the correctness of the codes in wrapper.

Demeter tools read the propagation pattern and class dictionary graph, check their consistency, and synthesize all the traversal codes, which are distributed across classes as methods. The automatic generation of these codes removes the mundane tasks from the

programmer: iterating over lists, invoking methods on objects in the list, and hand-coding the traversal of a quite complex graphs with the help of small, distributed methods. The code also incorporates the user-specified code fragments that are executed during traversal [10].

Adaptive programming provides a compact and efficient way to specify the traversal of a software structure. In AP, user defined low-level code can be integrated with the specification to perform various desired functions of the system. The actual traversal codes are synthesized; designers are not burdened with low-level implementation details of the traversal. All of these concepts can be used to model or specify the model interpreter naturally. The main task of a model interpreter is to traverse the software structure, and perform various actions during the traversal, which generate the desired outputs. So in this thesis, I use the propagation pattern of Adaptive Programming as the main concept to study a way to visually specify the model interpreter in the GME, and construct a metamodeling environment to automatically generate the model interpreter for a specific domain.

CHAPTER III

APPROACH

The goal of this thesis is a metamodeling environment that can visually specify MGA model interpreters for specific domains, and synthesize the model interpreters automatically. Based on the observations made above, a visual specification of the model interpreter has been developed by using the propagation pattern concepts of the AP. Also, a preliminary metamodeling environment has been created to allow designers to specify and automatically synthesize the model interpreter for a specific domain. This metamodeling environment is integrated with the existing UML/GME metamodeling environment to achieve the complete metamodeling of a Domain Specific Modeling Environment (DSME) and evolution of the entire DSME.

Based on the background study in chapter 2, in order to specify a model interpreter, the following information or components should be defined:

1) Model Structure specification (class dictionary in AP). The model structure defines the entities (classes) available in a model paradigm and the relationships between these entities.
2) Behavior specification of the model interpreter (propagation pattern in AP). The behavior specification specifies a function or method of a model interpreter. It includes how the models should be traversed, what actions should be taken during the traversal, in what order the actions should be taken, and how the possible objects or

variables should be carried along the traversal. The behavior specification is closely related to the model structure specification.

3) A meta-interperter that understands the syntax and semantics of the model structure and behavior specification of a model interpreter and translates the specification in the metamodeling environment into the model interpreter.

The following sections illustrate how these components are achieved in the metamodeling environment of GME.

Model Structure Specification

The main task of model structure specification is to provide a succinct and easy to understand form to describe the classes available in the modeling paradigm, and the relationships between these classes.

In the existing UML/GME metamodeling environment, the general design methodology utilizes UML to specify the modeling paradigm syntax, and uses MCL to specify semantics. Presentation specifications are created to take the form of a mapping between the UML entities and relationships and GME objects representing elements of the target modeling environment [1]. The UML class diagram in the existing metamodeling environment specifies the classes, attributes, and relationships between the classes. It is a perfect specification language to express the model structure in this situation.

Because the UML class diagram of the existing metamodeling environment provides a ready to use model structure specification of the paradigm, we will use it as the model structure specification for the model interpreter specification.

Figure 5 shows the UML metamodel representation of a simple modeling paradigm called the ACME [20].

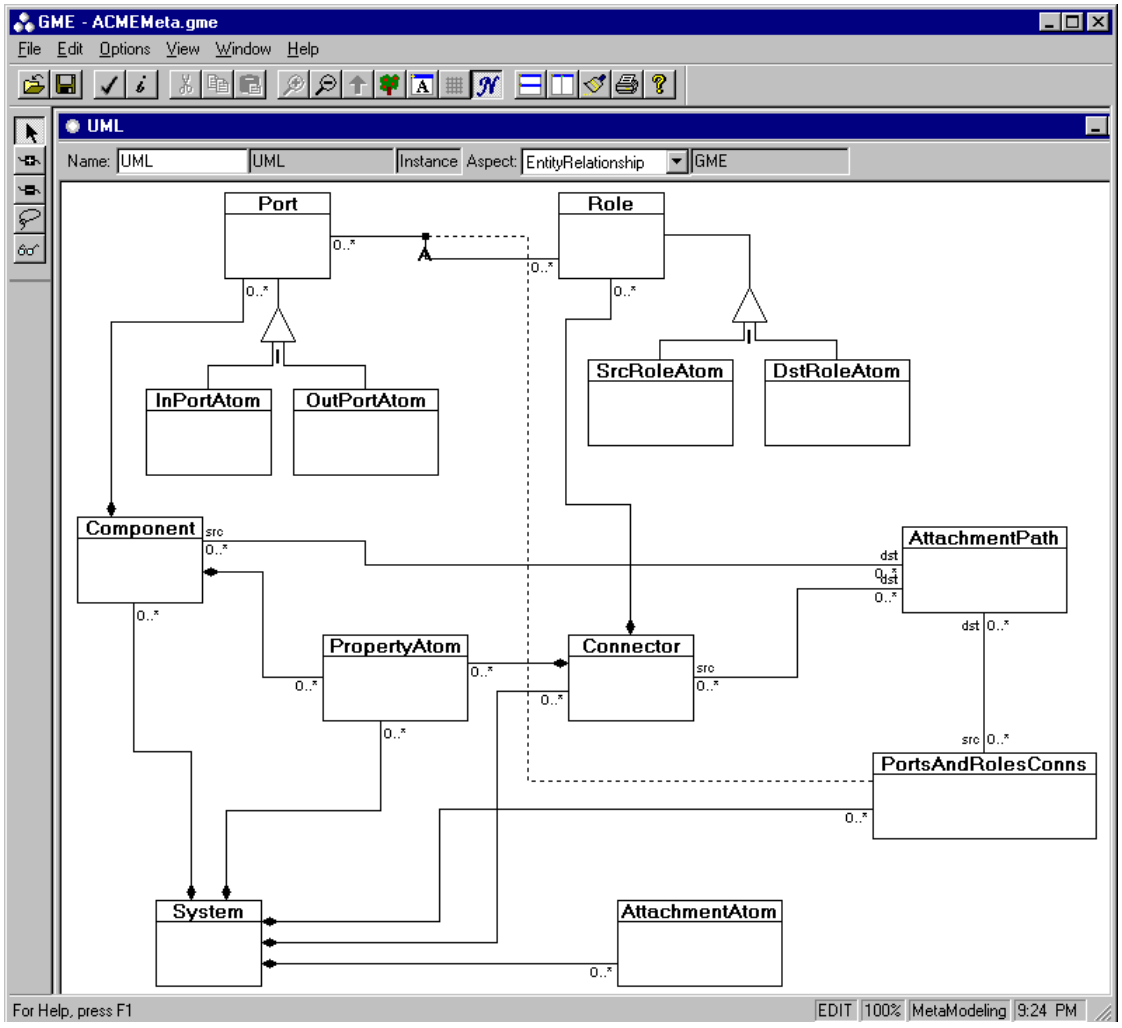

Figure 5: UML model of the ACME metamodel

We will use ACME as an example to illustrate the model interpreter specification concepts. Before I go into the detail of the model interpreter specification, I will explain a little bit more about the ACME modeling environment and what the ACME model interpreter will do.

ACME is an Architectural Description Language (ADL) used to describe software systems. ACME uses components and connectors as basic architectural elements, both of which have interconnection interfaces. Components and connectors are interconnected in a straightforward manner – component ports attach to connector roles, and connector roles attach to component ports – to form representation of software architectures. Sets of connections and components may be grouped together to form attachment groups. ACME uses aggregations of components, connectors, and attachment groups to form systems – larger and more complex software representations [20].

ACME is text-based. However, because the ACME metaphor of design is popular and well known, a graphical representation of the language would be of use to a large community of software system designers [1]. For this reason, the ACME metamodeling environment has been designed and created to allow designers to create ACME by using a graphical representation. After the ACME models are created by using GME, the model interpreter will translate those graphical modeling concepts into text-based ACME models.

In the UML class diagram, the classes can be viewed as nodes in the class dictionary graph of AP. Several kinds of associations are allowed in UML class diagrams to specify the relationships between classes. Among these associations, only the inheritance and aggregation associations will be viewed as edges in the class dictionary graph. These will be used to compute the traversal path of the behavior specification (introduced in next section). Through these two kind of associations, all the entities in a model paradigm can be reached from a model, except the GME reference objects. The reason we only use inheritance and aggregation in the model structure specification is to

avoid the possible circular traversal paths in the object graph instanced from the class graph. If we only use or traverse along the inheritance and aggregation edges, the instanced object graph can be viewed as tree, which guarantees the avoidance of circular traversal path [11].

Figure 6 shows the UML class diagram of ACME metamodel with only inheritance and aggregation associations. If we specify that the traversal will always progress from the model to its parts (if it has parts) and from the base class to its derived classes (if the class is base class). Then we can prove that the possible traversal paths will not be circular. From figure 6 we can see that all the possible traversal paths are among the following:

- System -> Component -> Port ->InPortAtom or OutPortAtom
- System ->Component -> PropertyAtom
- System -> AttachmentAtom
- System -> PropertyAtom
- System -> Connector -> PropertyAtom
- System -> Connector -> Role ->SrcRoleAtom or DstRoleAtom
- System -> PortsAndRolesConns

Through these paths, we can reach every kind of the classe specified in UML class diagram from *System* class, except for *AttachmentPath*, which is mapped to a reference object in the GME paradigm according to the presentation specification of existing metamodeling environment. (Refers to [1] for more detailed information).

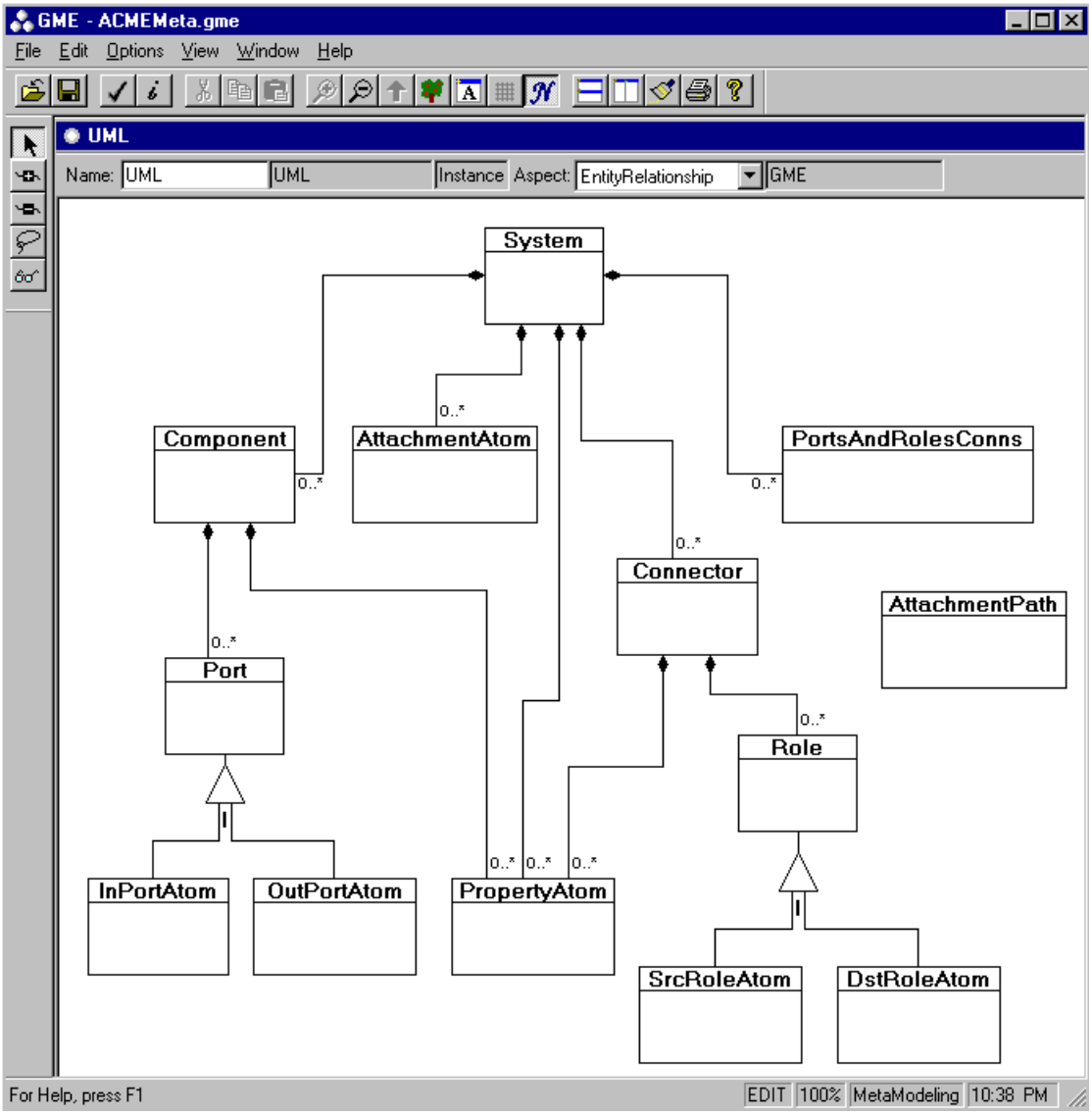

Figure 6: UML Diagram of the ACME with Only Inheritance and Aggregation

Behavior Specification

The behavior specification addresses the following questions for the behaviors of a model interpreter:

1) Signature of the behavior: This should specify the parameters of an operation (an atom that groups all the traversal specification, transportation specification and actions specification in the metamodeling environment).

2) Traversal specification: How should we traverse the model structure? What is the start and end points of a traversal? Which traversal paths should we use?
3) Transportation specification: What objects or variables will be carried along the traversal path during the traversal? What will be the initial values of the objects or variables?
4) Action specification: What actions will be taken during the traversal?
5) Visiting sequence specification. In what order the parts of a model should be visited during the traversal?

The existing metamodeling environment specifies all the necessary information (syntax, semantics and presentation) of a DSME paradigm. In order to integrate the model interpreter specifications into the existing metamodeling environment, a new category called "*metainterpreter*" has been added to the existing UML/GME metamodeling paradigm. The new "*metainterpreter*" category will define all the necessary behavior specifications for a model interpreter, which fully describes a model interpreter together with the model structure specification in the UML class diagram of the existing metamodeling environment.

Figure 7 shows the behavior specification for the model interpreter of ACME metamodeling environment.

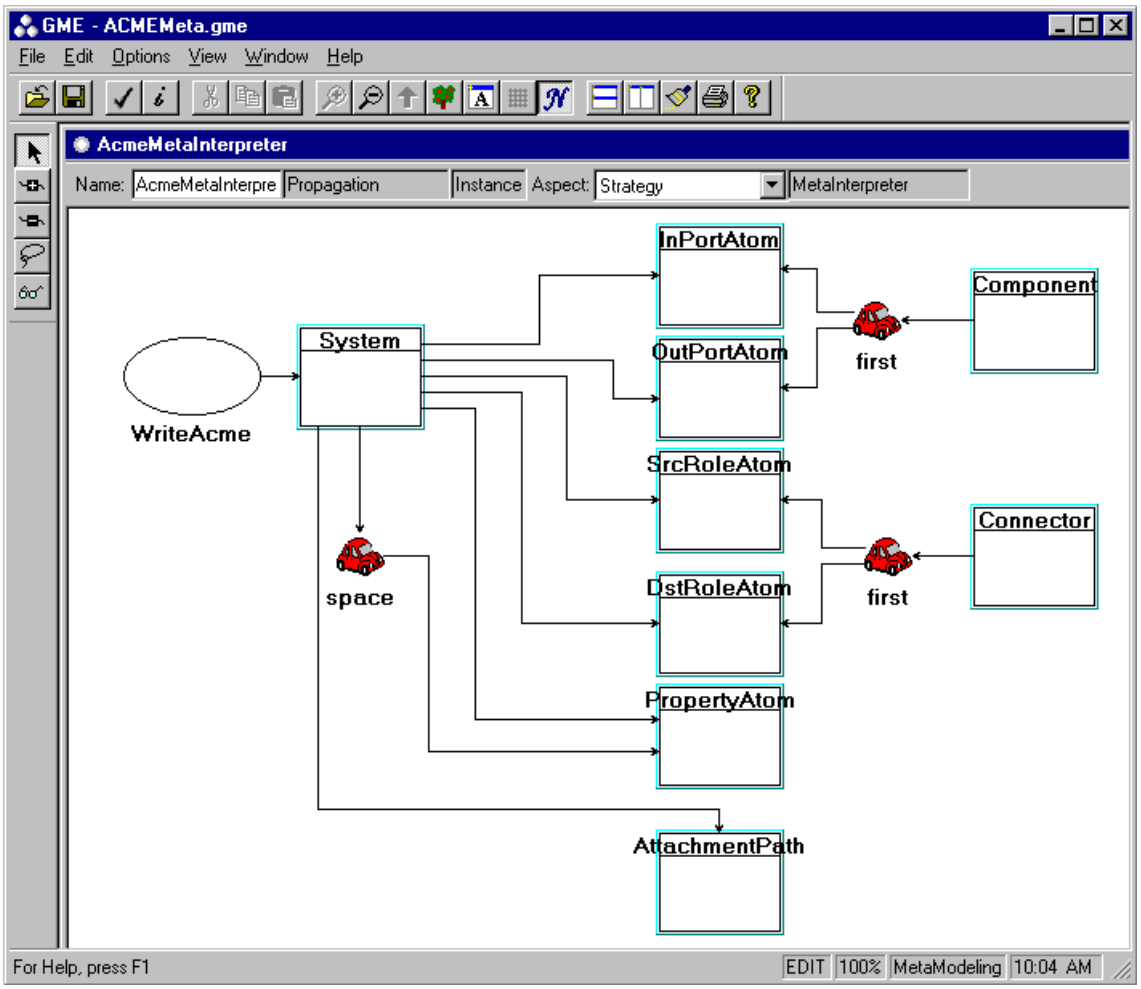

Figure 7: ACME MetaInterpreter Specification

We have seen the UML class diagram of the ACME paradigm and known the available classes in the ACME paradigm and the relationships between classes. The goal of ACME model interpreter is to translate the graphical specification of ACME model into text-based ACME specification. In the following section, we will discuss how the ACME model interpreter is specified in the metamodeling environment.

The Interpreter Model

In the "*metainterpreter*" category of the metamodeling paradigm, "*interpreter*" models are defined, which contain all the necessary information to specify model interpreters. For every "*interpreter*" model in the metamodeling environment, a model interpreter will be synthesized. Because for a specific domain many model interpreters may be defined to translate the models into different outputs, multiple interpreter models are allowed in the metamodeling environments for a specific domain. They will be translated into different model interpreters for the domain. As I explained in the background section, the GME model interpreters can be written in any language that interfaces with COM. In this thesis, the synthesized model interpreters are implemented with C++ code that utilizes the high-level C++ interpreter interface provided by GME.

There are two textual attributes for the interpreter model: "*Header*" and "*Interpret*". The "*Header*" attribute allows designers to define the header information of the "*Interpreter.cpp*" file, which is synthesized from the metamodeling environment. The "*Interpreter.cpp*" file defines the entry point of a model interpreter. It defines a "*CInterpreter*" class with a member function called "interpret", which is the starting point of the interpretation. The "*Header*" attribute provides the header definition of the "*Interpreter.cpp*" file, including include files and global variable definitions.

The "*Interpret*" attribute of the interpreter model allows designers to input the C++ code that calls and executes the operations of the model interpreter specified in the metamodeling environment. The code defined in the "*Interpret*" attribute will be the implementation of the "*Interpret*" member function of the "*CInterpreter*" class, which

comes from the high level C++ interpreters interface. Figure 8 shows the attribute definitiona of the "*ACMEMetaInterpreter*" model.

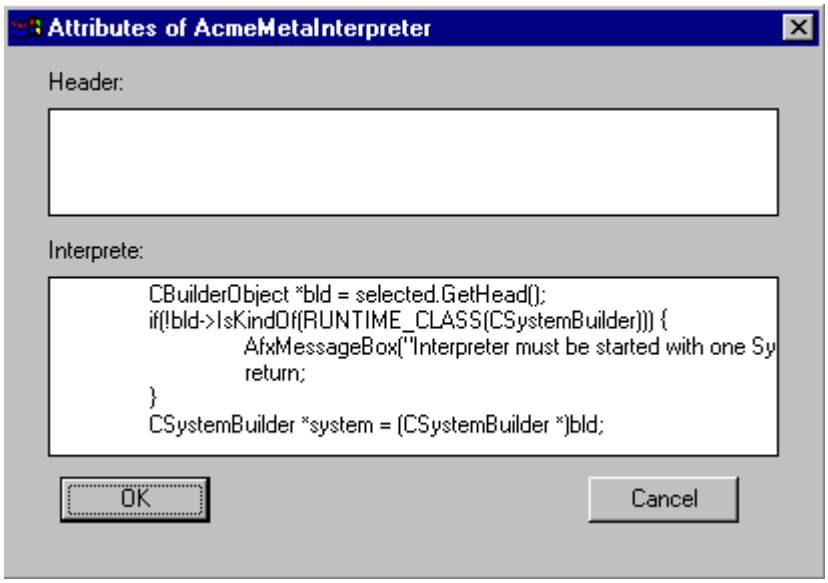

Figure 8: The Attributes of the AcmeMetaInterpreter Model

An operation atom is defined as a part of the interpreter model. It is equal to a propagation pattern definition in the Adaptive Programming. Every operation atom in the interpreter model groups a set of traversal specifications, transportation specifications and action specifications, and indicates a behavior of a model interpreter over a set of collaborative objects. It is represented as an ellipse in the GME. In the ACME model interpreter specification, an operation called "*WriteAcme*" is defined (see figure 7).

Figure 9 shows the attribute definitions of the "*WriteAcme*" operation Atom. There are two field attributes in an operation atom: "*parameters*" and "*Sequence to visit parts*".

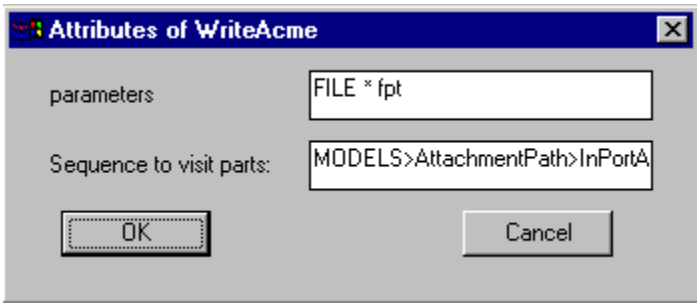

Figure 9: The Attribute of the WriteAcme Operation Atom

The "*parameters*" attribute defines the parameters of the function or the method specified by the operation atom, which is similar to the signature definition of a propagation pattern in the AP. Multiple parameters separated by commas can be defined in the "*parameters*" attribute. The data type of the parameters may be any C++ built-in type or any validated user defined data type. The parameters can be passed by value, pointer or reference. If the parameter is passed as reference the variable value changed inside the traversal will be visible to the outside of the traversal. It can be used to bring back some information or to work as a return variable of the operation.

The "*Sequence to visit parts*" attribute of the operation atom allows designers to specify in which order the parts of models will be visited. For example, in the ACME metamodel, the "*System*" model may have parts of different types. It can have objects of "*Component*", "*PropertyAtom*", "*Connector*", "*AttachmentAtom*" or "*PortsAndRolesConns*" types as its parts. Once we enter a "*System*" model and want to traverse its parts, we need to know in which order these parts should be traversed. This can be specified in the "*Sequence to visit parts*" attribute of the operation atom. The object types are input in the field separated by a ">" sign. The object types specified in the field can be of any object types defined in the metamodeling environment. They can

also be "*MODELS*", "*ATOMS*" or "*CONNS*", which represents a group of models, atoms or connections respectively. For example: "*MODELS* > *AttachmentPath* > *InPortAtom* > *OutPortAtom* > *SrcRoleAtom* > *DstRoleAtom* > *PropertyAtom* > *CONNS*", means that once we enter the model, we need first visit all its model parts in unspecified order; then we need to visit its atom parts in the order specified, which is first visiting "*AttachmentPath"* part, then "*OutPortAtom*", then "*SrcRoleAtom*", then "*DstRoleAtom*", and finially "*PropertyAtom*". After visiting atom parts, we visit connection parts of the model. In the "*Sequence to visit parts*" attribute, designer can supply a partial order of the possible parts to be visited during the traversal. The objects that are not specified in the attribute will be traversed in an unspecified order after all the objects specified in the attribute have been traversed.

Traversal Specification

After defining an operation in the interpreter model, we need to specify the traversal path of the operation. This should answer the question: "if we are at node of type X, where do we go next?" The "next" should be an object that is reachable from objects of type X either directly or indirectly, possibly through inheritance [10].

As I stated in the model structure specification section, we will use the UML class diagram as the model structure specification. In the UML class diagram, we view model structure as a tree. The classes are nodes and the inheritance and aggregation associations are edges in the tree. All the traversals will go from a model to it parts, or from a base class to the derived classes. Under this restriction, the class graph specified in

the UML class diagram can be viewed as a directed tree, avoiding possibility of traversal loops in the instanced object graph.

One approach specifying the traversals is to use the exact propagation graph, which explicitly specifies every detail of the traversal path, i.e. where to go from every possible node of the graph. For example, if we want to retrieve all the "*Port*" atoms contained in a "*System*" model (see figure 6), we may specify *"from System to visit all its parts of Component type, then from the Component model to visit all its parts of InPortAtom and OutPortAtom type*". In this way, we specify every detail of the traversal, explicitly indicating what is the next object to be visited on every object. However as stated in the Adaptive Programming, this will introduce difficulty in maintaining and evolving the software. We want to specify the traversal in a simple and concise way so that the specification is minimally dependent on the given class graph. We should build only minimal class structure information into the specification, which will lead to more compact and reusable specifications. So according to the AP, we specify the traversal to retrieve all the "*Port*" atoms in a "*System*" model as "*form System to Port*". In this way, we only specify the start point and the end point of the traversal. We do not explicitly specify the intermediate node during the traversal, which could be computed according to the class dictionary.

In the interpreter model of the metamodeling environment, the start point of an operation (traversal) is connected to the operation atom with the "*traverseFrom*" connection. Every operation atom will connect to one and only one class reference object through the "*traverseFrom*" connection, which is represented as a solid line with arrowhead. In the "*ACMEMetaIntepreter*" model (Figure 7), the "*WriteACME*" operation

will always traverse from the "*System*" model. The "*System*" here is a reference object that refers to the "*System*" Class atom in the UML class diagram of the ACME metamodel.

There may be one or several end points in a traversal. The end points of a traversal are connected to the start point of the traversal with "*traversalTo*" connections. The "*traversalTo*" connections are also represented as solid lines with arrowhead. In the "*ACMEMetaInterpreter*", the "*WriteACME*" operation will traverse from "*System*" model down to all accessible "*InPortAtom*", "*OutPortAtom*", "*AttachementPath*", "*PropertyAtom*" and "*Role*" atoms. Here the end points of the traversal are also the reference objects that refer to the Class atoms in the UML metamodel. The traversal only walks through the aggregation and inheritance associations, which means we will only traverse inside a model to reach all its directly or indirectly accessible parts of the types specified in the traversal.

In order to specify accurate and complete traversal paths, constraints must be expressed in the specification. For example, the "*from System to PropertyAtom*" specification will result in three possible paths (see Figure 6):

1) System → Component → PropertyAtom
2) System → PropertyAtom
3) System → Connector → PropertyAtom

All these paths satisfy the specification of "*from System to PropertyAtom*". In case that we only want to retrieve the "*PropertyAtom*" objects in the "*Component*" parts of a "*System*" model, we will need to constraint the specification. The "*traverseByPassing*" and "*traverseThrough*" connections are introduced to achieve this

goal. We may constraint a traversal by using "*from A to B through X*", which indicates all the possible paths from A to B through X. The "*from System to propertyAtom through Component*" will lead to only one path in the "*WriteACME*" operation: "System → Component → PropertyAtom". The specification "*from A to B byPass X*" indicates all the possible paths form A to B that do not contain any objects of type X. The "*from System to PropertyAtom byPass Component*" will lead to two paths: "System → PropertyAtom" and "System → Connector → PropertyAtom". These two constraints are expressed in the traversal specification of the interpreter model as "*traverseThrough*" connection and "*traverseByPass*" connection. The "*traverseThrough*" connection is represented as a solid line that connects the operation atom to a reference class. The "*traverseByPass*" connection is represented as a doted line that connects the operation atom to a class referrence class. Multiple "*traverseThrough*" and "*traverseByPass*" connecitons are allowed for an operation specification. Figures 10 and 11 show an example of "*through*" and "*byPass*" constraint specification.

The base classes are allowed to be used in the traversal specification. For example the specification "from Component to Port" will result in two traversal paths: "Component → InPortAtom" and "Component → OutPortAtom", because, according to the UML metamodel of ACME, the "Port" class is the base class of the "InPortAtom" and "OutPortAtom".

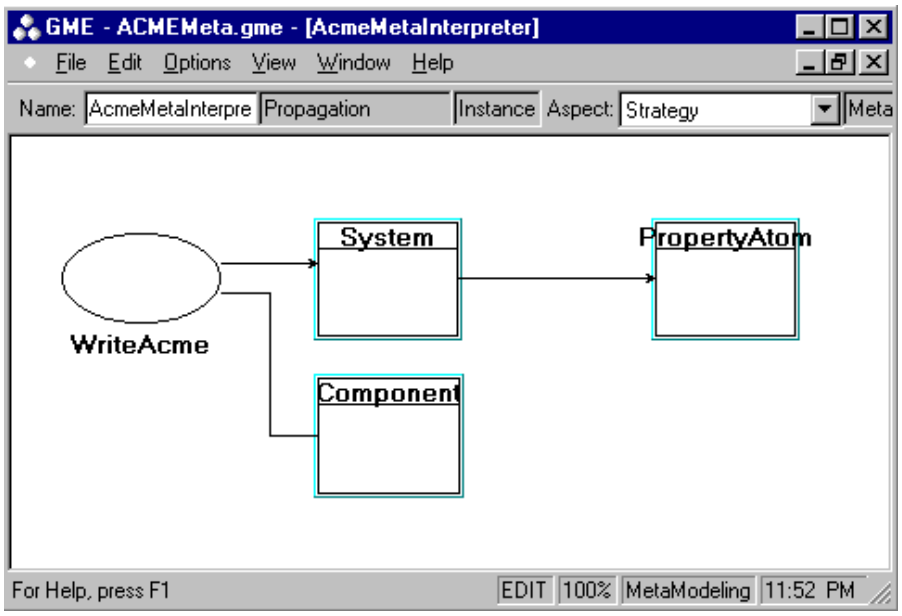

Figure 10: *from* System *to* PropertyAtom *through* Component

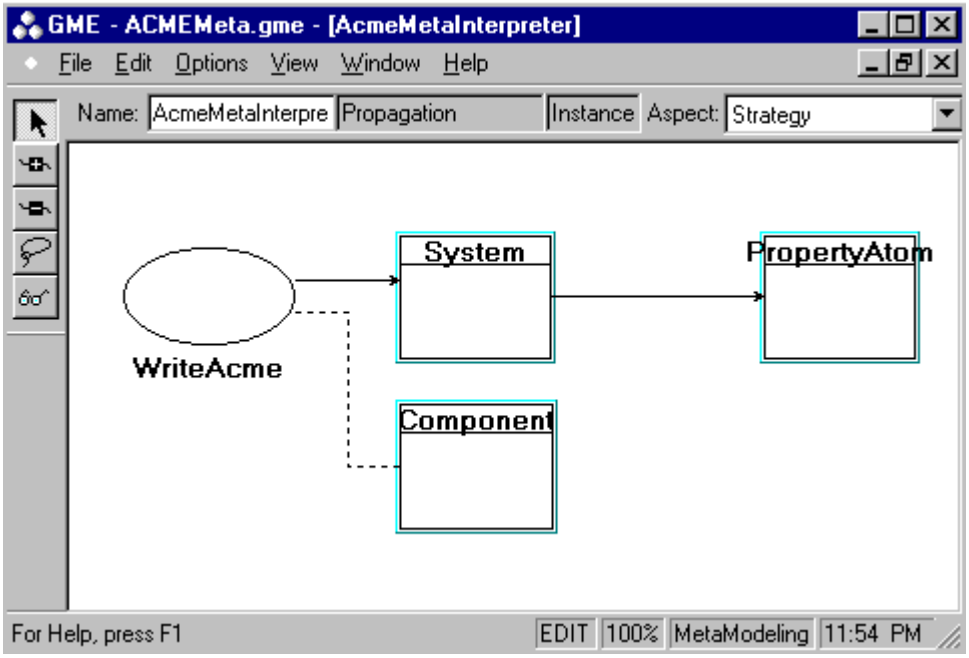

Figure 11: *from* System *to* PropertyAtom *byPass* Component

Transportation Specification

During the traversal, we may need to carry an object down to a sub-object or up to a containing super-object, so the following questions should be addressed in the transportation specification: what objects or variables need to be transported during the traversal, how should they be initialized, and what are the ranges of the transportation?

We could define the transportation of objects by simply adding extra parameters into the signature of the operation. However, if this is done, the passed objects or variables will be visible to the entire traversal path. This introduces redundancy and inefficiency, and a structure dependency problem. Again, we want only minimal class structure information to be involved to ensure good reusability.

In this case, the transportation specification is used to specify the desired objects or variables to be carried along the traversal and the range of the transportation. In the interpreter model of the metamodeling environment, an "transportation" atom is defined to indicate the desired object or variable that needs to be passed during the traversal. The "transportation" atom is represented as a small car in the interpreter model. In the ACME metainterpreter specification (Figure 7), there are three transportations defined: one "*space*" and two "*first*" variables. Similar to the traversal specification, in the transportation specification, only the start point and the end points of the transportation paths are specified. The range of the transportation will be a subgraph of the traversal graph. Two connections are defined to specify the start point and the end point of transportation: "*transFrom*" and "*transTo*" connection. Similar to the "*traversalFrom*" and "*traversalTo*" connection, the "*transFrom*" and "*transTo*" connections are represented as solid lines with arrowheads to indicate the transportation direction. Every

"*transportation*" atom must connect to one and only one class reference object in the interpreter model, with the "*transFrom*" connection indicating the start point of the transportation. A "*transportation*" atom can connect to multiple class reference objects by the "*transTo*" connection to specify the end points of the transportation. For example, the "*space*" transportation is specified to be carried from "*System*" models along all the traversal paths to the "*PropertyAtom*" atoms. This means the "*space*" variable will be available to all nodes and edges in the transportation graph defined, which are "System → Component → PropertyAtom", "System → PropertyAtom", and "System → Connector → PropertyAtom". The transportation specification lives in the context of a traversal specification, which means that the transportation graph is a sub-graph of the traversal graph. Necessary error checking or constraints should be available to enforce this rule.

Figure 12 shows the attribute definitions of the "*space*" transportation atom.

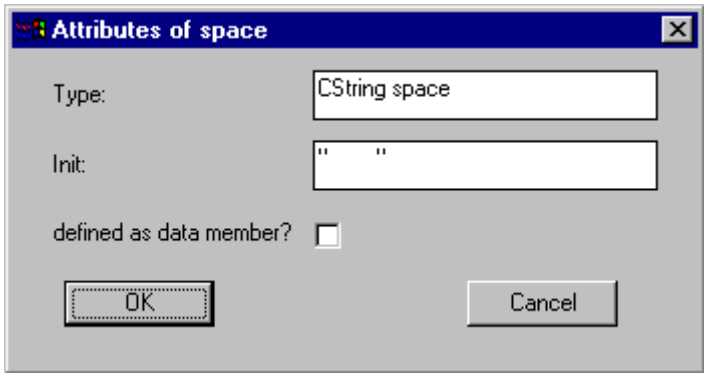

Figure 12: The Attributes of the "space" Transportation Atom

There are three attributes defined in a transportation atom: "*Type*", "*Init*" and "*defined as data member?*". The "*Type*" attribute is a field attribute that allows users to define the name and the type of the object or variable to be transported. The type could be

any C++ built-in type or user defined data type. The "*Init*" attribute supplies the possible initial values of transportations. The initial value of a transportation must match the data type specified in the "*Type*" attribute. It is designer's responsibility to guarantee the correctness of the inputs. The "*defined as data member?*" attribute specifies whether this transportation should be defined as a data member of the Traverse class, which is the implementation of the behavior specification, and is synthesized automatically from the behavior specification. I will give a detailed discussion about this later in the Code Generation section.

Action Specification

Action specifications capture what should be done when visiting a particular kind of object. Here, the action specifications will be user-defined C++ code. The action specifications are modeled as the attributes of the class reference object. Figure 13 shows the attributes of the "*Component*" reference in the "*ACMEMetaInterpreter*" model.

There are five attributes defined for a reference object in the interpreter model: "*Class Data Member*", "*before visiting*" action, "*after visiting*" action, "*before traverse children*" action and "*after traverse children*" action. The "*Class Data Member*" attribute allows designers to define necessary data members for the Class represented by the class reference. The "*before visiting*" action and "*after visiting*" action specify the actions that happen before or after visiting the object of the class type represented by the reference class. These two actions ("*before visiting*" and "*after visiting*") actually happen in the parent object of the object represented by the reference class.

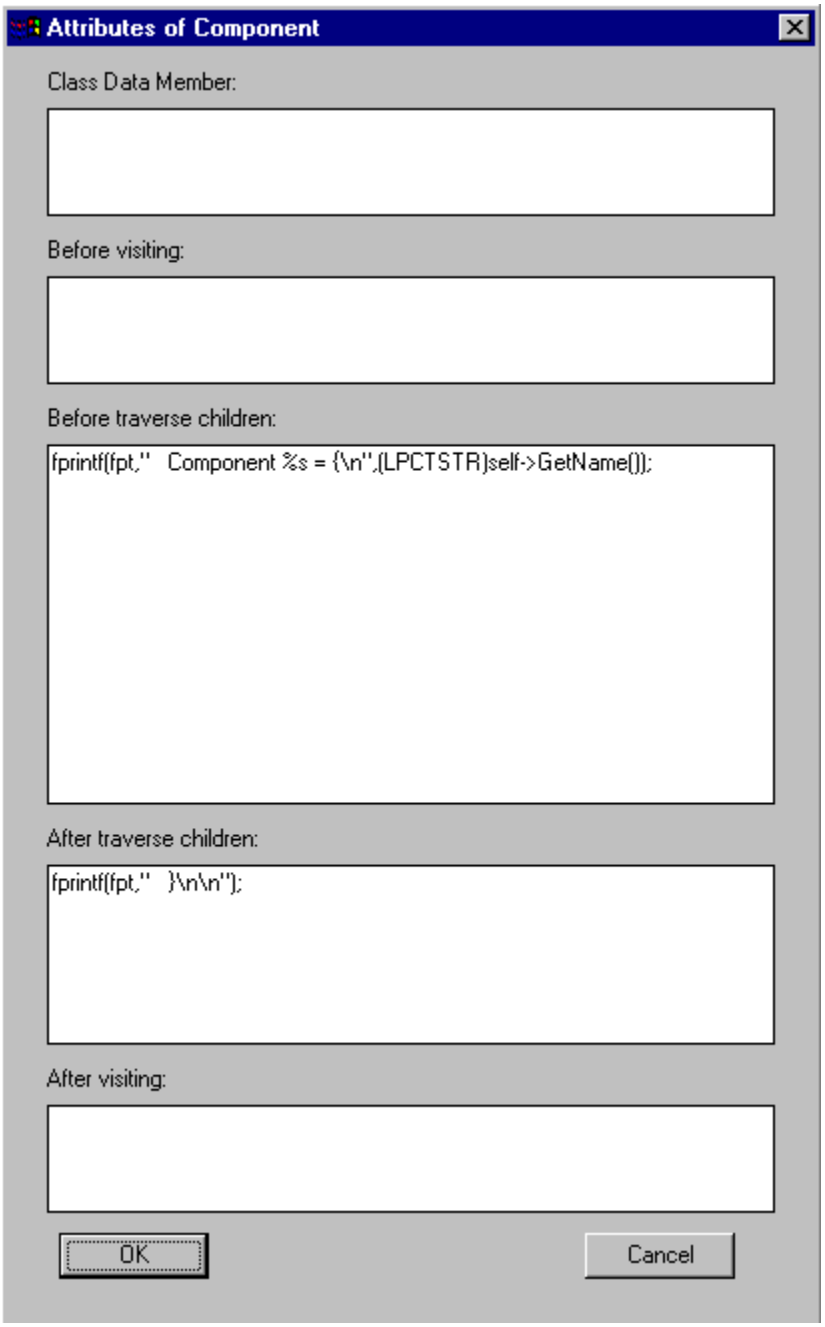

Figure 13: Attributes of the "Component" Reference Model

For example, the "*before visiting*" action of the object of "*Component*" type actually will happen in the object of "*System*" class that contains the object of "*Component*" type. The "*before traverse children*" and "*after traverse children*" actions will happen in the object of the reference class just before or after the traversing of its

parts. For example, the "*before traverse children*" action of the object of the "*Component*" type will be executed before the traversing the parts of the object. The exact position that the action will be executed will be explained and discussed in the Code Generation section.

Code Generation: Implementation of the Behavior Specification

In this thesis, a model interpreter generator has been implemented. It understands the syntax and semantic of the interpreter specification and synthesizes the model interpreter automatically from the specifications. The synthesized model interpreter is implemented in C++ code, which utilizes the high level C++ interpreter interface of GME. In this section, how the model interpreters are implemented and how we translate the model interpreter specification into the implementation are introduced.

As I stated at the beginning of this chapter, two main components constitute a model interpreter: model structure definition and model interpreter behavior definition. Model structure definition here is specified by the UML class diagram and the model behavior definition is specified in the interpreter model by the operations, traversal specifications, transportation specifications, and action specifications. According to the model interpreter specification in the metamodeling environment, six C++ source files are synthesized: "*ClassDic.cpp*", "*ClassDic.h*", "*Strategies.cpp*" and "*Strategies.h*", "*Interpreter.cpp*" and "*enum.h*". These files together implement the model interpreter. Mainly the "*ClassDic*" files implement the class definitions and the relationships between the classes. The "*Strategies*" files implement the behaviors of the model interpreter,

which include traversing the object structure, and performing related actions to generate the desired outputs.

In addition to the "*ClassDic*" files and "*Strategies*" files, an "*Interpreter.cpp*", "*Interpreter.h*", and a VC++ project are also synthesized automatically from the specification. The "*Interpreter.cpp*" file defines the entry point of a model interpreter, which triggers the model interpretation procedure.

Class Definition

In the metamodeling environment, the UML class diagram is used to specify the system structure information. That is, it specifies what classes are available in the paradigm and what are the relationships between them. The "*ClassDic.cpp*" and "*ClassDic.h*" are synthesized according to this UML class diagram. For every class atom in the GME UML metamodel, We define a class in the "*ClassDic*" files. Figure 14 shows the "*CSystemBuilder*" class definition resulted from the "*System*" class atom definition in the ACME UML metamodel.

For the "*System*" class atom in the ACME metamodel, a "*CSystemBuilder*" class is defined, which is a derived class from "*CBuilderModel*" class. As I stated before, the synthesized model interpreter is implemented by utilizing the high-level C++ interpreter interface provided by the GME, so all the classes defined in "*ClassDic*" files are derived from the "*CBuilderModel*", "*CBuilderAtom*" or "*CBuilderObject*" according to their different presentations in the GME. For example, the "*System*" class atom in the ACME UML metamodel is mapped to a model in the GME, so it is defined as a class derived from "CBuilderModel" class. The "PropertyAtom" class

atom in the ACME UML metamodel is represented as an atom in the GME, so the corresponding "*CPropertyAtomBuilder*" class will be derived from the "*CBuilderAtom*" class.

```
class CSystemBuilder;
typedef CTypedPtrList<CPtrList, CSystemBuilder*> CSystemBuilderList;

class CSystemBuilder: public CBuilderModel
{
        DECLARE_CUSTOMMODEL(CSystemBuilder,CBuilderModel)
          public:
                CPropertyAtomBuilderList* get_Property() const;
                CAttachmentAtomBuilderList* get_AttachmentPart() const;
                CPortsAndRolesConnsBuilderList* get_PortsAndRolesConns() const;
                CAttachmentPathBuilderList* get_AttachmentPath() const;
                CComponentBuilderList* get_CompPart() const;
                CConnectorBuilderList* get_ConnPart() const;
          public:
                  //member functions that wraps the action taken on this node
                virtual void WriteAcme(FILE * fpt);
};
```

Figure 14: The Class Definition of the "System" Class Atom in ACME Metamodel

Two special situations are the "*Connection*" and "*Conditional*" object in the GME. Some class atoms defined in the UML metamodel are presented as "*Connections*" or "*Conditional*" object in the GME. In this case, the corresponding class definitions are not derived from any base classes. Instead, they wrap a "*CBuilderConnection*", or a "*CBuilderConditional*" object, respectively, in their class definition. For example, the "*PortsAndRolesConns*" class atom in the UML metamodel is mapped to a Connection in the GME. The figure 15 shows the definition of the "*CPortsAndRolesConnsBuilder*" class.

We can see that there is a private pointer to the "*CBuilderConnnection*" defined in the "*CPortsAndRolesConnsBuilder*" class. This pointer provides access to the connections of "*PortsAndRolesConns*" type in the GME.

```
class CPortsAndRolesConnsBuilder
{
public:
        CPortsAndRolesConnsBuilder(CBuilderConnection* c) { conn = c;}
        ~CPortsAndRolesConnsBuilder() { delete conn;}
        CRoleBuilder * get_Src() const;
        CPortBuilder * get_Dst() const;
        CBuilderModel* GetParent() const;
private:
        CBuilderConnection* conn;
};
```

Figure 15: The Class Definition of the "PortsAndRolesConns" Class Atom

In the "*ClassDic*" files, the inheritance relationships are also defined and implemented. For example, in the ACME UML metamodel, the "*SrcRoleAtom*" and "*DstRoleAtom*" are derived from the "*Role*" class (Figure 16). Figure 17 shows the corresponding class definition of these classes. The "*CSrcRoleBuilder*" and "*CDstRoleBuilder*" are derived from the "*CRoleBuilder*" class.

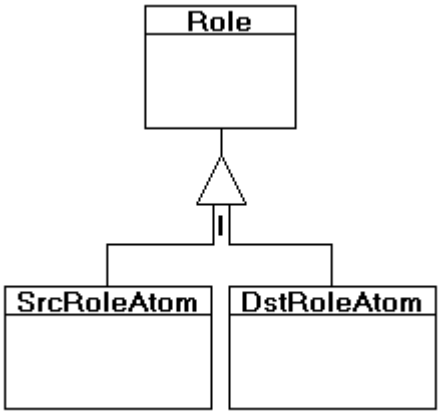

Figure 16: The Inheritance Relationship in the ACME UML Metamodel.

```
class CRoleBuilder: public CBuilderAtom
{
        DECLARE_CUSTOMATOM(CRoleBuilder,CBuilderAtom)
public:
};

class CDstRoleAtomBuilder: public CRoleBuilder
{
        DECLARE_CUSTOMATOM(CDstRoleAtomBuilder,CRoleBuilder)
public:
//member functions that wraps the action taken on this node
        virtual void WriteAcme(FILE * fpt,bool & first);
};

class CSrcRoleAtomBuilder: public CRoleBuilder
{
        DECLARE_CUSTOMATOM(CSrcRoleAtomBuilder,CRoleBuilder)
public:
//member functions that wraps the action taken on this node
        virtual void WriteAcme(FILE * fpt,bool & first);
};
```

Figure 17: The Class Definitions of the "Role" and "SrcRole" and "DstRole" Classes

The final part to be explained is the data member and member function definitions in the synthesized classes. In the class definition, a set of class member functions is defined to retrieve the available parts of the object. For example, in the "*CSystemBuilder*" class definition, the "*CPropertyAtomBuilderList* get_Property() const*" member function are defined and implemented to retrieve the parts of "*Property*" type. This function will return a list of the parts of "*Property*" type in the "*System*" object. Also a set of member functions are defined to directly retrieve the values of the object attributes. In addition to these functions, a set of virtual functions are defined and implemented in the class definition. For example, the "*virtual void WriteAcme(FILE * fpt)*" function in the "*CSystemBuilder*" class. These virtual functions are the trigger points for a traversal starting at the current object. Every virtual function corresponds to an operation definition in the interpreter model of the metamodeling environment. For example, the

"*virtual void WriteAcme(FILE * fpt)*" function is synthesized from the "*WriteAcme*" operation atom in the "*ACMEMetaInterpreter*" model. These virtual functions will active traversals that start at the current object and traverse down along the path specified in the traversal specification. Figure 18 shows the implementation of the "*WriteAcme*" function of the "*CSystemBuilder*" class.

```
void CSystemBuilder::WriteAcme(FILE * fpt)
{
        WriteAcme_T trv;
        WriteAcme_V * vis= trv.get_Visitor();
        vis->visit(this,fpt);
}
```

Figure 18: The Implementation of the "CSystemBuilder::WriteAcme" Member Function

We can see that the "*WriteAcme*" function will create a pair of objects of the "*WriteAcme_T*" traversal class and "*WriteAcme_V*" visitor class, and call the "visit" function of the visitor object by passing itself as a parameter. Detailed discussion will be given in next section about the traversal and visitor classes, which are the implementations of the behavior specification of the model interpreter.

The data members of the class definition come from the behavior specification in the metamodeling environment. As I explained in the behavior specification section, there is a "*Class Data Member*" attribute defined in the class reference object of the interpreter model (see Figure 13). This "*Class Data Member*" attribute allows the designer to define the class data members for the synthesized class. The defined class data members could be used to help the implementation of the behavior of the model interpreter, like passing messages or storing information.

Strategy (Traversal/Visitor Classes) Definition

We have seen how the class definitions are synthesized from themetamodeling specification. In this section, I will explain how the model interpreter behaviors are implemented and synthesized from the behavior specification of the interpreter model in the metamodeling environment.

As I stated before, the behavior specifications are grouped by operations. Every operation atom in the interpreter model groups a set of traversal specifications, transportation specifications, and action specifications. It specifies a function or method of the model interpreter. For every operation atom, a pair of Traversal and Visitor classes are synthesized, which implement the functionality specified by the operation atom.

The Traversal class captures how the models should be traversed. It addresses the question: "If we are at node of type X, which node do we go to next?" The Traversal class encapsulates the traversal code fragments (in C++) that navigate through a group of related objects in the specified order to accomplish some task. The Visitor class captures the actions to be taken when visiting a node of a particular type. They encapsulate the C++ code fragments that perform desired operations and provide a context for the traversal. The Traversal and Visitor classes are directly linked to each other, and are operated in a co-routine like manner [10]. Suppose the Traversal starts at a specific type of model node. Based on its specification, it determines how to follow pointers emanating from that type of node, and call the Visitor on the objects that the pointers are pointing to. The Visitor might take an action, and/or active the Traversal to proceed from the accessed node. So the control flow oscillates between the Traversal and the Visitor: the

Traversal determines where to go next, the Visitor "visits" (i.e. takes actions) and calls back the Traversal to proceed further [10].

The model interpreter generator will translate each traversal specification, transportation specification, and action specification grouped by an operation atom into a pair of Traversal and Visitor classes. The Traversal class and Visitor classes are tied together via reference pointers to each other. Figure 19 shows the Traversal class definition and the Visitor class definition for the "*WriteAcme*" operation of the "*ACMEMetaInterpreter*" model. The Traversal class is called "*WriteAcme_T*", and the Visitor class is named as "*WriteAcme_V*". We can see that the "*WriteAcme_T*" class keeps a pointer to the "*WriteAcme_V*" class, and the "*WriteAcme_V*" class keeps a pointer to the "*WriteAcme_T*" class.

A set of traverse and visit functions is defined in the Traversal and Visitor class respectively. The Traversal and Visitor classes together guide a traversal through an object graph in the specific order and perform specific operations on the nodes along the traversal to accomplish some task. In this case, a visit function is defined for every type of object in the traversal object graph and a traverse function is defined for every type of model in traversal object graph. The first parameter passed to both the traverse function and visit function is a pointer to the type of the model/atom object where this traverse/visit function will be called. The rest of the parameters are the "*Signature*" attribute definition of the operation atom and possible transportation specifications. For example in the "*void traverse(CSystemBuilder* self,FILE * fpt,CString space)*" function, the first parameter "*self*" is a pointer to "*CSystemBuilder*", the "*FILE* fpt*" parameter comes

from the "*Signature*" attribute of "*WriteAcme*" operation atom, and the "*Cstring space*" parameter comes from the "*space*" transportation.

```
class WriteAcme_V;
class WriteAcme_T {
public:
        WriteAcme_T();
        ~WriteAcme_T() {}
        void traverse(CConnectorBuilder* self,FILE * fpt,CString space,bool & first);
        void traverse(CComponentBuilder* self,FILE * fpt,CString space,bool & first);
        void traverse(CSystemBuilder* self,FILE * fpt,CString space);
        WriteAcme_V* get_Visitor() { return vis;}
private:
        WriteAcme_V* vis;
};
class WriteAcme_V {
public:
        WriteAcme_V(WriteAcme_T* _t);
        ~WriteAcme_V() {}
        void visit( CConnectorBuilder* self,FILE * fpt,CString space);
        void visit( COutPortAtomBuilder* self,FILE * fpt,bool & first);
        void visit( CInPortAtomBuilder* self,FILE * fpt,bool & first);
        void visit( CComponentBuilder* self,FILE * fpt,CString space);
        void visit( CDstRoleAtomBuilder* self,FILE * fpt,bool & first);
        void visit( CPropertyAtomBuilder* self,FILE * fpt,CString space);
        void visit( CSrcRoleAtomBuilder* self,FILE * fpt,bool & first);
        void visit( CAttachmentPathBuilder* self,FILE * fpt);
        void visit( CSystemBuilder* self,FILE * fpt);
private:
        WriteAcme_T* trv;
};
```

Figure 19: The Traversal and Visitor Class Definition for the "WriteAcme" Operation of the "ACMEMetaInterpreter"

Figure 20 shows a typical implementation of a traverse function of a Traversal class and a visit function of a Visitor class. In the visit function, the <before visiting actions> and <after visiting actions> come from the "*Before visiting*" and "*After visiting*" attributes of the class reference object (in action specification). If the current node is not the end node of the traversal, the visit function will also call the traverse function of the corresponding Traversal class. In the traverse function, the available parts of the current node are retrieved in the order specified in the "*Sequence to visit parts*" attribute of the

operation atom. Then the corresponding visit functions are called on the parts (the control flow back to Visitor class). In the traverse function, the <before traversal parts action> and <after traversal parts action> come from the "*Before traverse children*" and "*After traverse children*" attributes of the class reference object (in action specification). These user-defined codes are executed in the specific order at the specific positions. The Visitor class may also have data members that come from the transportation specifications when the transportations are specified to be defined as data members.

```
void SomeVisitor::visit(CurrentObject * self, parameters ){
        <Before visiting actions>
        trv->traverse(self, parameters);
        <after visiting actions>
}

void SomeTraversal::traverse( CurrentObject* self,signatures, transportations){
        <before traversing parts1 actions>
        Part1List* part1=self->get_Part1();
        while( there is entry in the part1List){
                Part1* arg = Part1->GetNext();
                vis->visit(arg,singature, transportations);
        }
        <after traversing parts1 actions>


        <before traversing parts2 actions>
        Part2List* part2=self->get_Part2();
        while( there is entry in the part2 list){
                Part2* arg = part2->GetNext();
                vis->visit(arg,singature, transportations);
        }
        <after traversing parts2 actions>
                :
                :
                :

        <before traversing partsN actions>
        PartNList* partN=self->get_PartN();
        while( there is entry in the partN list){
                PartN* arg = partN->GetNext();
                vis->visit(arg,singature, transportations);
        }
        <after traversing partsN actions>
}
```

Figure 20: A Typical Implementation of a Traverse and Visit Function

The Visitor and Traversal classes are defined in the "*Strategy.cpp*" and "*Strategy.h*" files.

CInterpreter Class Definition

The "*CInterpreter*" class is a class standardized in the GME high-level C++ interpreter interface. It provides an "*interpret*" member function that is the entry point to the interpretation process. The implementation of the "*interpret*" function comes from the "*Interpret*" attribute of the interpreter model in the metamodeling environment. The "*Header*" attribute of the interpreter model will contribute the header definition of the "*CInterpreter.cpp*" file.

Visual C++ Project

The model interpreter generator also generates the Visual C++ workspace project for the model interpreter. The name of the interpreter model will be the name of the project. In the project, all necessary resource files, make files and source files are generated. It can be loaded by the Microsoft Visual C++ developing environment and built directly.

In addition to the project, "*ClassDic*" and "*Strategy*" files, an "*enum.h*" file is also synthesized. It defines the menu attributes specified in the metamodeling environment as C++ "enum" types.

CHAPTER IV

EXAMPLE

In addition to the ACME metamodeling environment, two other metamodeling environments (ORMS and SignalFlow) have been created with interpreter metamodels. In this chapter, I will use the ORMS metamodeling environment as an example to demonstrate the full range and capability of the metamodeling environment for the model interpreter.

The ORMS (Outage Restoration Management System) is a diagnosis engine developed by the Institute for Software Integrated System at Vanderbilt University for Joe Wheeler Electrical Membership Corporation, a small electric utility company in Trinity, AL [21]. The ORMS first gathers the electrical circuit information stored in various databases, such as a Geographic Information System (GIS), an IVR (Interactive Voice Recognition) system and a CIS (Customer Information System). Then it executes the diagnosis algorithm to find the possible outages and their locations in the electrical network. Because the real data of the Joe Wheeler electrical network is extremely huge, it is difficult to simulate, test and debug the ORMS diagnosis algorithm using the real data. In this situation, we built an ORMS utility modeling environment. The ORMS modeling environment allows us to build electrical utility network models of reasonable size and to modify the data flexibly in the GME. Then we can use the ORMS model interpreter to translate the GME utility network models into the ORMS diagnosis engine, and test the diagnosis engine on the utility network models built in the GME. The interpreter also

provides the ability to allow user interactions with the ORMS. Thus, we can simulate the outages in the electric network using the models created in the GME, and test and debug the ORMS diagnosis engine conveniently.

For this reason, we created an ORMS metamodel and synthesized the ORMS modeling environment, and the ORMS model interpreter. The UML diagram of the ORMS metamodeling environment is shown in Figure 21.

To represent an electrical network, we define a utility model that consists of various components. The components are divided into unidirectional components and bi-directional components. Every component has two types of terminals: "*InputTerminals*" and "*OutputTerminals*". Components are connected through a "*Conn*" connection. All "*SubStation*", "*CB*", "*MS*", "*Transformer*", "*Load*", "*Line*" and "*Switch*" component classes are derived from the "*Component*" class directly or indirectly. They have some common attributes. However, they may also have their own attributes, which are defined in the GME category of the metamodeling environment and are not shown here.

Based on the UML metamodel and the GME presentation specification of the metamodeling environment, we can synthesize the paradigm of the ORMS modeling environment and then use the ORMS paradigm to build utility models to represent the electrical networks. Figure 22 shows a GME model for a simple electrical network with its electrical components created by using the ORMS paradigm.

The task of the ORMS model interpreter is to build the object network in the ORMS diagnosis engine according to the electrical network models created in the GME. Basically, the interpreter will traverse through the electrical network represented by the Utility model. For every component within the utility model it creates a corresponding

component object in the ORMS run-time engine. If the two components are connected in the utility model of GME, it connects the corresponding components in the ORMS.

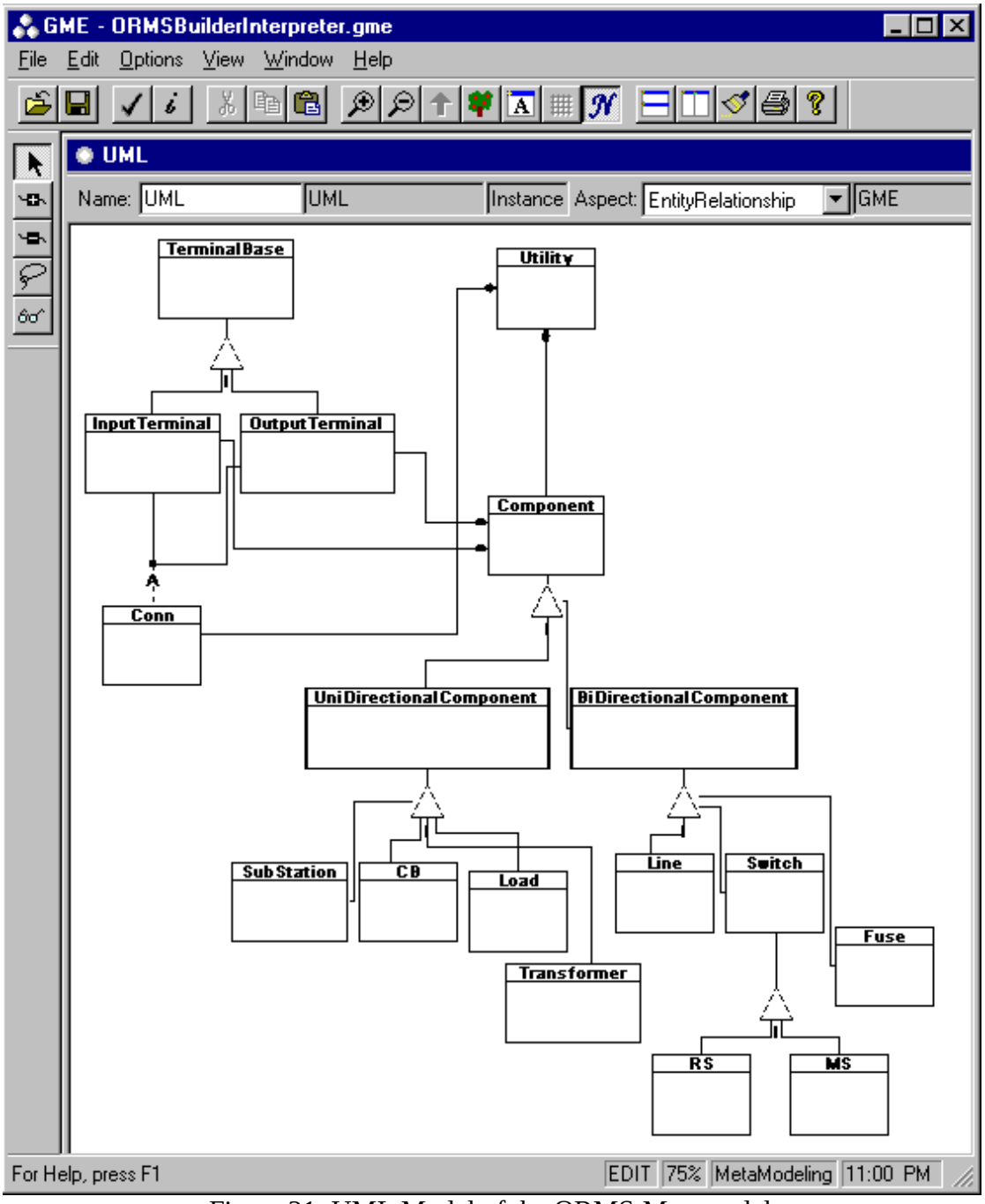

Figure 21: UML Model of the ORMS Metamodel

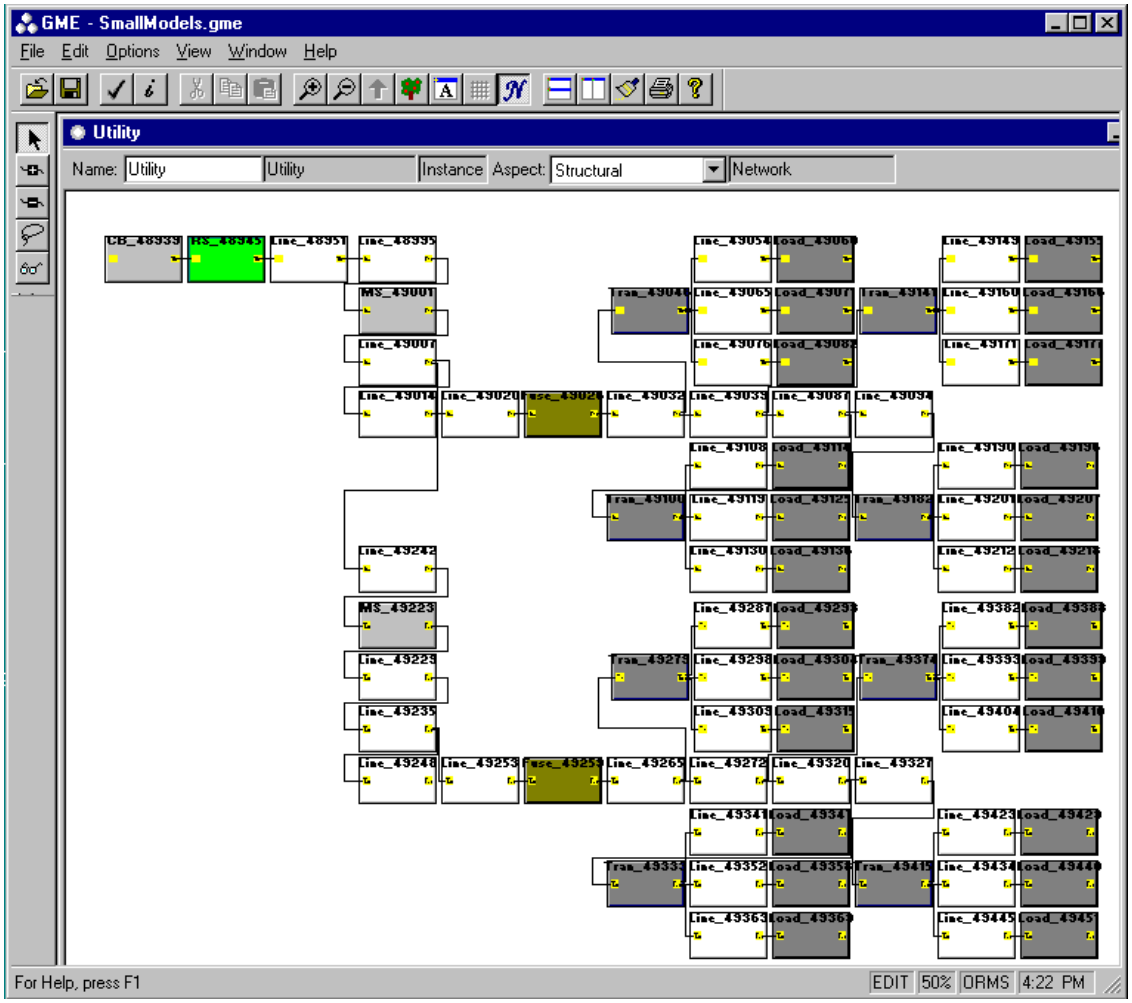

Figure 22: An Example Model of the ORMS Electrical Utility Network

Figure 23 shows the "*ORMSBuilder*" interpreter model of the metamodeling environment that specifies the ORMS model interpreter.

In the "*ORMSBuilder*" interpreter model, an operation called "*BuildNetWork*" is created, which groups a set of traversal, transportation and action specifications, and specifies the behaviors of the ORMS model interpreter. The traversal specification says that the traversal will start from a Utility model and traverse down to all the directly and

indirectly accessible terminal atoms and *"Conn"* connections contained in the Utility model.

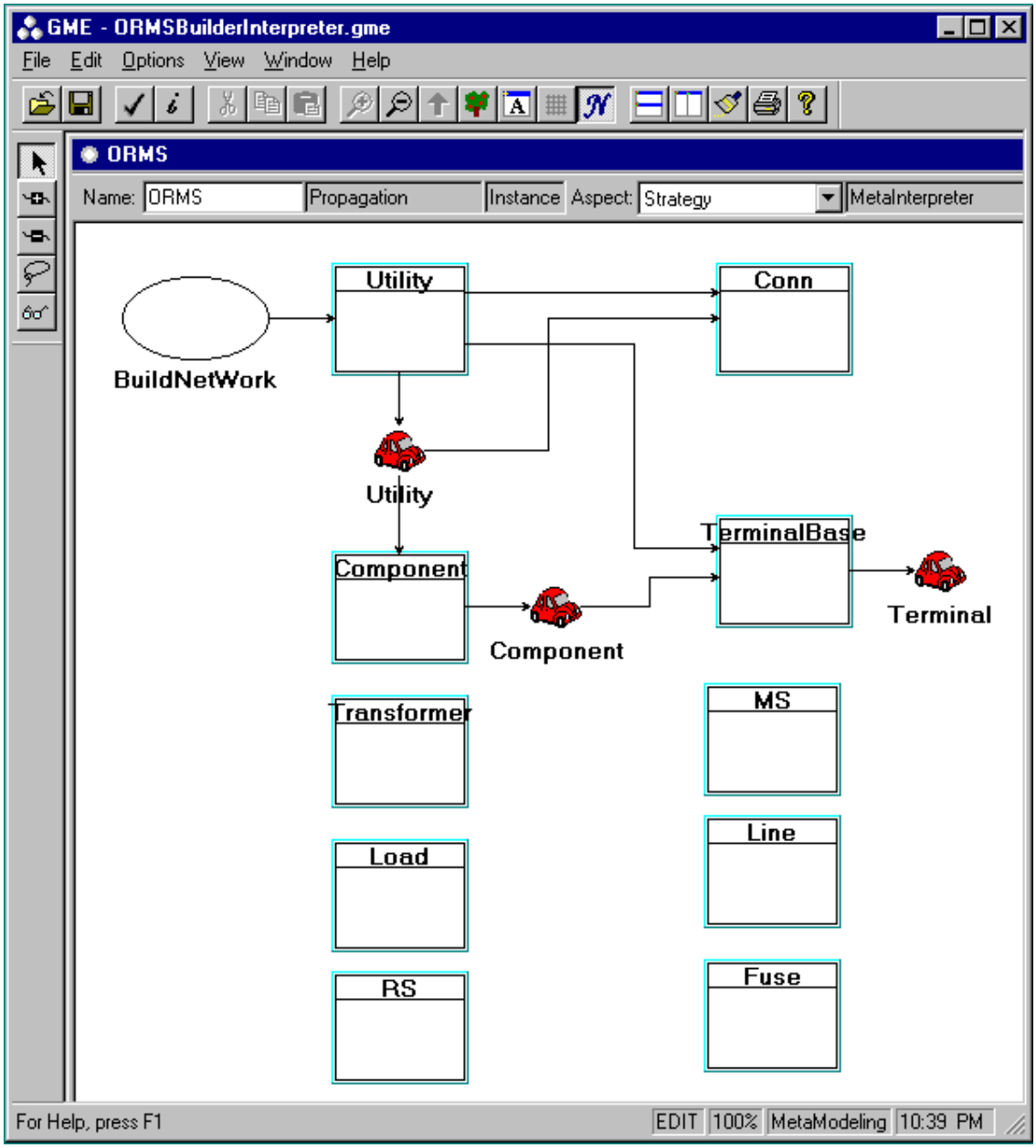

Figure 23: The ORMSBuilder Interpreter Model of ORMS Metamodeling Environment

During the traversal, three transportations ("*Utility*", "*Component*" and "*Terminal*" ORMS objects) are defined. The "*Utility*" ORMS object is passed from "*Utility*" model to all accessible "*Component*" models. The "*Component*" ORMS object is passed from all "*Component*" model to all accessible "*Terminal*" atoms. There is no end point specified for the "*Terminal*" transportation, since the "*Terminal*" transportation here is used to only define a "*Terminal*" data member for the corresponding Traverse class synthesized from the traversal specification.

Figure 24 shows the attributes of the "*Terminal*" transportation. We can see that the "*Terminal*" transportation is specified to be defined as a data member.

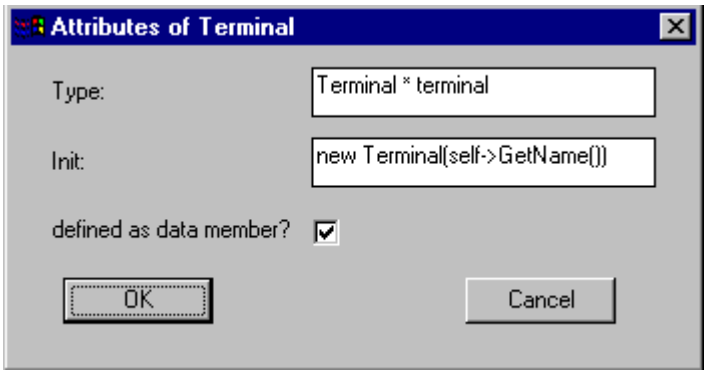

Figure 24: Attributes of "Terminal" Transportation.

As I explained before, the corresponding actions to be executed at the nodes along the traversal path are specified as the attribute of the class reference objects. Figure 25 shows the actions need to be taken in all "*Component*" models. In the "*Component*" model, corresponding ORMS component objects are created according the "*Component*" type (MS, Transformer, RS, Load, etc.). Also, the various specific actions that need to be taken for different components are specified in the attributes of the specific component reference classes. Figure 26 shows the actions need to be taken for the "*Load*" model. Because the "*Load*" model is derived from the "*Component*" model, the actions specified

for the "*Component*" model will also be executed in the "*Load*" model. However, the action specified in the "*Load*" model will only be executed in the "*Load*" model. In the current metamodeling environment, the actions specified in the based class will be executed before the actions specified in the derived class. This may not be true for all model interpreters. More accurate specification should and will be used in the future to address this problem. Based on the "*ORMSBuilder*" interpreter specification in the ORMS metamodeling environment, the ORMS model interpreter has been synthesized and successfully used in the project.

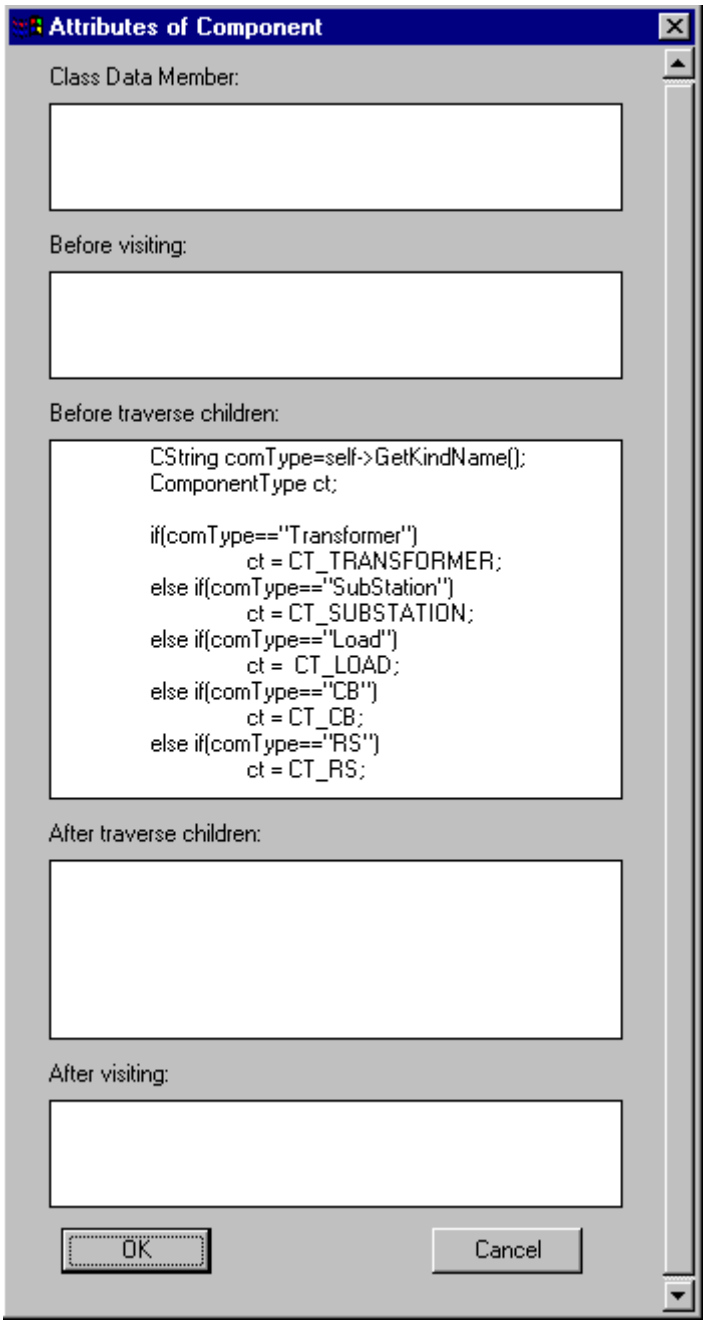

Figure 25: Action Specification of the Component Model

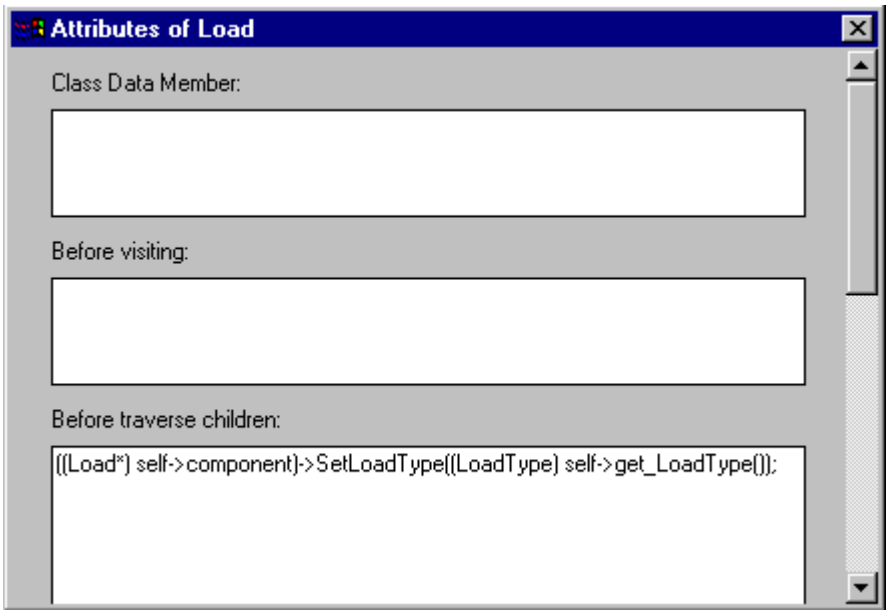

Figure 26: Action Specification for Load Model.

CHAPTER V

CONCLUSIONS

The metamodeling environment for the model interpreter studied in this thesis allows the specification and synthesis of model interpreters for the GME-based modeling environment. It is integrated with the existing UML/GME metamodeling environment for the Domain Specific Modeling Environment (DSME). They together allow the entire DSME to be modeled and synthesized, and further allow the complete evolution of the DSME.

By properly separating the model structure specification and the model interpreter behavior specification, the approach studied in this thesis enhances the ability to evolve and maintain model interpreters. When the system requirements change, changes can be made to the model interpreter specification, and the new model interpreter can be correctly resynthesized from the new specification. This reduces the cost and effort to develop and evolve model interpreters.

Although this thesis research has led to several advances in DSME specification and synthesis capabilities, limitations still exist:

1) In order to avoid the circular traversal path, the traversal specification can only follow the inheritance and aggregation relationships in the model structure. Hence, this sometimes leads to complex specification and more effort in specifying access to other relationships existed in the modeling environment. For example, the reference relationships can not be specified

directly in the traversal specification, because it is not accessible through aggregation or inheritance relationships in the GME UML metamodel.

2) Although most of the model interpreter behaviors can be modeled and specified in the metamodeling environment, not all the model interpreter behaviors are easy to be specified by the behavior specification studied in the thesis. Sometimes in order to specify a general simple behavior of a model interpreter (like, generating text files), many operations and traversal specifications are required. It is desired to have the ability to allow user to write these trivial behaviors and integrate them into the synthesized model interpreter.

3) Now, the synthesized model interpreter is based on the high-level C++ interpreter interface provided by the GME. If the high-level C++ interpreter interface changes in the future, the metaintepreter modeling environment does not have the ability to evolve to meet the change of the interpreter interface. It is desired to build a modeling environment to synthesize the metaintepreter modeling environment itself, and thus allow the metamodeling environments to evolve themselves.

ELECTRICAL ENGINEERING

VISUAL SPECIFICATION OF MODEL INTERPRETERS

JIANFENG WANG

Thesis under the direction of Professor Gabor Karsai

Model Integrated Computing (MIC) has been accepted and proven as an efficient and effective technology for developing, evolving and maintaining the large computer-based systems (CBSs), where functional, performance, and reliability requirements demand the tight integration of physical processes and information processing [1]. MIC is a model-based approach to software development. It allows designers to create models of domain-specific systems using model-integrated program synthesis environments (MIPS), and synthesize application program from these models. The MultiGraph Architecture (MGA), under development at the Institute for Software Integrated Systems at Vanderbilt University, is a toolkit for creating such MIPS systems. The MGA has been used as a basis for developing a wide variety of engineering-based MIPS environments.

The model interpreter plays a key role in MIC. It is the bridge between the domain-specific modeling environment (DSME) and the real application domain. The model interpreter traverses the models and their parts, examines the relationships between the models, transforms the necessary information in the models into the languages used by tools, or executable specifications used to automatically synthesize software. MIC relies on the interpreters to translate the information specified in the DSME, domain specific models into various desired outputs for the application domain. Writing a model interpreter is a non-trivial task. The designer or programmer needs to study and

understand the structure of domain specific models, the exact output required of the interpreter, and the relationship or mapping between these two. The designer also needs to implement the translation between the DSME and the output, which involves lots of repetitive and error prone programming. Instead of hand writing these interpreters, the entire DSME should be modeled itself, and generated from these metamodels. This would allow the complete MGA design environment to evolve in the case of changing domain requirements. Model interpreters need to be specified and synthesized from the metamodeling environment of MGA.

Analysis shows that it is possible and desirable to model both the syntactic and semantic behavior of model interpreters and to synthesize model interpreters automatically. This leads to a completed metamodeling environment and allows both the domain-specific applications and the DSME itself designed to evolve.

In this thesis, I present an approach to specify and synthesize domain-specified model interpreters for MGA. The metamodeling environment will allow DSME designers to create visual specifications of the model interpreter for a specific domain, and the domain-specific model interpreters will be synthesized automatically from the metamodeling environment. This completes the functionality of the metamodeling environment to specify and synthesize the entire domain-specific MIPS environment, and reduces the risk and cost of MIC system development.

Approved _____ Date _____