MARS: A Metamodel Recovery System Using Grammar Inference

FAIZAN JAVED[1], MARJAN MERNIK[2], JING ZHANG[1], JEFF GRAY[1], AND BARRETT R. BRYANT[1]

[1]Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL, USA 35294-1170
Phone: 1-205-934-2213
[javedf, zhangj, gray, bryant]@cis.uab.edu

[2]Faculty of Electrical Engineering and Computer Science
University of Maribor
2000 Maribor, Slovenia
Phone: 386-2-220-7455
marjan.mernik@uni-mb.si

Abstract

Domain-specific modeling (DSM) assists subject matter experts in describing the essential characteristics of a problem in their domain. Various software artifacts can be generated from the defined models, including source code, design documentation, and simulation scripts. The typical approach to DSM involves the construction of a metamodel, from which instances are defined that represent specific configurations of the metamodel entities. Throughout the evolution of a metamodel, repositories of instance models can become orphaned from their defining metamodel. In such a case, instance models that contain important design knowledge cannot be loaded into the modeling tool due to the version changes that have occurred to the metamodel. Within the purview of model-driven software engineering, the ability to recover the design knowledge in a repository of legacy models is needed. A correspondence exists between the domain models that can be instantiated from a metamodel, and the set of programs that can be described by a grammar. In this paper, we propose MARS, a semi-automatic inference-based system for recovering a metamodel that correctly defines the mined instance models through application of grammar inference algorithms. The paper contains a case study that demonstrates the application of MARS, as well as experimental results from the recovery of several metamodels in diverse domains.

Key Terms – *Metamodeling, grammar inference, generative programming, program transformation*

1. Introduction

During the various phases of the software development process, numerous artifacts may be created (e.g., documentation, models, source code, testing scripts) and stored in a repository. Some of the artifacts involved in development, such as source code and models, depend on a language schema definition that provides the context for syntactic structure and semantic meaning. Over time, evolution of the schema definition is often required to address new feature requests. If the repository artifacts are not transformed to conform to the new schema definition, it is possible that the repository becomes stale with obsolete artifacts. For example, new versions of the Java JDK often restructured and renamed parts of the API, which resulted in legacy source code dependent on deprecated features. Similarly, evolution of a metamodel may prompt requisite adaptations to instance models to bring them up to date. This paper offers a contribution toward recovering the metamodel schema definition for domain-specific models that have been separated from their original defining metamodel.

1.1. Domain-Specific Modeling

Throughout the history of programming languages, abstraction was improved through evolution towards higher levels of specification. Domain-specific modeling (DSM) has adopted a different approach by raising the level of abstraction, while at the same time narrowing the design space to a single domain of discourse with visual models [14]. When applying DSM, models are constructed that follow the domain abstractions and semantics, allowing developers to perceive themselves as working directly with domain concepts. The DSM language captures the semantics of the domain and the production rules of the instantiation environment.
An essential characteristic of DSM is the ability to generate, or synthesize, new software artifacts from domain models. This is typically accomplished by associating a model interpreter with a particular modeling language. A model interpreter transforms the concept structures into physical implementations in code. With the generative [9] approach available in DSM, there is no longer a need to make error-prone mappings from domain concepts to design concepts, and on to programming language concepts. Example domains where DSM has been applied successfully are the Saturn automotive factory [23], DuPont chemical factory [12], numerous government projects supported by DARPA and NSF [1], electrical utilities [28], and even courseware authoring support for educators [16].

An important activity in DSM is the construction of a metamodel that defines the key elements of the domain. Instances of the metamodel can be created to define specific configurations of the domain. An example is shown in Figure 1.1 that represents the metamodel for a simple modeling language for specifying properties of a network. The metamodel contains networking concepts (e.g., routers, hosts, and ports) as well as the valid connections among all entities. Two instances of this metamodel are shown in Figure 1.2, which illustrates different company configurations that are connected on the network. The domain-specific nature of this model is evident from the icons and visualization of domain representations. The networking example presented in this paper is taken from one of the samples provided with the Generic Modeling Environment (GME) (available at http://www.isis.vanderbilt.edu, and summarized in Section 2.2). The metamodel of Figure 1.1 is the target goal of the paper with individual instances driving the inference process.

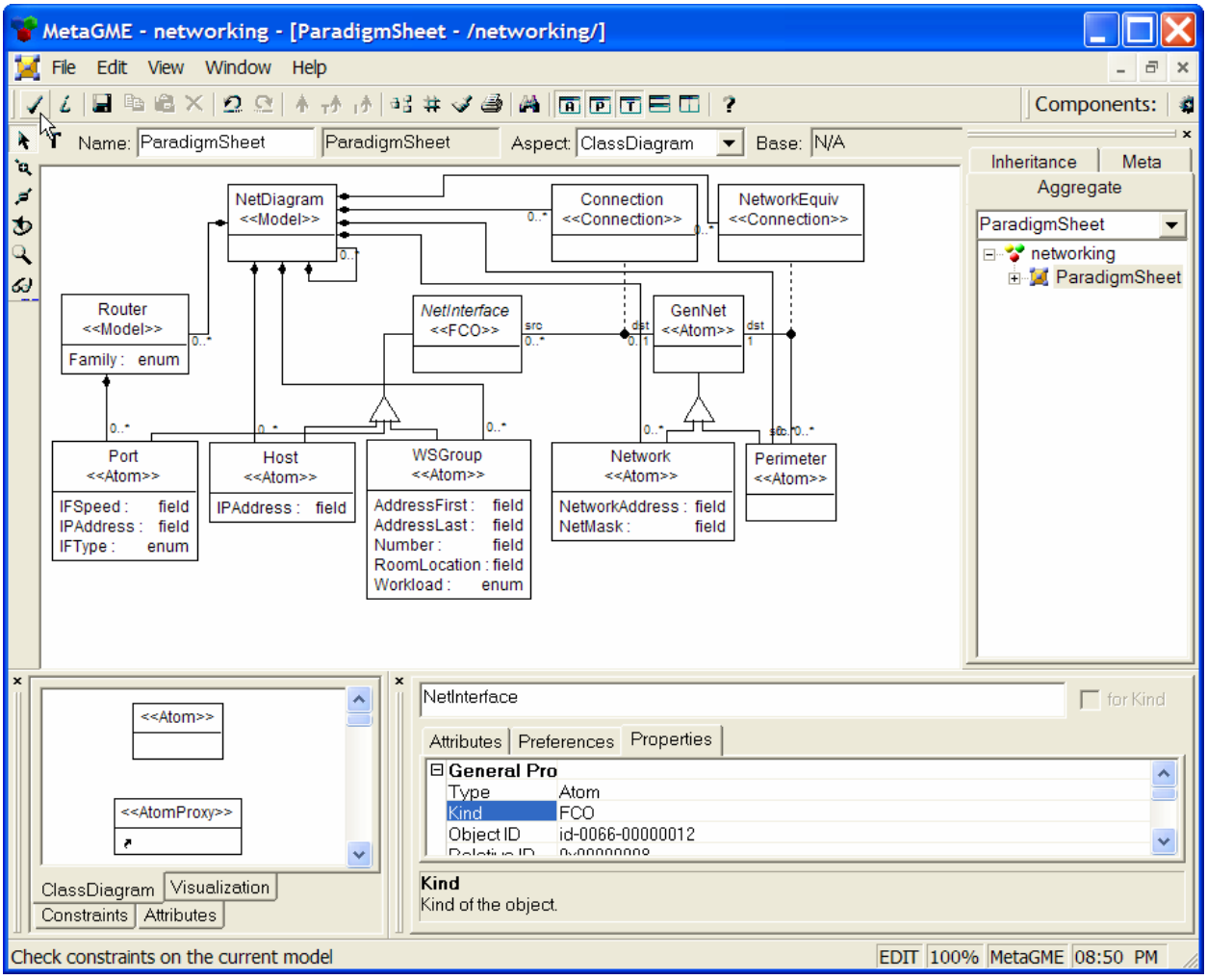


Figure 1.1: A metamodel for creating network diagrams

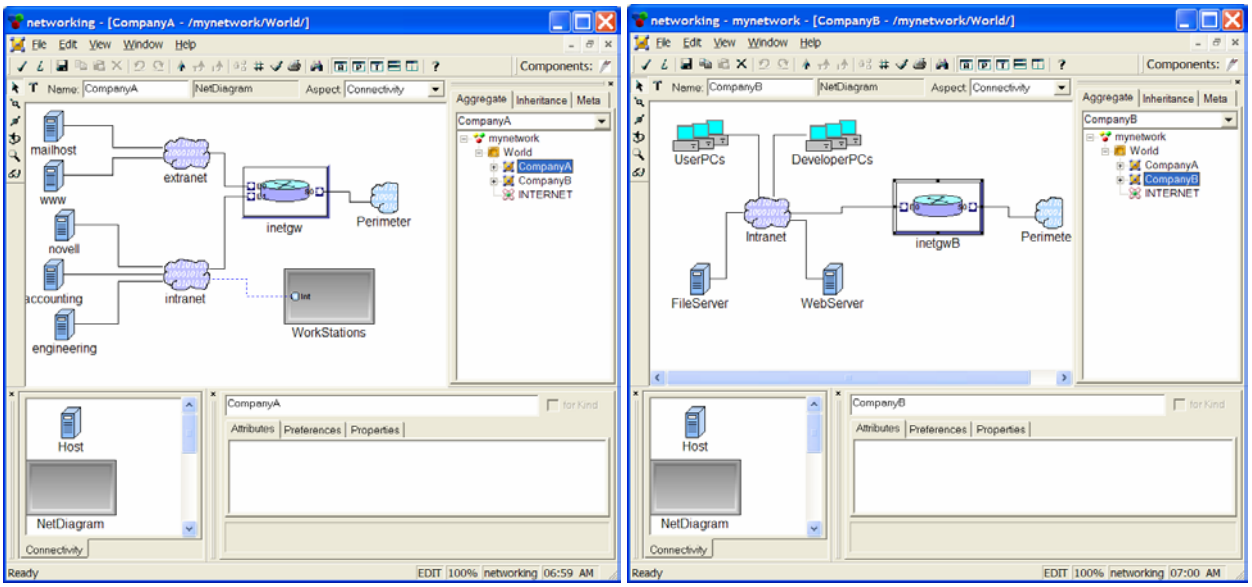


Figure 1.2: Two instances of a network

1.2 Challenges of Mining Domain Models to Infer a Metamodel

From our experience in applying DSM, it is often the case that a metamodel undergoes frequent evolution, which may result in previous instances being orphaned from the new definition. We call this phenomenon *metamodel drift*. When the metamodel is no longer available for an instance model, the instance model may not be loaded into the modeling tool. However, if a metamodel can be inferred from a mined set of instance models, the design knowledge contained in the instance models can be recovered. There are several challenges involved in mining a set of instance models in order to recover the metamodel. The three key challenges, and the solutions presented in this paper, are as follows:

1. *Inference Algorithms for Mined Models*: The research literature in the modeling community has not addressed the issue of recovering a metamodel from a set of instance models. However, a rich body of work has been reported in the grammar inference community to infer a defining grammar for a programming language from a repository of example programs (see Section 6 for a summary of grammar inference research). A key contribution of this paper is an adaptation of grammar inference algorithms to the metamodel inference problem.
2. *Model Representation Problem*: Most modeling tools provide a capability to export a model as an XML file. However, there is a mismatch between the XML representation of

a model, and the syntax expected by the grammar inference tools. To mitigate the effect of this mismatch, the technique presented in this paper translates the XML representation of an instance model into a textual domain-specific language (DSL) that can be analyzed by traditional grammar inference tools.

3. *Mining Additional Model Repository Artifacts*: In addition to the instance models, there are other artifacts that can be mined in the modeling repository. For example, the model interpreters contain type information that cannot be inferred from the instance models. The key challenge with mining information from a model interpreter is the difficulty of parsing the model interpreter source (e.g., a complex C++ program), and performing the appropriate analysis to determine the type information. To address this problem, the paper describes a technique that uses a program transformation system to parse the model interpreter code and recover the type information of metamodel entities.

1.3 Overview of Paper Sections

An overview of the MetAmodel Recovery System (MARS) is illustrated in Figure 1.3. The activities contained in the box represent the key generators and intermediate results of the inference process. The three artifacts to the right of the box correspond to the required inputs and final output of the process. The metamodel inference process begins with the translation of various instance models into a DSL that filters the accidental complexities of the XML representation in order to capture the essence of the instance model (represented as step 1 in Figure 1.3). The second step shown in Figure 1.3 represents the mining of name and type information from model interpreters associated with the instance models. A program transformation tool, the Design Maintenance System (DMS) [3], is utilized to ascertain the appropriate type information for attributes defined in the metamodel (note: there is a distinction between the acronyms DSM and DMS). The intermediate output of the first and second steps serves as input to the heart of the inference process, represented as step 3 in Figure 1.3. The inference is performed within the LISA [26] language description environment, with application of grammar inference algorithms. The result of the inference process is a context-free grammar (CFG) that is generated concurrently with the XML file containing the metamodel that can be used to load the instance models into the modeling tool.

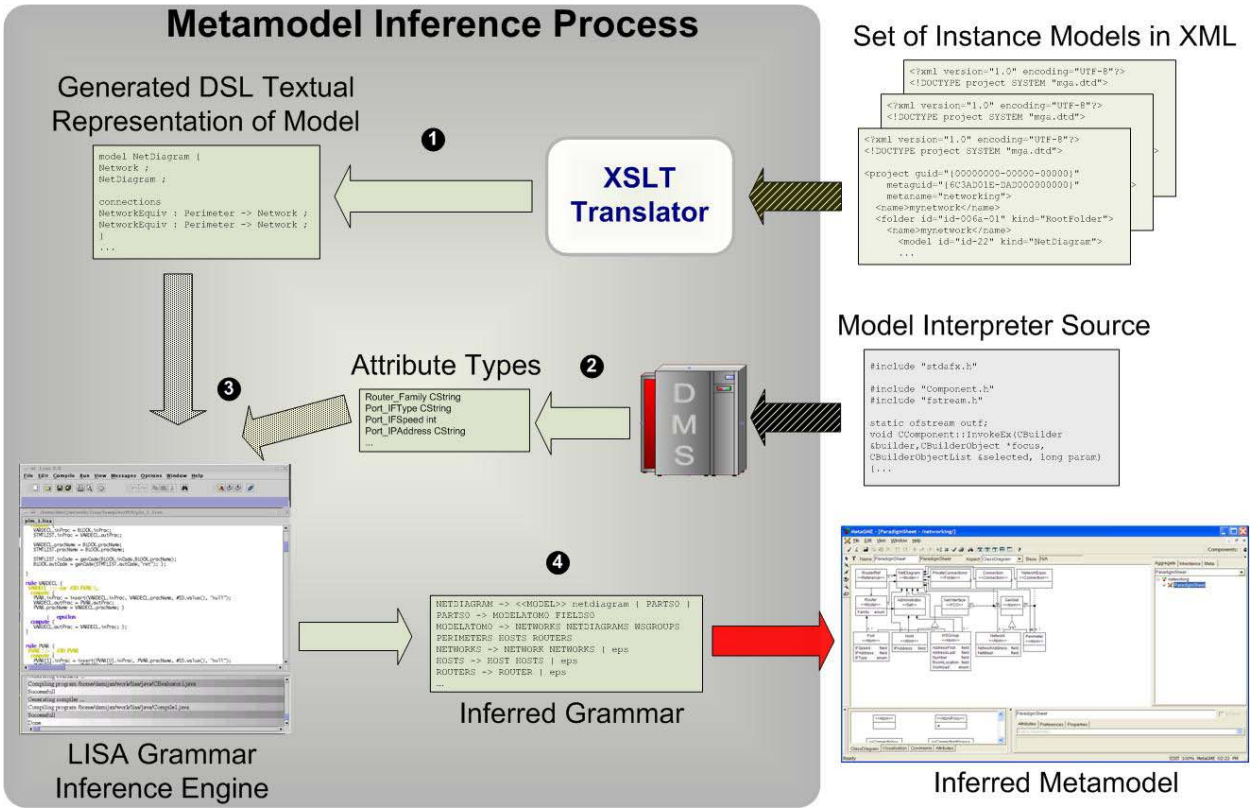


Figure 1.3: Overview of the MARS system

The remainder of the paper is organized as follows: Section 2 provides an overview of the background literature and technologies used throughout the paper. The translation of instance models into the representative DSL is described in Section 3. The key contribution of this section is a description of a transformation process based on XSLT that converts XML instance models into a representative DSL. A process for recovering name and type information from the source code of model interpreters is presented in Section 4. The heart of the paper is contained in Section 5, which explains the process of inferring a metamodel from the information provided by the DSL (Section 3) and the type resolution information (Section 4). Section 6 provides a survey of related literature. The final section offers a summary, as well as concluding remarks that describe current limitations and future work.

2. Background

This section provides a brief overview of several tools and research areas used in MARS. The section begins with a summary of grammar inference, with subsequent sub-sections introducing the various tools depicted in Figure 1.3.

2.1 Grammar Inference Overview

Inductive learning is the process of learning from examples [30]. The related area of grammatical inference can be defined as a particular instance of inductive learning where the examples are sets of strings defined on a specific alphabet. These sets of strings can be further classified into positive samples (the set of strings belonging to the target language) and negative samples (the set of strings not belonging to the target language). Primarily, grammatical learning research has focused on regular and context-free grammars. It has been proved that exact identification in the limit of any of the four classes of languages in the Chomsky hierarchy is NP-complete [13], but recent results have shown that the average case is polynomial [10]. A detailed survey of related literature on grammar inference is provided in Section 6.

2.2 Generic Modeling Environment

The Generic Modeling Environment (GME) [22] is the modeling tool used in the research described in this paper. The GME is a modeling environment that can be configured and adapted from meta-level specifications (called the metamodel, or modeling paradigm) that describe the domain. When using the GME, a modeler loads a metamodel into the tool to define an environment containing all the modeling elements and valid relationships that can be constructed in a specific domain [19]. Models are stored in the GME as objects in a database repository, and an API is provided by GME for model traversal using a standard integration mechanism (i.e., Microsoft Component Object Model – http://www.microsoft.com/com). From the API, it is possible to create interpreters that traverse the internal representation of the model and generate new artifacts (e.g., XML configuration files, source code, or even hardware logic) based on the model properties. In the GME, a metamodel is described with UML class diagrams and constraints that are specified in the Object Constraint Language (OCL) [38].

2.3 The LISA Language Development Environment

LISA [26] is an interactive environment for programming language development where users can specify, generate, compile-on-the-fly and execute programs in a newly specified language. From the formal language specifications and attribute grammars, LISA produces a language-specific environment that includes a compiler/interpreter and various language-based tools (e.g., language knowledgeable editor, visualizers, and animators) for the specified language. LISA was created to assist language designers and implementers in incremental language development [27]

of DSLs. This was achieved by introducing multiple inheritance into attribute grammars where the attribute grammar as a whole is subject to inheritance. Using the LISA approach, the language designer is able to add new features (e.g., syntax constructs and/or semantics) to the language in a simple manner by extending lexical, syntax and semantic specifications. LISA was chosen for this project because of its benefits in designing DSLs. Moreover, the LISA system has been used successfully in our evolutionary-based CFG inference engine [8]. The Model Representation Language described in Section 3 and the metamodel inference engine described in Section 5 were implemented using LISA.

2.4 Design Maintenance System

The Design Maintenance System (DMS) [3] is a program transformation system and re-engineering toolkit developed by Semantic Designs (http://www.semdesigns.com). The core component of DMS is a term rewriting engine that provides powerful pattern matching and source translation capabilities. In DMS terminology, a language domain represents all of the tools (e.g., lexer, parser, pretty printer, rule applier) for performing transformation within a specific programming language. In addition, DMS defines a specific language called PARLANSE, as well as a set of APIs (e.g., Abstract Syntax Tree API, Symbol Table API) for writing DMS applications to perform sophisticated program analysis and transformation tasks. DMS was chosen for this project because of its scalability for parsing and transforming large source files in several dozen languages (e.g., C++, Java, COBOL, Pascal). Because different modeling tools may use numerous programming languages to produce model interpreters, different parsers are needed for handling each language. Another consideration for the choice of DMS comes from our past success in using it to parse millions of lines of C++ code [15]. In Section 4, DMS is used in a technique to parse model interpreters to mine the type information of metamodel elements.

3. Generating a DSL from Domain Models

In [24] the role of grammar-based systems was investigated, where a grammar-based system was defined as, "*any system that uses a grammar and/or sentences produced by this grammar to solve various problems outside the domain of programming language definition and its implementation. The vital component of such a system is well structured and expressed with a grammar or with sentences produced by this grammar in an explicit or implicit manner.*" One of the identified benefits of such systems is that some problems can be solved simply by converting the representation at hand to a CFG, since appropriate tools and methods for working with CFGs already exist.

A difficulty in inferring metamodels is the mismatch in notations. Each GME instance model is persistently stored as an XML file, but the grammar inference process is better suited for a more constrained language. The inference process could be applied to the XML representation, but at a cost of greater complexity. To bridge the gap between the representations and make use of already existing inference tools and techniques, a DSL was created called the *Model Representation Language* (MRL). The defining properties of DSL's are that they are small, more focused than General-Purpose Programming Languages (GPL's), and usually declarative. Although DSL development is hard, and requires domain knowledge and language development expertise, language development toolkits exist which facilitate the DSL implementation process [25]. LISA was used to develop the MRL language and supporting tools.

3.1 Model Representation Language Design

The primary use of the MRL language is to describe the components of the domain models in a form that can be used by a grammar inference engine. In particular, an MRL program contains the various metamodel elements (e.g., models, atoms and connections) that are stated in a specific order. The most useful information contained in the domain models is the kind (type) of the elements, which can be either a model, atom, field or connection. This stage of the inference process is not concerned with the name of the modeling instance, but rather its type. Thus, an MRL program is a collection of <kind, identifier> implicit bindings. In terms of declaration order, models and atoms can be declared in any order. However, there is a particular order to be followed when declaring the composition of models and atoms. A model must first declare any constituent atoms and models, followed by field and then connection declarations. Because MRL

lacks imperative language characteristics (e.g., assignment statements, explicit execution order and control structures), it is more declarative than imperative. The complete grammar for the MRL language (in LISA specification format) is given in Figure 3.1. The MRL is sufficiently succinct in its design so as to be feasible for the inference process, but also expressive enough to represent the GME metamodels accurately.

```
START ::= GME
GME ::= MODEL_OR_ATOM GME | MODEL_OR_ATOM
MODEL_OR_ATOM ::= MODEL | ATOM
MODEL ::= model #Id \{ M_BODY \}
M_BODY ::= MODELS FIELDS CONNECT
MODELS ::= #Id \; MODELS | epsilon
FIELDS ::= fields OTHER_FIELDS \;
OTHER_FIELDS ::=  #Id \, OTHER_FIELDS | epsilon
CONNECT     ::= connections CONNECTIONS  | epsilon
CONNECTIONS ::= #Id \: #Id \-> #Id \; CONNECTIONS | epsilon
ATOM ::= atom #Id \{ FIELDS \}
```

Figure 3.1: Context-Free Grammar for MRL in LISA specification format

3.2 From GME Models to MRL: The XSLT Translation Process

A first stage of this work was to design a DSL that could accurately represent the (visual) GME domain models in a textual form. This was accomplished through a visual-to-textual-representation transformation process. Because the GME models are represented as XML files, the transformation to MRL is done by the Extensible Stylesheet Language Transformation Language (XSLT) [6]. XSLT is a transformation language that can transform XML documents to any other text-based format. XSLT uses the XML Path Language (XPath) [7] to locate specific nodes in an XML document. XPath is a string-based language of expressions. We detail the transformation process using our running example of the Network domain. One of the relatively intricate problems was to extract all the connection sets of a particular model. In a GME model, a connection is described by a connection name, the source element, and the destination element. Listing 3.1 shows an XML fragment from a Network domain instance model XML file.

```
01      <connection id="id-0068-00000070" kind="Connection" role="Connection">
02              <name>Connection</name>
03              <connpoint role="dst" target="id-0066-00000080"/>
04              <connpoint role="src" target="id-0066-00000086"/>
05          </connection>
06          <connection id="id-0068-00000071" kind="Connection" role="Connection">
07              <name>Connection</name>
08              <connpoint role="dst" target="id-0066-00000084"/>
09              <connpoint role="src" target="id-0066-00000081"/>
10          </connection>
11          <connection id="id-0068-0000006c" kind="Connection" role="Connection">
12              <name>Connection</name>
13              <connpoint role="src" target="id-0066-00000083"/>
14              <connpoint role="dst" target="id-0066-00000084"/>
15          </connection>
```

Listing 3.1: XML code fragment from a Network domain instance model

```
01 connections <br/>
02  <xsl:for-each select = "connection"> <xsl:variable
03 name = "conn2" select = "@id" />
04
05  <xsl:variable name = "connpt2"
06                  select = "connpoint[@role='src']/@target" /> <xsl:variable name =
07                          "connpt4" select = "connpoint[@role='dst']/@target" />
08
09 <xsl:value-of select="name"/> :
10  <xsl:call-template name = "scopeX">
11    <xsl:with-param name = "targ" select = "$connpt2"> </xsl:with-param>
12  </xsl:call-template>
13 ->
14  <xsl:call-template name = "scopeX">
15    <xsl:with-param name = "targ" select = "$connpt4"> </xsl:with-param>
16  </xsl:call-template>  ;
17 <br/>
18
19 <xsl:template name = "scopeX" >
20  <xsl:param name = "targ"> </xsl:param>
21     <xsl:variable name = "parenter" select = "//parent::node()[@id = $targ]/name" />
22     <xsl:variable name = "IDer" select = "//parent::node()[@id = $targ]/@kind" />
23  <xsl:value-of select = "$IDer"/>
24 </xsl:template>
```

Listing 3.2: XSLT code to extract the connection elements.

The fragment describes three XML container elements of type *connection*. Container elements are composite structures that can contain other elements. The connection element is composed of *name* and *connpoint* elements (see lines 02-04). The connpoint elements describe the source and destination of the connection. In a GME model XML file, each element is assigned a unique ID. The connpoint elements do not contain the name of the source or destination elements. Rather, only their IDs are mentioned in the *target* attribute tag. Thus, in order to retrieve the name and type of the source and destination elements, the instance model XML file had to be searched to find the element with the requisite ID, and then extract the type and name from that particular element declaration. This was accomplished by the XSLT transformation in Listing 3.2. The XPath expression *"connpoint[@role='src']/@target"* in line 06 translates to "the *target* attribute of the connpoint element which has a *role* attribute value of 'src' ". In lines 06 and 07, these XPath expressions are used in XSLT statements to assign the source and destination IDs to variables that are then passed as parameters to the XSLT template *scopeX* (lines 19-23). The main purpose of the scopeX template is to educe the value of the *kind* attribute of the element ID that is passed to it. The attribute *kind* indicates the metamodel element of an instance. Note that we are more concerned with the *kind* of the element than its *name*. The XPath expression *"//parent::node()[@id = $targ]/@kind"* selects the *kind* attribute of the element whose ID attribute matches the $targ variable. Part of the MRL program generated after the transformation is shown in Figure 3.2, which shows an instance of the NetDiagram model. Using our XSLT translation engine, the connections information in Listing 3.1 is transformed into an equivalent MRL representation. However, it is beneficial to prune the generated MRL program before it is used as an input for the grammar inference phase detailed in Section 5. The domain instance models usually have similar instances of atoms and models in their definitions (note the duplications in Figure 3.2). Multiple declarations of a particular model occasionally vary in their compositions and can be useful in inferring the correct cardinality of the model's composing elements. However, the atom definitions are static and multiple declarations of the same atom can be removed. Although the generated program can be used as is for the next phase, it is desirable to pare down the program for improved readability and succinctness.

```
…….
model NetDiagram {
WSGroup ;
Perimeter ;
Host ;
Network ;
WSGroup ;
Host ;
Router ;

fields;

connections
Connection : Port -> Network ;
Connection : Host -> Network ;
Connection : Port -> Perimeter ;
Connection : WSGroup -> Network ;
Connection : WSGroup -> Network ;
Connection : Host -> Network ;
}
……..
```

Figure 3.2: Part of the generated MRL program

4. Name and Type Inference from Model Interpreters

The previous section described the generative process for creating the MRL specification of the entities (e.g., models, atoms) and relations (e.g., connections), as well as their corresponding attributes from the domain models represented by XML. However, for each attribute of the model elements, it is not possible to infer the element type from the representative XML of the model instances. For example, in one of the instances of the Network domain, there is an attribute called "Port_IFSpeed" in a "Port" atom that is named "S0" (located in the "inetgw" of Figure 1.2). The value of this attribute is "128," but, the representative type could be integer, string, or even an enumerated type. In order to narrow down the selection scope of the possible types, additional model artifacts need to be mined. This section introduces a technique that infers some of the model types from existing model interpreters associated with the mined instance models.

4.1 An Example GME Model Interpreter

Listing 4.1 illustrates a fragment of a GME model interpreter implemented in C++ for processing the routers in the Network domain diagram. The "`ProcessRouter`" method takes an instance of Router as an argument, displays the router attribute "`Router_Family`," navigates each port

inside and prints out the port attributes "`Port_IFType`," "`Port_IFSpeed`," and "`Port_IPAddress`." The method "`GetAttribute`" is used to retrieve the attribute value according to the attribute name ("`Router_Family`" in Line 12) in the model and store it in a variable ("`fam`" in Line 12). The attribute name should be exactly the same as the name shown in the corresponding model because the interpreter is referencing metamodel concepts. Consequently, the type of the variable that is used in the interpreter to represent the attribute corresponds to the actual attribute type in the model (i.e., the attribute "`Router_Family`" can be inferred as type string based on the variable "`fam`" that is declared as a "CString" in Line 5 of the interpreter code fragment). From this perspective, model interpreters can be used to help infer the attribute types of metamodel elements. The next sub-section will introduce the implementation of a name and type inference technique that is based on analysis of model interpreters.

```
1  void CComponent::ProcessRouter(CBuilderModel *r) {
2
3     ASSERT(r->GetKindName() == "Router");
4
5     CString fam;
6
7     {
8        int fam;
9        ……
10    }
11
12    r->GetAttribute("Router_Family", fam);
13
14    outf << "\tRouter " << r->GetName() << " (" << fam << ")" << endl;
15
16    int ifspeed;
17
18    const CBuilderAtomList *ports = r->GetAtoms("Port");
19    POSITION pos = ports->GetHeadPosition();
20
21    while(pos) {
22
23       CBuilderAtom *port = ports->GetNext(pos);
24
25       CString iftype, ipaddr;
26
27       port->GetAttribute("Port_IFType", iftype);
28       port->GetAttribute("Port_IFSpeed", ifspeed);
29       port->GetAttribute("Port_IPAddress", ipaddr);
30
31       outf << "\t\tPort " << port->GetName();
32       outf << "(" << iftype << ";"<< ifspeed << "Kbps), Addr:" << ipaddr << endl;
33    }
34 }
```

Listing 4.1: Example of the model interpreter to process routers in the Network domain diagram

4.2 Metamodel Type Inference through Analysis of Model Interpreters

The general idea of implementing a type inference system is to set up a symbol table for the model interpreter source code. A symbol table stores all of the variables along with appropriate attributes (e.g., scope of validity, type, and value). Figure 4.1 describes a simplified symbol table for the "`ProcessRouter`" method in Listing 4.1. This symbol table contains three symbol spaces that represent three different lexical scopes: method body, block (corresponds to Line 7 to 10 in Listing 4.1), and a while block (corresponds to Line 21 to 33 in Listing 4.1). Each symbol space contains the variable names as well as their declaration types that are valid within the current lexical scope. By using the DMS Symbol Table API, a symbol table can be created easily during the parsing process.

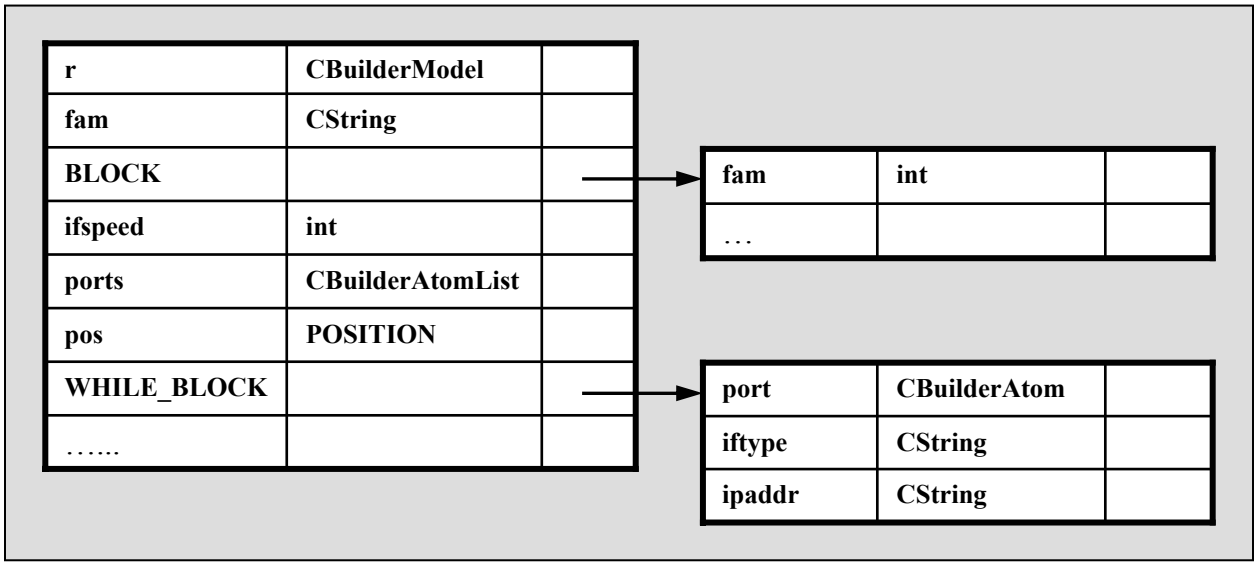


Figure 4.1: Symbol Table for the ProcessRouter method

Once the symbol table is constructed, it can be used to find out the variables that represent the model attributes. DMS offers the facilities to manipulate an Abstract Syntax Tree (AST) by invoking interface functions. Part of the PARLANSE implementation shown in Listing 4.2 searches the attribute variables in the interpreter source code. The "`AST:ScanNodes`" function traverses each node in the syntax tree. If the current visited node has a literal string value "`GetAttribute`" (Lines 7 and 8), the analysis determines the corresponding sub-tree "`expr_list`" from which the "`attribute_string`" (i.e., the real attribute name in the model) and "`attribute_id`" (i.e., the variable that is used to represent the attribute) can be extracted. After such an attribute name and variable pair is found, PARLANSE will look up this

variable in the symbol table and return its corresponding type. As a result, a file of attribute name and type-pair listings (see "Attribute Types" icon in Figure 1.3) will be generated to serve as an input for the inference techniques described in the next section.

```
1  (AST:ScanNodes syntax_tree
2     (lambda (function boolean AST:Node )function
3        (value (local (;; [attribute_string (reference string)]
4                          [attribute_id (reference string)]
5                          [expr_list AST:Node]
6                       );;
7           (ifthen (== (AST:GetNodeType ?) _identifier)
8             (ifthen (== (@ (AST:GetString ?)) 'GetAttribute')
9                (;;(= expr_list (AST:GetThirdChild (AST:GetParent
10                                  (AST:GetParent (AST:GetParent (AST:GetParent ?))))))
11                  (AST:ScanNodes expr_list
12                     (lambda (function boolean AST:Node )function
13                        (value (local (;; );;
14                           (;;(ifthen (== (AST:GetNodeType ?) _STRING_LITERAL)
15                                 (= attribute_string (AST:GetString ?))
16                              )ifthen
17                              (ifthen (== (AST:GetNodeType ?) _identifier)
18                                 (;; (= attribute_id (AST:GetString ?))
19                                     ……
20 <closing parenthesis removed>
```

Listing 4.2: PARLANSE code fragment for searching the attribute variables in the interpreter source program

5. Metamodel Inference Engine

As previously discussed in Section 3, a correspondence exists between a metamodel and its instances, and a programming language and valid programs defined on the language. By considering a metamodel as a representation of a CFG, named G, the corresponding instance models can be delineated as sentences generated by the language L(G). This section describes the subcomponent of the MARS system that applies CFG inference methods and techniques to the metamodel inference problem.

5.1 Models as MRL Programs

In the first step toward defining a process to infer a metamodel, all of the relationships and transformation procedures between metamodels and CFG's were identified. A key part of the

process involves the mapping from the metamodel representation to the non-terminals and terminals of a corresponding grammar. The role of non-terminal symbols in a CFG is two fold. At a higher level of abstraction, non-terminal symbols are used to describe different concepts in a programming language (e.g., an expression or a declaration). At a more concrete lower level, non-terminal and terminal symbols are used to describe the structure of a concept (e.g., a variable declaration consists of a variable type and a variable name). Language concepts, and the relationships between them, can be represented by CFG's. This is also true for the GME metamodels [19], which describe concepts (e.g., model, atom) and the relationships that hold between them (e.g., connection). Therefore, both formalisms can be used for the same purpose (at differing levels of abstraction), and a two-way transformation from a metamodel to a CFG can be defined. The transformations relating a metamodel to a CFG are depicted in Table 5.1.

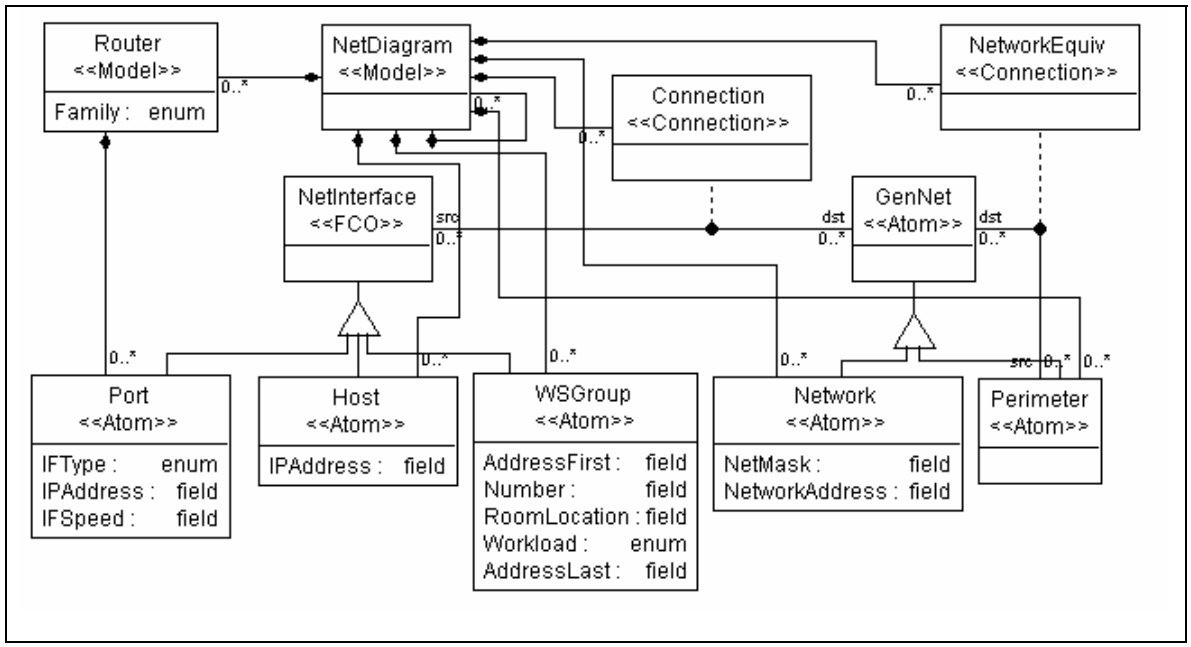
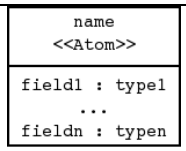
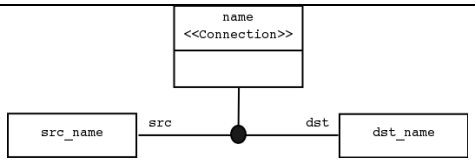
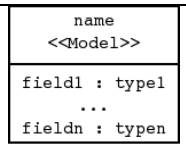
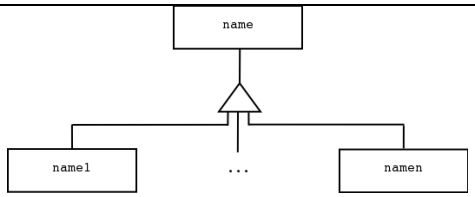
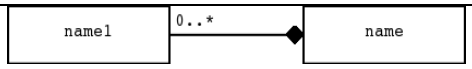
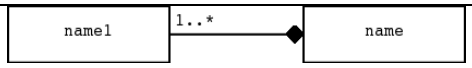
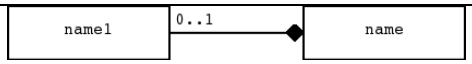
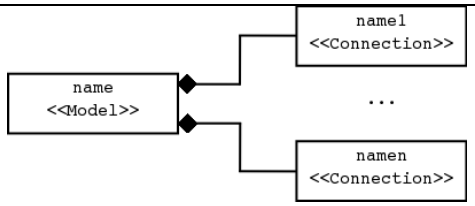
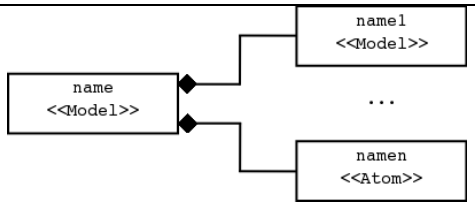


Figure 5.1: Network domain metamodel (extracted from Figure 1.1)

Table 5.1: Transformation from a metamodel to a context-free grammar

1.		NAME→<<ATOM>> name {FIELDS} FIELDS → <<FIELDS>> field1 ... fieldn
2.		NAME → <<CONNECTION>> name: SRC -> DST; SRC → SRC_NAME DST → DST_NAME
3.		NAME → <<MODEL>> name {PARTS} PARTS → MODELATOM FIELDS CONNECTIONS FIELDS → <<FIELDS>> field1 ... fieldn MODELATOM → ... CONNECTIONS → ... (see transformation 8 and 9)
4.		NAME → NAME1 | ... | NAMEn
5.		NAME → NAME1S NAME1S → NAME1 NAME1S | ε
6.		NAME → NAME1S NAME1S → NAME1 NAME1S | NAME1
7.		NAME → NAME1S NAME1S → NAME1 | ε
8.		CONNECTIONS → NAME1 ... NAMEn (see transformation 3)
9.		MODELATOM → NAME1 ... NAMEn (see transformation 3)

As an example application of the transformations in Table 5.1, the Network metamodel shown in Figure 5.1 is semantically equivalent to the corresponding CFG represented in Figure 5.2. The obtained CFG is a quite natural representation of the metamodel in Figure 5.1. Productions 1 and 2 state that a NetDiagram is a model consisting of models, atoms, fields or connections. Models and atoms (production 3) that can be used in a NetDiagram are Networks, NetDiagrams, WSGroups, Perimeters, Hosts and Routers. A NetDiagram has no fields (production 10) and two kinds of connections: NetworkEquiv and Connection (production 11). The cardinalities of particular models and atoms are easily identified from productions 4 to 9, namely 0 .. *. Note that elements like <<MODEL>> and <<FIELDS>> are special annotated terminal symbols that appear without triangle brackets in the generated sentences. From such a description, a metamodel can be easily drawn manually using the GME tool. However, this manual process can be automated. Using the transformation described in Section 5.3, the CFG is transformed to the GME metamodel XML representation that can be loaded by GME.

```
1.  NETDIAGRAM → <<MODEL>> netdiagram { PARTS0 }
2.  PARTS0 → MODELATOM0 FIELDS0 CONNECTIONS0
3.  MODELATOM0 → NETWORKS NETDIAGRAMS WSGROUPS PERIMETERS HOSTS ROUTERS
4.  NETWORKS → NETWORK NETWORKS | ε
5.  NETDIAGRAMS → NETDIAGRAM NETDIAGRAMS | ε
6.  WSGROUPS → WSGROUP WSGROUPS | ε
7.  PERIMETERS → PERIMETER PERIMETERS | ε
8.  HOSTS → HOST HOSTS | ε
9.  ROUTERS → ROUTER ROUTERS | ε
10. FIELDS0 →  ε
11. CONNECTIONS0 → NETWORKEQUIV CONNECTION
12. NETWORKEQUIV → <<CONNECTION>> networkequiv : SRC0 -> DST0 ; NETWORKEQUIV
    | ε
13. SRC0 → PERIMETER
14. DST0 → <<ATOM>> GENNET
15. GENNET → NETWORK | PERIMETER
16. CONNECTION → <<CONNECTION>> connection : SRC1 -> DST1 ; CONNECTION | ε
17. SRC1 → <<FCO>> NETINTERFACE
18. DST1 → GENNET
19. NETINTERFACE → PORT | HOST | WSGROUP
20. ROUTER → <<MODEL>> router { PARTS1 }
21. PARTS1 → MODELATOM1 FIELDS1 CONNECTIONS1
22. MODELATOM1 → PORTS
23. PORTS → PORT PORTS | ε
24. FIELDS1 → <<FIELDS>> Family
25. CONNECTIONS1 → ε
26. PERIMETER → <<ATOM>> perimeter { FIELDS2 }
27. FIELDS2 → <<FIELDS>>
28. PORT → <<ATOM>> port { FIELDS3 }
29. FIELDS3 → <<FIELDS>> IFSpeed IPAddress IFType
30. WSGROUP → <<ATOM>> wsgroup { FIELDS4 }
31. FIELDS4 → <<FIELDS>> AddressFirst AddressLast Number Workload
    RoomLocation
32. HOST → <<ATOM>> host { FIELDS5 }
33. FIELDS5 → <<FIELDS>> IPAddress
34. NETWORK → <<ATOM>> network { FIELDS6 }
35. FIELDS6 → <<FIELDS>> NetworkAddress NetMask
```

Figure 5.2: Context-Free Grammar representation of the metamodel in Figure 5.1

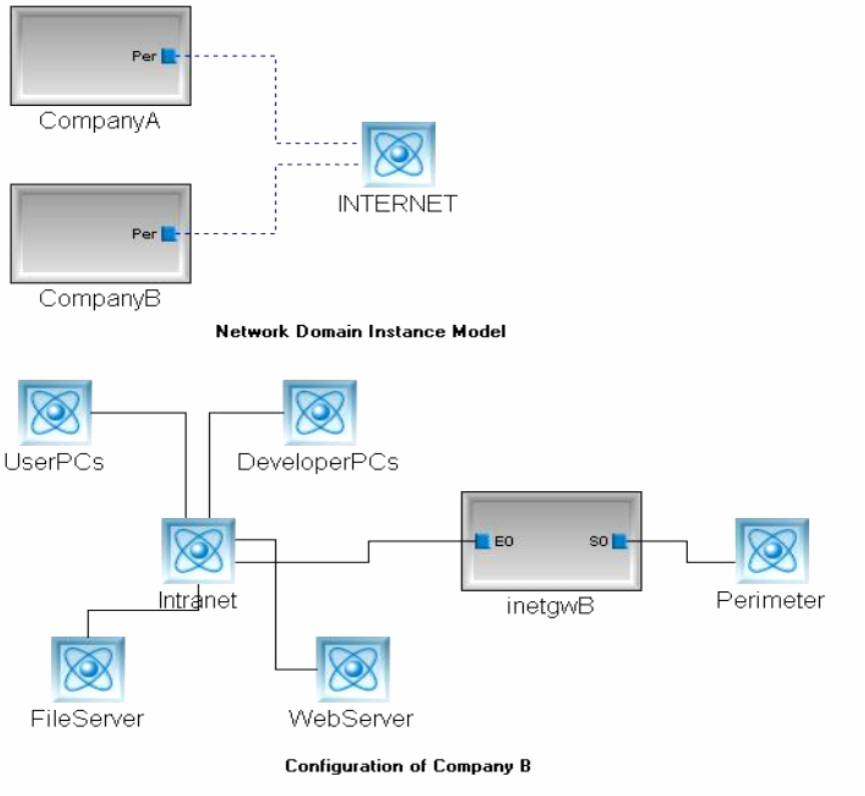


Figure 5.3: Network domain instance model

```
model netdiagram {
   atom network {fields NetworkAddress NetMask}
   model netdiagram { ... }
   model netdiagram { ... }
   connection networkequiv :
      atom perimeter { fields } ->
      atom network {fields NetworkAddress NetMask};
   connection networkequiv :
      atom perimeter { fields } ->
      atom network {fields NetworkAddress NetMask};
}
```

Figure 5.4: MRL program as representation of an instance model from Figure 5.3

The instance model shown in Figure 5.3, which is based on the metamodel of Figure 5.1, is represented as the sentence (in Figure 5.4) generated from the CFG in Figure 5.2 (note: the native visualization icons, as shown in Figure 1.2, are not available at this point in the process). The primary artifact from which a metamodel will be inferred is the intermediate textual representation as translated in the MRL language. The remainder of this section describes the metamodel inference process using LISA and the MRL.

5.2 The Induction Process

Because existing CFG inference algorithms are successful only at inducing small programming languages, the MRL language description has been purposely simplified, but is complete such that all of the essential mappings between the metamodel and MRL are possible. Note that for the inference process, it is sufficient to know the important constituent elements of the metamodel. For example, it is sufficient to know that a particular model consists of a Network, two NetDiagrams and two connections of type NetworkEquiv. From this particular model one can infer that a network diagram consists of 1 network, 1..* net diagrams and 1..* network equivs (more model instances are needed for more accurate description). Therefore, the NetDiagram example is further simplified (Figure 5.5).

```
model netdiagram {
network
netdiagram
netdiagram
networkequiv : perimeter -> network
networkequiv : perimeter -> network
}
```

Figure 5.5: Simplification of instance model from Figure 5.3

Because we were previously successful in inferring DSL's of a similar size using the evolutionary-based CFG inference engine [8], the same inference system was applied to induce the appropriate CFG's for the MRL language. The evolutionary-based CFG inference engine accepts positive and negative sentences (in the case of metamodel inference, negative sentences do not exist) and generates an initial population of CFGs based on positive sentences. These CFGs are then evaluated and better grammars are selected for the next generation. Before one evolutionary cycle is completed, mutation, crossover and heuristic operators are performed on randomly chosen CFGs. Through many generations, CFGs evolve in such a manner that they correctly recognize all positive sentences and reject all negative sentences. From many NetDiagram models, such as presented in Figure 5.5, the evolutionary-based CFG inference engine will induce the grammar shown in Figure 5.6.

```
1.  NT7 -> NT6 AT1
2.  NT6 -> NT5 NT3
3.  NT5 -> NT4 NT2
4.  NT4 -> AT4 AT0
5.  NT3 -> FR1 NT3
6.  NT3 -> ε
7.  NT2 -> AT4 NT2
8.  NT2 -> ε
9.  FR1 -> AT5 AT3 AT4 AT2 AT4
10. AT5 -> networkequiv | connection
11. AT4 -> netdiagram | router | port | host | wsgroup | network | perimeter
12. AT3 -> :
13. AT2 -> ->
14. AT1 -> }
15. AT0 -> {
```

Figure 5.6: Inferred grammar using evolutionary-based CFG inference engine

An example of a derivation tree using the grammar from Figure 5.6 is shown in Figure 5.7, where one can see that the inferred grammar actually describes a model as a set of models/atoms (nonterminal AT4) and connections (nonterminal FR1).

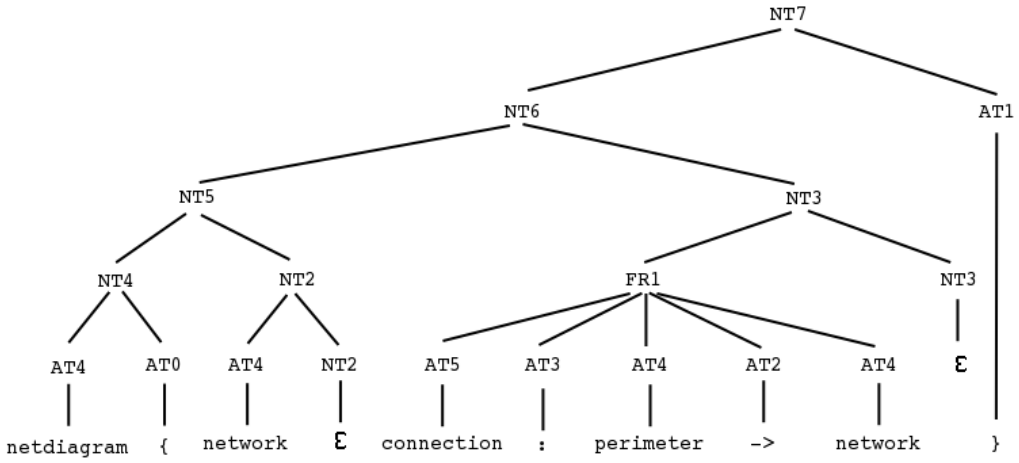


Figure 5.7: Derivation tree

However, this grammar is not appropriate for our task because the NetDiagram metamodel can consist of several connections (see productions 5 and 6: `NT3` → `FR1 NT3`, `NT3` → ε and

production 9: `FR1 -> AT5 AT3 AT4 AT2 AT4`) where the source and destination of a connection can be any model and atom element (see production 11). For example, source can be a router or netdiagram even though none of the input models expose such a case. Another problem is that the inferred CFG is not very close to the suggested grammar from Table 5.1. In other words, the inferred CFG cannot be used in the transformation back to metamodels. Negative examples help converge the grammar to be inferred, and keep it from becoming too generalized. Because negative examples do not exist in the metamodel inference paradigm, the inferred CFG was greatly generalized. The inferred grammar has a direct correspondence to the definition of the GME meta-metamodel.

One of the most important results in the field of general grammatical inference states that a CFG cannot be induced solely from positive samples, except under special circumstances [13]. However, this limitation can be overcome if additional knowledge about a metamodel is used. For example, a model can contain other elements that are of type model, atom, field, and connection. Therefore, the following productions are required:

```
MODELNAME -> <<Model>> name { PARTS }
PARTS -> MODELATOMS FIELDS CONNECTIONS
```

An atom is a primitive type that cannot contain other elements, just fields.

```
ATOMNAME -> <<ATOM>> name { FIELDS }
FIELDS -> <<FIELDS>> field1 ... fieldn
```

Additional knowledge can also be assumed for connections, which always contain two parts: *source (src)* and *destination (dst)*. The source and destinations of a connection usually encapsulate several different model or atom elements, and are suitable for generalization. Therefore, the following production is required:

```
CONNECTIONNAME -> <<CONNECTION>> name : SRC -> DST1; CONNECTIONNAME | ε
```

Furthermore, cardinality information can be inferred easily if enough model examples are available. It is not possible to generalize the cardinality value from a collection of instance

models that always exhibit the same cardinality, but the cardinality can be generalized from a sample of models that differ in their definitions. Of course, the most general case can be assumed (i.e., a cardinality value of 0..*). All of this information can be collected and stored in attributes (e.g., `in_fields`, `in_con`, `ma`) during the parsing of models translated to MLR, as illustrated in Figure 5.8. For example, the rule called `Model` of Figure 5.8 defines syntax and semantics for the model. The meaning of the model is stored in attribute `ma` which is an object of class Model. The following information is stored: model name (`#Id.value()`), models and/or atoms that this particular model consists of (`M_BODY.mas`), fields (`M_BODY.fields`) and connections (`M_BODY.conn`). Atttributes `M_BODY.mas`, `M_BODY.fields` and `M_BODY.conn` are computed in the rule `Model_Body`. Using this additional knowledge about metamodels, it is possible to infer the CFG that represents the metamodel. The inferred CFG for the Network example is shown in Figure 5.9, which is quite similar to the manually created CFG shown previously in Figure 5.2

```
rule Model {
    MODEL ::= model #Id \{ M_BODY \} compute {
      MODEL.ma=new Model(#Id.value(), M_BODY.mas, M_BODY.fields, M_BODY.conn);
    };
  }

  rule Model_Body {
    M_BODY ::= MODELS FIELDS CONNECT compute {
      MODELS.in_ma = new Vector();
      FIELDS.in_fields = new Vector();
      CONNECT.in_con = new Vector();
      M_BODY.mas = MODELS.out_ma;
      M_BODY.fields = FIELDS.out_fields;
      M_BODY.conn = CONNECT.out_con;
    };
  }
```

Figure 5.8: An MLR attribute grammar fragment

```
NETDIAGRAM -> <<MODEL>> netdiagram { PARTS0 }
PARTS0 -> MODELATOM0 FIELDS0 CONNECTIONS0
MODELATOM0 -> NETWORKS NETDIAGRAMS WSGROUPS PERIMETERS HOSTS ROUTERS
NETWORKS -> NETWORK NETWORKS | eps
NETDIAGRAMS -> NETDIAGRAM NETDIAGRAMS | eps
WSGROUPS -> WSGROUP WSGROUPS | eps
PERIMETERS -> PERIMETER | eps
HOSTS -> HOST HOSTS | eps
ROUTERS -> ROUTER | eps
FIELDS0 ->  eps
NETWORKEQUIV -> <<CONNECTION>> networkequiv : SRC0 -> DST0 ; NETWORKEQUIV | eps
SRC0 -> <<FCO>> FCO1
DST0 -> <<FCO>> FCO2
FCO1 -> PERIMETER
FCO2 -> NETWORK
CONNECTION -> <<CONNECTION>> connection : SRC1 -> DST1 ; CONNECTION | eps
SRC1 -> <<FCO>> FCO3
DST1 -> <<FCO>> FCO4
FCO3 -> PORT|HOST|WSGROUP
FCO4 -> NETWORK|PERIMETER
CONNECTIONS0 -> NETWORKEQUIV CONNECTION
ROUTER -> <<MODEL>> router { PARTS1 }
PARTS1 -> MODELATOM1 FIELDS1 CONNECTIONS1
MODELATOM1 -> PORTS
PORTS -> PORT PORTS | PORT
FIELDS1 -> <<FIELDS>>  Family
CONNECTIONS1 -> eps
WSGROUP -> <<ATOM>> wsgroup { FIELDS2 }
FIELDS2 -> <<FIELDS>> Workload RoomLocation Number AddressLast AddressFirst
NETWORK -> <<ATOM>> network { FIELDS3 }
FIELDS3 -> <<FIELDS>> NetMask NetworkAddress
PORT -> <<ATOM>> port { FIELDS4 }
FIELDS4 -> <<FIELDS>> IFType IFSpeed PortIPAddress
HOST -> <<ATOM>> host { FIELDS5 }
FIELDS5 -> <<FIELDS>> IPAddress
PERIMETER -> <<ATOM>> perimeter { FIELDS6 }
FIELDS6 -> eps
```

Figure 5.9: Inferred CFG for the Network Domain metamodel

5.3 CFG to XML: Transformation Back to the Metamodel Representation

The final step in the MARS inference system is the transformation of the inferred CFG back to the GME metamodel representation. This step requires that the XML representation of the metamodel be generated, and the same transformation rules presented in Table 5.1 can be applied to this task. In fact, the metamodel XML file is concurrently generated with the CFG. The GME metamodel conforms to a specific XML schema definition that must be adhered to during the transformation. A fragment of the LISA code to perform the transformation is shown in Figure 5.10. The sample code illustrates the use of information collected from the MRL representation of the instance model through the use of attribute grammars. The code fragment specifically generates the XML representation for GME modeling elements in accordance with the GME metamodel schema rules. Concurrently, the grammar rule specifying atom components is also generated.

```
if ( mAtoms.containsKey(atomNm) ) {
        aID = (String)mAtoms.get(atomNm);
        xmlfile.write("      <atom id = \"" + aID + "\" kind = \"Atom\" role=\"Atom\" > \n");
        xmlfile.write("          <name>" + atomNm + "</name> \n ");
        xmlfile.write("          <attribute kind = \"IsAbstract\" status= \"meta\" > \n");
        xmlfile.write("                 <value>false</value> \n");
        xmlfile.write("          </attribute> \n");
        xmlfile.write("          <attribute kind="InRootFolder" status="meta">\n");
        xmlfile.write("             <value>false</value> \n");
        xmlfile.write("          </attribute>\n");
        xmlfile.write("      </atom> \n\n ");
                          }

        StringBuffer atname = new StringBuffer(atoms0 + " -> " + "<<ATOM>> "
                                  + atoms0.toLowerCase() + "{ FIELDS" + fdnum + " }\n");

        outfile.write(atname.toString());
```

Figure 5.10: LISA code fragment to generate the CFG and metamodel XML file for atoms

Figure 5.11 shows the inferred metamodel for the Network domain. In terms of syntactic and semantic functionality, the inferred and original metamodel are comparable. The only difference is in the total number of components generated and the arrangement of the inheritance and

generalization hierarchy. The original metamodel (Figure 5.1) had the *port, host* and *wsgroup* atoms inheriting from the NetInterface FCO, which is an abstract generalization used to modularize interactions among models. However, this information cannot be mined because no remnants of this hierarchy are detailed in the instance model XML file. Any abstraction that is not explicitly mentioned in the instance model cannot be inferred to a metamodel. Despite this difference, the inferred metamodel enables the legacy instance models to be loaded into the GME, which achieves the primary goal of the research.

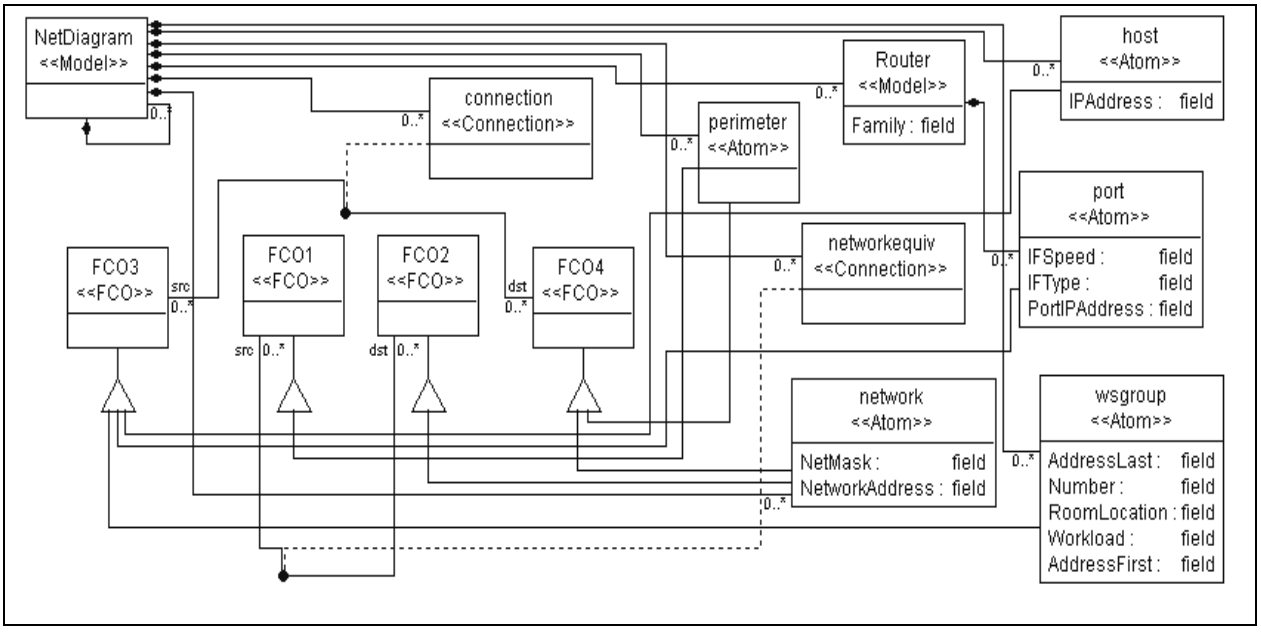


Figure 5.11: The inferred metamodel for the Network domain

6. Related Work

Most of the work in the area of grammatical inference has focused on devising new algorithms for inducing regular and CFG grammars. Grammatical inference algorithms have been applied in domains such as bioinformatics, pattern recognition and systems modeling. In this section, we elaborate on the current state of inference algorithms, and compare our work to two systems that take a grammatical inference approach to XML schema extraction.

6.1 An Overview of Regular and Context-Free Grammar Inference

Considerable advances have been made in inferring regular grammars, but learning CFG's has been proved to be more formidable. Most of the recent research in the regular language inference

domain has focused on inducing Abbadingo-style Deterministic Finite Automata (DFA) [2]. The Abbadingo-style problems have superceded the Tomita language [36] as the current benchmark set of grammars used for DFA induction. The advent of this benchmark problem set resulted in two successful DFA inference algorithms: the Evidence Driven State Merging (EDSM) [20] algorithm, and the Self-Adaptive Greedy Estimate (SAGE) [18] algorithm. Both algorithms substantially build on the accomplishments of the Traxbar [37] algorithm, which was one of the first attempts at inducing canonical DFA's in polynomial time using a state-merging process on an Augmented Prefix Tree Acceptor (APTA). EDSM employs a variant of the state-merging process in which only the pair of nodes whose sub-tree's share the most similar labels are merged. EDSM has a disadvantage in that considering every potential merge pair at each stage of the inference process is computationally expensive. Although EDSM variants have managed to decrease the running time, they are still an order of magnitude greater than those of the Traxbar algorithm. SAGE uses random sampling techniques on search trees to control the search. However, this random selection process ignores the clues in the training data, resulting in a lot of unnecessary search. As a result, SAGE is more computationally expensive than EDSM and its derivatives. Ed-Beam [21], a SAGE derivative, makes use of the matching labels heuristic and has a lower computation time.

CFG's are more expressive than regular grammars, and find applications in many areas like pattern and speech recognition, as well as syntax design of programming languages. Various approaches to CFG induction have been investigated, ranging from use of structural data to Bayesian methods. Sakakibara [32] used skeletal parse trees for inducing CFG's in polynomial time, and genetic algorithms have been used as a heuristic in various approaches [8, 29, 33]. SEQUITUR [29] is an algorithm that provides good compression rates and runs in linear time, but does not generalize the inferred CFG. SubDueGL [17] uses the Minimum Description Length (MDL) [31] principle for inducing context-free graph grammars. So far, all attempts at CFG induction have focused on toy problems with very few, if any, experiments on real examples.

6.2 Grammatical Inference Applied to XML Schema Extraction

Grammatical inference algorithms have also been applied to Document Type Descriptor (DTD) [4] and XML Schema extraction from XML documents. A DTD specifies the internal structure of an XML document using regular expressions. However, due to their limited capabilities in both syntax and expressive power for wide ranging applications, a host of grammar-based XML

schema languages like XML Schema [35] have been proposed to replace DTD. The XTRACT [11] system concentrates on inducing the DTD from XML documents using a regular grammar induction engine to infer equivalent regular expressions from DTD patterns, and then utilizes the MDL principle to choose the best DTD from a group of candidate DTD's. In [5], an Extended Context-Free Grammar (ECFG) based system is proposed to allow extraction of more complex XML schema's. Our induction system parallels these works in the sense that it is XML based, makes use of grammar inference algorithms, and extracts the metamodel (analogous to DTD or other forms of XML schemas) of a specific instance model (analogous to the XML document). Although our system is currently configured for the GME domain, it can be generalized easily to the XML schema extraction domain. We compare our work with the techniques for schema extraction using the following three measures: 1) intermediate representations, 2) content generalization, and 3) data-type identification. We chose these measures because they are the pivotal components of the induction process and represent areas that are best contrasted or mirrored in our approach.

1. Intermediate Representations: The XTRACT system proceeds to infer a DTD by deriving a regular expression for each element that appears in the XML document. The ECFG-based implementation represents the XML documents as structured examples of an unknown ECFG and uses pre-existing CFG inference algorithms to infer an ECFG. Hence, the XTRACT system and the ECFG implementation use regular expressions and ECFG's as intermediate representations respectively during the induction process. By comparison, the MARS induction process uses the MRL (with a corresponding parser written as LISA specifications) as an intermediate form, resulting in a higher level of abstraction. Unlike the ECFG-based system, no actual manipulation or transformation of grammars takes place in our system. Rather, the MRL grammar definition is static; with the use of attribute grammars coupled with an auxiliary LISA method, information pertaining to the generation of the metamodel XML file is gathered.

*2. Content Generalization***:** Content generalization refers to the process of being able to infer a concise rule or hypothesis regarding a set of elements. Ideally, the rule should neither over-generalize (i.e., predict elements which are generated from a more generic rule), nor be rigid (i.e., generate only the set of elements used to derive the rule itself). The XTRACT system's generalization heuristics make use of frequent, neighboring occurrences of subsequences and symbols within each input sequence resulting in a set of candidate DTD's from which the DTD with the best MDL score is chosen. The ECFG-

based system generalizes by merging non-terminals as long as no production in the grammar contains ambiguous terms. Our generalization mechanism is similar to the one found in the XTRACT system. Containment information regarding models and atoms in an instance model XML file is used to compute the cardinality of the constituent elements of the models and the connections.

3. Data Type Identification: Simple XML elements can have optional data types. There is no information available on the technique used by XTRACT to infer the data types of elements. The ECFG-based system analyzes the values of elements and then assigns the most suitable and constrictive data type to them. As described in Section 4, we use DMS PARLANSE to infer the model attribute types from the model interpreters, resulting in a high level of accuracy. This technique utilizes multiple artifacts of the modeling process in order to mine the required information to construct the metamodel.

By evaluating the MARS system with similar existing systems using these three measures, it can be observed that our work introduces a novel way to think about the inference problem in a context that has not been investigated previously (i.e., the domain-specific modeling context). The application of grammar inference to the metamodel problem introduces advanced inference techniques in the form of innovative uses of already existing tools and formulation of new algorithms.

7. Conclusion

The goal of the MARS project is to infer a metamodel from a collection of instance models. The motivating problem was to address the issue of metamodel drift, which occurs when instance models in a repository are separated from their defining metamodel. In most metamodeling environments, the instance models cannot be loaded properly into the modeling tool without the metamodel. The key contribution of the paper is the application of grammar inference algorithms to the metamodel recovery problem. The paper provided a description of all tasks involved in the inference process, with a common case study representing a network modeling language. Although the application of MARS has focused specifically on recovering metamodels for the GME, we believe that the same process can be applied to other metamodeling tools, such as MetaCase's metaEdit+ (http://www.metacase.com) and Honeywell's Domain Modeling Environment (DOME) (http://www.htc.honeywell.com/dome/). As an area of future work, we plan to explore the ability to apply MARS to these other environments. The remainder of this

concluding section provides a discussion of experimental results, limitations of the current approach, and details about where to find more information about the MARS project.

7.1 Experimental Study

To measure the effectiveness of MARS, we applied the system to domain models ranging from "toy" to "real-life" application domain models. We classify a domain model as a "toy" domain model if it can be used as a proof-of-concept for instructional purposes but is inadequate for any real-life applications. As a result, real-world application domain models are more complex and multi-faceted than their toy counterparts. We have applied the MARS system to the following domains (arranged in increasing order of complexity):

- Finite State Machine (FSM)
- Network Diagram (case study of this paper)
- Embedded Systems Modeling Language (ESML)

The FSM model is a simple model of computation consisting of a set of states, a start state, an input alphabet, a transition function, and a set of end states. It is also smaller and less elaborate than the Network model. Both Network and FSM models are examples of toy domain models. ESML is a domain-specific graphical modeling language developed for modeling Real-Time Mission Computing Embedded Avionics applications and is specifically used for modeling component-based avionics applications designed for the Bold-Stroke component deployment and distribution middleware infrastructure [34]. We classify ESML as a real-world application domain.

We were successful in inferring accurate metamodels for all the domains. Since it is not possible to infer accurately the generalization hierarchy of a metamodel from the instance model, the total number of elements generated in the inferred metamodel is almost always higher than the elements used in the original metamodel. However, we would still like to constrain the total number of elements generated in the inferred metamodel as much as possible. Thus, to measure the effectiveness of our approach using quantitative measures, we employ the use of a "Bloat" score, which is calculated using the number of elements generated in a metamodel. The Raw Bloat Score is calculated as follows:

<number of elements in inferred metamodel> – <number of elements in original metamodel>

Table 7.1 lists the Raw Bloat Score for the domain models. It was observed that as the complexity and size of the domain model increased, the Raw Bloat Score exhibited a higher divergence from the previous score. More experiments on domain models of varying size are needed before we can conclude whether the Raw Bloat Score growth factor is quadratic or exponential.

Table 7.1 Raw Bloat Scores for the Domain Models

Domain Model	Classification	# of Elements in Original Metamodel	# of Elements in Inferred Metamodel	Raw Bloat Score
Finite State Machine	Toy	7	8	1
Network	Toy	12	17	5
ESML	Real-Life	38	52	14

Another quantitative measure used was the Scaled Bloat Score, which is based on z-score normalization. In z-score normalization [39], the values for an attribute X are normalized based on the mean and standard deviation of X. The attribute set to be scaled is (# of Elements in Original Metamodel, # of Elements in Inferred Metamodel). The Scaled Bloat Score of a value w of an attribute A in the attribute set is normalized to w' by computing:

$$w' = (w - \check{A}) / \sigma_A ,$$

where σ_A and $\check{A}$ are the standard deviation and mean, respectively, of attribute A. Table 7.2 details the Scaled Bloat Scores of the domain models. An interesting observation here is that the scores are very similar for both the attributes. The conclusion derived is that although the Raw Bloat Score indicates a growing disparity in the number of elements, the scaled scores show that the increase in the number of elements in the models is approximately proportionate for each attribute. In other words, the number of elements in the models for each attribute differs by an almost constant factor. Despite this positive result, we would like to improve the MARS inference process to decrease this constant factor.

Table 7.2 Scaled Bloat Scores for the Domain Models

Domain Model	Scaled Bloat Score (original metamodel)	Scaled Bloat Score (inferred metamodel)
Finite State Machine	-0.883	-0.930
Network	-0.515	-0.456
ESML	1.398	1.388

7.2 Limitations and Future Work

There are several limitations to the current investigation. Each limitation can be eased by considering additional input from a human modeler. The complete metamodel inference process can be described as a semi-automatic technique that derives much of the required metamodel from the mined instance models, but must rely on a domain or modeling expert to complete a few parts of the task. The parts of the metamodel inference process that involve interaction with a user are:

- *Enumeration type determination*: The technique presented in Section 4 can be used to determine the metamodel type for integers and floating-point numbers, as well as distinguish between numeric types and strings. However, if the type is determined to be a string, then it cannot be decided whether the type is truly a string or an enumeration type. The GME persistently stores enumeration values as strings. The primary purpose of an enumeration type in a metamodel is to constrain the possible values of a string representation. However, this cannot be determined solely from an instance model and must involve human input. Future work will extend Section 4 such that a user is asked to categorize a string type as a true string, or as an enumeration type.
- *Domain-specific visualization*: In the GME, the visualization of a domain-specific model (i.e., the icons used in representing domain concepts) is specified in the metamodel. Each metamodeling entity can be assigned a file name that contains the graphic to be displayed when rendering the domain element. Because the visualization cannot be determined from the instance models, the inferred metamodel contains generic icons (such as those in Figure 5.3, when compared to the true visualization shown in Figure 1.2). To complete the visualization of the inferred metamodel, a domain expert needs to be consulted to associate the conceptual domain entities with appropriate visualization.

- *OCL Constraints*: An OCL constraint is specified at the GME metamodeling level to capture domain semantics that cannot be captured by static class diagrams. These constraints are evaluated when an instance model is created. Any violation of a constraint raises an error to inform the modeler that some aspect of the model is incorrect with respect to the metamodel. Constraints are very important in modeling, but MARS is unable to infer OCL constraints from instance models. The constraint definitions do not appear in the instance models explicitly. Because of this, an inferred metamodel needs to be augmented manually with constraints. Without constraints, the inferred metamodel may be too liberal with respect to the instance models that are to be accepted by the original metamodel that was lost (i.e., the original metamodel may have constraints that would have rejected instance models that are legally defined by the inferred metamodel).
- *Unavailability of Negative Counter Examples*: If negative examples were available, perhaps simple OCL constraints could be inferred. It is not certain, however, what degree of constraint complexity could be inferred from negative examples. To a large degree, the level of detail that can be inferred depends on the amount of available sample models, and the degree to which they differ. The concept of inferring a constraint is a non-issue, however, because negative examples typically do not exist in the modeling process. This is not a limitation of MARS, per se, but an acknowledgement of the lack of availability of such examples in the process itself.

7.3 More Information about the MARS Project

MARS is part of the GenParse project, which is focused primarily on the application of grammatical inference algorithms to grammar-based systems. Several of the listings in this paper are fragments of the complete representation. All of the extended listings (e.g., XSLT rules, DMS transformations, sample metamodels and instance models, grammars, and model interpreters) are available at the MARS website, which can be found at http://www.cis.uab.edu/softcom/GenParse/mars.htm

Bibliography

[1] *OMG's First Annual Model-Integrated Computing (MIC) Workshop, http://www.omg.org/news/meetings/mic2004/*, Arlington, VA, October 2004.

[2] *Abbadingo One: DFA Learning Competition*, http://abbadingo.cs.unm.edu.

[3] I. Baxter, C. Pidgeon, and M. Mehlich, "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004, pp. 625-634.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Third Edition)," *W3C Technical Report*, February 2004, http://www.w3.org/TR/2004/REC-xml-20040204.

[5] B. Chidlovskii, "Schema Extraction from XML Data: A Grammatical Inference Approach," *Eighth International Workshop on Knowledge Representation meets Databases*, Rome, Italy, September 2001.

[6] J. Clark, "XSL Transformations (XSLT) (Version 1)," *W3C Technical Report*, November 1999, http://www.w3.org/TR/1999/REC-xslt-19991116.

[7] J. Clark and S. DeRose, "XML Path Language (XPath) (Version 1.0)," *W3C Technical Report*, November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116.

[8] M. Crepinsek, M. Mernik, F. Javed, B. R. Bryant, and A. Sprague, "Extracting Grammar from Programs: Evolutionary Approach," *ACM SIGPLAN Notices*, 2004 - to appear.

[9] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[10] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. Shapire, and L. Sellie, "Efficient Learning of Typical Finite Automata from Random Walks," *Twenty-Fifth Annual Symposium on Theory of Computing*, San Diego, CA, May 1993, pp. 315-324.

[11] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim, "XTRACT: A System for Extracting Document Type Descriptors from XML Documents," *ACM SIGMOD International Conference on Management of Data*, Dallas, TX, May 2000, pp. 165-176.

[12] J. Garrett, Á. Lédeczi, and F. Decaria, "Toward a Paradigm for Activity Modeling," *International Conference on Systems, Man, and Cybernetics*, Nashville, TN, October 2000, pp. 2425-2430.

[13] E. M. Gold, "Language Identification in the Limit," *Information and Control*, vol. 10, no. 5, pp. 447-474.

[14] J. Gray, M. Rossi, and J-P Tolvanen, "Preface: Special Issue on Domain-Specific Modeling," *Journal of Visual Languages and Computing*, vol. 15, no. 3-4, July-August 2004, pp. 207-209.

[15] J. Gray, J. Zhang, Y. Lin, S. Roychoudhury, H. Wu, R. Sudarsan, A. Gokhale, S. Neema, F. Shi, and T. Bapty, "Model-Driven Program Transformation of a Large Avionics Framework," *Generative Programming and Component Engineering (GPCE)*, Vancouver, Canada, October 2004, pp. 361-378.

[16] L. Howard, "CAPE: A Visual Language for Courseware Authoring," *2nd Workshop on Domain-Specific Visual Languages (OOPSLA)*, Seattle, WA, October 2002. http://www.cis.uab.edu/info/OOPSLA-DSVL2/Papers/Howard.pdf.

[17] I. Jonyer, L. B. Holder, and D. J. Cook, "MDL-Based Context-Free Graph Grammar Induction and Applications," *International Journal of Artificial Intelligence Tools*, 2004, vol. 13 no. 1, pp. 65-79.

[18] H. Juille and J .B. Pollack, "A Sampling-Based Heuristic for Tree Search Applied to Grammar Induction," *Fifteenth National Conference on Artificial Intelligence*, Madison, WI, July 1998, pp. 26-30.

[19] G. Karsai, M. Maroti, Á. Lédeczi, J. Gray, and J. Sztipanovits, "Composition and Cloning in Modeling and Meta-Modeling," *IEEE Transactions on Control System Technology*, vol. 12, no. 2, March 2004, pp. 263-278.

[20] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo One DFA Learning Competition and a new Evidence-Driven State Merging Algorithm," *Fourth International Colloquium on Grammatical Inference*, Springer-Verlag LNCS 1433, Ames, IA, July 1998, pp. 1-12.

[21] K. J. Lang, "Evidence-Driven State Merging with Search," *NECI Technical Report TR98-139.* 1998.

[22] Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, vol. 34, no. 11, November 2001, pp. 44-51.

[23] E. Long, A Misra, and J. Sztipanovits, "Increasing Productivity at Saturn," *IEEE Computer*, August 1998, pp. 35-43.

[24] M. Mernik, M. Crepinšek, T. Kosar, D. Rebernak, and V. Žumer, "Grammar-Based Systems: Definition and Examples," *Informatica*, to appear, 2004.

[25] M. Mernik, J. Heering, and T. Sloane, "When and How to Develop Domain-specific Languages," *CWI Technical Report*, SEN-E0309, 2003.

[26] M. Mernik, M. Lenic, E. Avdicauševic, and V. Žumer, "LISA: An Interactive Environment for Programming Language Development," *11th International Conference on Compiler Construction*, Springer-Verlag LNCS 2304, Grenoble, France, April 2002, pp. 1-4.

[27] M. Mernik and V. Žumer, "Incremental Programming Language Development," *Computer Languages, Systems and Structures*, vol. 31 no. 1, April 2005, pp. 1-16.

[28] M. Moore, S. Monemi, and J. Wang, "Integrating Information Systems in Electrical Utilities," *International Conference on Systems, Man and Cybernetics*, Nashville, TN, October 2000, pp. 8-11.

[29] C. G. Nevill-Manning and I. H. Witten, "Compression and Explanation using Hierarchical Grammars," *The Computer Journal*, 1997, vol. 40 no. 2/3, pp. 103-116.

[30] M. Pazzani and D. Kibler, "The Utility of Knowledge in Inductive Learning," *Machine Learning*, vol. 9, 1992, pp. 57-94.

[31] J. Rissanen, "Modeling by Shortest-Data Description," *Automatica*, 1978, vol. 14, pp. 465-471.

[32] Y. Sakakibara, "Efficient Learning of Context-Free Grammar from Positive Structural Examples," *Information and Computation*, vol. 97, no. 1, March 1992, pp. 23-60.

[33] Y. Sakakibara and H. Muramatsu, "Learning Context-Free Grammars from Partially Structured Examples," *Fifth International Colloquium on Grammatical Inference and Applications*, Lisbon, Portugal, September 2000, pp. 229-240.

[34] D. Sharp, "Component-Based Product Line Development of Avionics Software," *First Software Product Lines Conference (SPLC-1)*, Denver, CO, August 2000, pp. 353-359.

[35] J. Siméon and P. Wadler, "The Essence of XML," *Symposium on Principles of Programming Languages*, New Orleans, LA, January 2003, pp. 1-13.

[36] M. Tomita, "Dynamic Construction of Finite Automata From Examples Using Hill-Climbing," *Fourth Annual Cognitive Science Conference*, Ann Arbor, MI, 1982, pp. 105-108.

[37] B. A. Trakhtenbrot and Y. M. Barzdin, *Finite Automata Behavior and Synthesis*, North-Holland Publishing Company, 1973.

[38] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 2003.

[39] N. Weiss, *Introductory Statistics*, Addison-Wesley, 2001.