USING A PROGRAM TRANSFORMATION ENGINE TO INFER TYPES IN A METAMODEL RECOVERY SYSTEM

[*]Faizan JAVED, [**]Marjan MERNIK, [*]Jeff GRAY, [†]Jing ZHANG, [*]Barrett R. BRYANT,
[*]Suman ROYCHOUDHURY

[*] Department of Computer and Information Sciences, University of Alabama at Birmingham,
115A Campbell Hall, 1300 University Boulevard, Birmingham, AL, USA, 35295-1170,
tel. (+1) 205 934 2213, E-mail: {javedf, gray, bryant, roychous}@cis.uab.edu

[**] Institute of Computer Science, Faculty of Electrical Engineering and Computer Science,
University of Maribor, Smetanova 17, 2000 Maribor, Slovenia, tel. (+3862) 220 7455, E-mail: marjan.mernik@uni-mb.si

[†] Motorola Research Labs, NIRL Autonomics Research,
1301 Algonquin Road, Schaumburg, Illinois, USA 60196 E-mail: j.zhang@motorola.com

SUMMARY

Domain-Specific Modeling (DSM) allows domain experts to concentrate on the essential characteristics of a problem space without being overwhelmed by the complexities that may occur in the solution space. DSM is focused on the creation of a metamodel for a specific domain, from which instances pertaining to specific configurations of that domain can be constructed. However, as the metamodel undergoes evolutionary changes, repositories of instance models (also called domain models) can become orphaned from their defining metamodel. Within the context of model-driven engineering (MDE), we have developed the Metamodel Recovery System (MARS) which addresses the problem of "metamodel drift" and recovers the design knowledge in a repository of legacy models. MARS is a semi-automatic system that uses grammar inference techniques to recover a metamodel by mining instance models. In addition to the instance models, there are other artifacts that can be investigated in the modeling repository. In this paper we describe an extension to MARS in the form of a type inference capability that is accomplished by the use of a program transformation engine that mines the model compiler code and recovers the type information of fields (or attributes) of metamodel entities.

Keywords: *Domain-specific modeling, grammar inference, program transformation.*

1. INDRODUCTION

Many software artifacts created during the software lifecycle (e.g., models and source code) may be stored in a repository and depend on a language schema definition that provides the context for syntactic structure. For example, in the programming language domain a context-free grammar (or grammar) defines the syntactic constructs of a programming language. Similarly in the Domain-Specific Modeling (DSM) paradigm, a model is defined by a metamodel. DSM allows a higher level of abstraction than general purpose languages (GPLs) while simultaneously narrowing the design space to a single domain of discourse with visual models [1]. DSM involves the construction of a metamodel that defines the key elements of a domain, and instances of the metamodel, called instance models (or models), represent specific configurations of the domain. To address new feature requests (e.g., adaptation of a metamodel to accomodate new stakeholder concerns or evolution of a language to provide new language features) the repository artifacts might need to be transformed to the new schema definition. If this is not done, the respository may be replete with archaic artifacts.

In the programming languages paradigm, the existence of over 500 general purpose and proprietary programming languages in commercial and public domains motivates the need to have expeditious and reliable software renovation tools. A strong case for applying a grammar-centric solution to solve software renovation problems in the programming language domain is made in [2]. These renovation tools can be used to solve re-engineering problems like recovering source implementations or translating them to a different dialect.

A rise in the use of modeling tools in industry and research [3] has resulted in an increase in the number of renovation problems in the modeling community. As a metamodel evolves, each new version captures some change in the modeling language and the instance models that are dependent on the metamodel definition need to be updated. An initial solution to this metamodel *schema evolution* problem using graph rewriting techniques is discussed in [4]. However, this schema evolution approach is not applicable when both the metamodels and the intermediate transformation steps do not exist, or are not accessible. Two example situations are: 1) losing a metamodel definition due to a hard-drive crash, and 2) encountering versioning conflicts when trying to load instance models based on obsolete metamodels. We use the term *metamodel drift* to refer to the phenomenon of frequent metamodel evolution which can result in previous model instances being orphaned from the new definition. A growing number of both commercial and research organizations have reported occurances of lost and evolved metamodels [5, 6]. When the metamodel is no longer available for an instance model, the instance model will fail to load into the modeling tool (this is similar in concept to a change in a language grammar that invalidates prior programs and the associated compiler). However, if a metamodel can be inferred from a set of instance models the design knowledge contained in the instance models can be recovered.

We have developed MARS [7], a semi-automatic grammar-driven system which uses grammar inference techniques to recover metamodels from instance models. Grammar inference [8] is the process of learning syntax from examples where the examples are sets of strings defined on a specific alphabet. MARS is able to accurately infer metamodel elements, generalizations, aggregations and connections. A current limitation of MARS is its inability to infer attribute types (or fields) of model elements from the model instances. For example, a string value associated with an attribute in an instance model could correspond to a string or an enumeration value. However, in addition to the instance models there are other artifacts (such as model compilers) that can be mined in the modeling repository. Model compilers can traverse the internal representation of a model and perform analysis and translation tasks like generating new artifacts (e.g., source code). A model compiler may contain type information that cannot be inferred from the instance models. The key challenge with mining information from a model compiler is the difficulty of parsing the model compiler source (e.g., a complex C++ program) and performing the appropriate analysis to determine the type information.

In this paper we demonstrate the use of a program transformation engine to parse the model compiler code and recover the type information of metamodel entities. We illustrate the technique on domain models from [7], where the focus was on inferring metamodels from models, and show how this new extension enables MARS to correctly infer attribute types of model elements. The rest of the paper is organized as follows: Section 2 gives an overview of MARS and applies the technique on an example domain. Section 3 elaborates on the program transformation technique for data type inference, and Section 4 is an experimental study of the new technique. Related work is covered in Section 5 and the paper concludes with a summary discussion and future work in Section 6.

2. THE METAMODEL RECOVERY SYSTEM

Figure 1 shows a metamodel for a Finite State Machine (FSM), originally presented in [7], which also will be the example used in this section. The metamodel specifies FSM concepts (e.g., start state, end state, and state) as well as the valid connections among all entities. An instance of this metamodel that shows a simple FSM composed of a start state, an end state and a connection between them is shown in Figure 2. The metamodel also contains two First Class Objects (FCOs). An FCO element facilitates better inheritance relationship design amongst model entities by providing an intermediate level of generalization. There are no fields (attributes) in this metamodel.

Although MARS uses the Generic Modeling Environment (GME) [9], its underlying principles can be applied to other modeling tools such as MetaCase's metaEdit+ (http://www.metacase.com) and Microsoft's DSL tools (http://msdn.microsoft.com/vstudio/dsltools/), amongst others. In the GME, a metamodel is described with UML class diagrams and constraints are specified in the Object Constraint Language (OCL) [10]. GME also provides an API for traversing a model and from the API its possible to create model compilers.

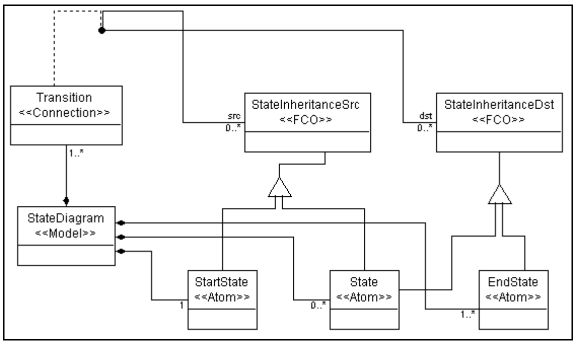


Fig. 1 A metamodel for creating finite state machines.

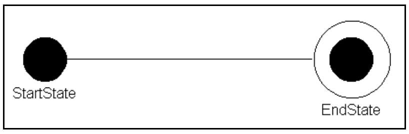


Fig. 2 An instance of a FSM

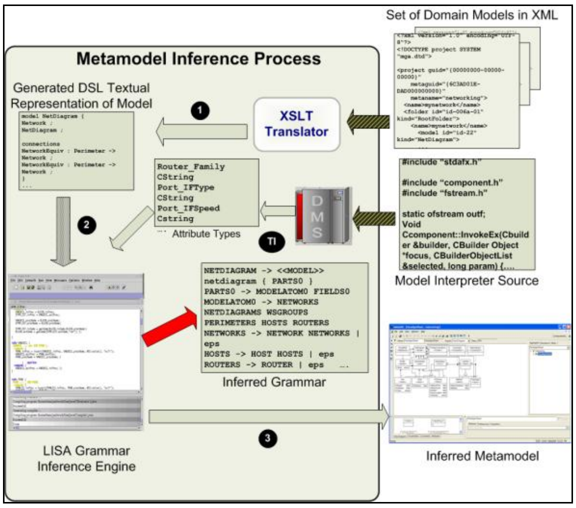
To the best of our knowledge, MARS provides a first solution to the problem of recovering metamodels from instance models. It accomplishes this by application of grammar inference algorithms from the machine learning and programming languages community to the modeling domain. An overview of MARS is shown in Figure 3, which is an extension to the architecture initially presented in

[7]. MARS has three primary steps (see steps 1, 2 and 3 in Figure 3) with an extension step labeled *TI*, which will be described in the next section. MARS takes as input a set of models exported as XML files, a capability provided by most modeling tools. However, there is a mismatch between the XML representation of a model and the syntax expected by the grammar inference tools. To overcome the mismatch in representation, MARS uses the Extensible Stylesheet Language Transformation Language XSLT [11] (step 1 in Figure 3) to map the XML files to a textual domain-specific language (DSL) [12] called the Model Representation Language (MRL), which describes the domain models in a form that can be used by a grammar inference engine. An MRL program is a textual representation of the various metamodel elements (e.g., models, atoms and connections). As an example, the MRL representation of the FSM instance model in Figure 2 would be as follows:

```
model StateDiagram {
   StartState;
   EndState;
   connection
   Transition : StartState → EndState;
}

atom StartState {  fields ;  }

atom EndState {  fields ;  }
```


Set of Domain Models in XML
Metamodel Inference Process
Generated DSL Textual Representation of Model
1
XSLT Translator
2
Attribute Types
TI
D M S
Model Interpreter Source
Inferred Grammar
3
LISA Grammar Inference Engine
Inferred Metamodel

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE project SYSTEM "mga.dtd">

<project guid="{00000000-00000-00000}"
     metaguid="{6C3AD01E-DAD000000000}"
     metaname="networking">
  <name>mynetwork</name>
  <folder id="id-006a-01" kind="RootFolder">
    <name>mynetwork</name>
      <model id="id-22" kind="NetDiagram">
      ...
```

```
model NetDiagram {
Network ;
NetDiagram ;

connections
NetworkEquiv : Perimeter ->
Network ;
NetworkEquiv : Perimeter ->
Network ;
}
...
```

```
Router_Family
CString
Port_IFType
CString
Port_IFSpeed
Cstring
...
```

```
#include "stdafx.h"

#include "component.h"
#include "fstream.h"

static ofstream outf;
Void
Ccomponent::InvokeEx(Cbuilder &builder, CBuilder Object *focus, CBuilderObjectList &selected, long param) {....
```

```
NETDIAGRAM -> <<MODEL>>
netdiagram { PARTS0 }
PARTS0 -> MODELATOM0 FIELDS0
MODELATOM0 -> NETWORKS
NETDIAGRAMS WSGROUPS
PERIMETERS HOSTS ROUTERS
NETWORKS -> NETWORK NETWORKS |
eps
HOSTS -> HOST HOSTS | eps
ROUTERS -> ROUTER | eps   ...
```

Fig. 3 Overview of MARS
(modified extension adapted from [7])

The MRL representations of the instance models are input to the metamodel inference process, which is performed within the language description environment LISA [13] (step 2 in Figure 3). The result of the inference process is a context-free grammar that is generated concurrently with the XML file containing the metamodel that can be used to load the instance models into the modeling tool (step 3 in Figure 3). For the FSM metamodel example in Figure 1, the inferred metamodel is shown in Figure 4.

If we compare the original metamodel in Figure 1 and the inferred metamodel Figure 4 we can observe that the inferred metamodel is almost exactly the same as the original metamodel except the names of the two *StateInheritance* FCOs in the original metamodel have been inferred as generic names FCO1 and FCO2. This presents no real consequence with respect to the essential capabilities as seen from an end-user's perspective. The generalization hierarchy and all the metamodel elements are inferred accurately.

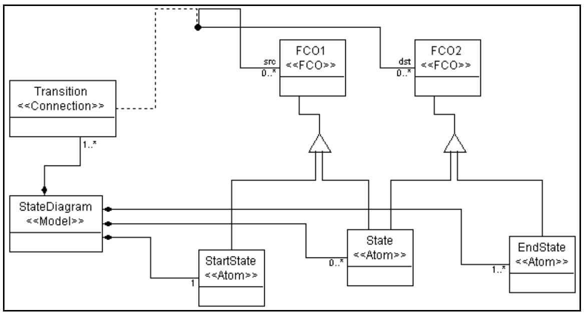


Fig. 4 The inferred metamodel for FSM.

The corresponding inferred grammar is shown below with the nonterminals in upper case letters, terminals in lower case letters and epsilon as ε.

1. STATEDIAGRAM → 'model' StateDiagram
{ PARTS0 }
2. PARTS0 → MODELATOM0 FIELDS0
CONNECTIONS0
3. MODELATOM0 → STARTSTATES
ENDSTATES STATES
4. STARTSTATES → STARTSTATE
5. ENDSTATES → ENDSTATE ENDSTATES
| ENDSTATE
6. STATES → STATE STATES | ε
7. FIELDS0 → ε
8. CONNECTIONS0 → 'connection'
TRANSITION TRANSITION →
transition : SRC0 → DST0 ;
TRANSITION | transition : SRC0 →
DST0 ;
9. SRC0 → 'fco' FCO1
10. FCO1 → STARTSTATE | STATE
11. DST0 → 'fco' FCO2
12. FCO2 → ENDSTATE | STATE
13. STARTSTATE → 'atom' StartState
{ FIELDS1 }
14. FIELDS1 → ε
15. ENDSTATE → 'atom' EndState { FIELDS2 }
16. FIELDS2 → ε
17. STATE → 'atom' State { FIELDS3 }
18. FIELDS3 → ε

The quality of the inferred metamodel depends on the total number of instance models used as well as the level of detail available in the instance models. If the set of supplied instance models do not make use of all the constitutent elements of the original metamodel or exhibit all the variations in cardinalities of the connections between the elements, then those particular elements and cardinalities cannot be inferred. For example, if the only input to MARS was the instance model in Figure 2, then it would not be possible to infer the most accurate FSM metamodel. The reason for this is that the instance model does not make use of the state element nor contains enough information to infer cadinalities of the connections accurately. We refer the reader to [7] for further details on MARS, its core algorithm, and detailed discussion on the metamodel inference results of the domain examples used in this paper.

Because the type information of fields is not available in the instance models, MARS infers all the fields as generic *field* types. As previously mentioned, model compilers may contain type information that can allow MARS to infer more complete and accurate metamodels. Recovering this type information would require the ability to parse and analyze the model compiler source (e.g., a complex C++ program). In the next section, we address this key problem by discussing the use of a program transformation tool. The Design Maintenance System (DMS) [14], to parse the model compiler code and ascertain the appropriate type information for attributes defined in the metamodel.

3. TYPE INFERENCE USING DMS

The previous section gave an overview of MARS and showed that the system is capable of inferring a metamodel from 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, consider the Network metamodel in Figure 5, which contains networking concepts (e.g., routers, hosts, and ports) as well as the valid connections among all entities (Note: This example metamodel is taken from the tutorial that is part of the GME installation). Figure 6 shows an instance of this metamodel where there is an attribute called *Port_IFSpeed* in a *Port* atom that is named *S0* (located in *inetgw*). 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 model types from existing model compilers associated with the mined instance models.

DMS is a program transformation engine and re-engineering toolkit [14]. The core component of DMS is a term rewriting engine that provides powerful pattern matching and source translation capabilities. DMS was chosen for this task because of its scalability for parsing and transforming large source files in several dozen languages (e.g., C++, Java, COBOL, Pascal). 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 tools to perform sophisticated program analysis and transformation tasks. Another consideration for the choice of DMS comes from our past success in using it to parse millions of lines of C++ code [15].

Table 1 illustrates a fragment of a GME model compiler 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 8) in the model and store it in a variable (*fam* in Line 8). The attribute name should be exactly the same as the name shown in the corresponding model because the model compiler is referencing metamodel concepts. Consequently, the type of the variable that is used in the model compiler 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 3 of the model compiler code fragment).

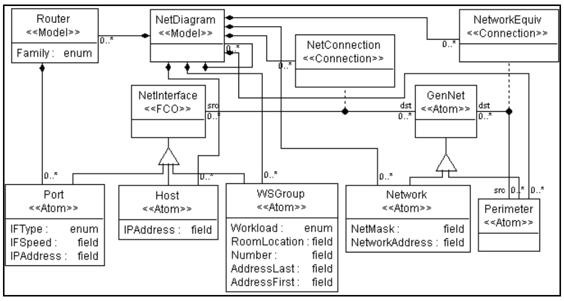


Fig. 5 A metamodel for Network diagrams

The general idea of implementing a type inference system is to set up a symbol table for the model compiler source code. A symbol table stores all of the variables along with appropriate attributes (e.g., scope of validity, type, and value). Figure 7 describes a simplified symbol table for the *ProcessRouter* method in Table 1. This symbol table contains three symbol spaces that represent three different lexical scopes: method body, block (corresponds to Lines 4 to 7 in Table 1), and a while block (corresponds to Lines 12 to 18 in Table 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.

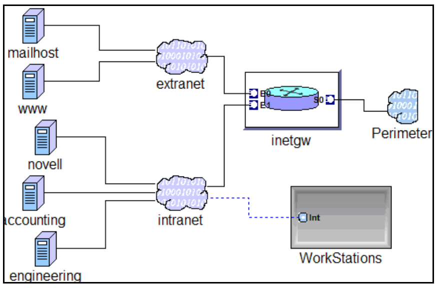
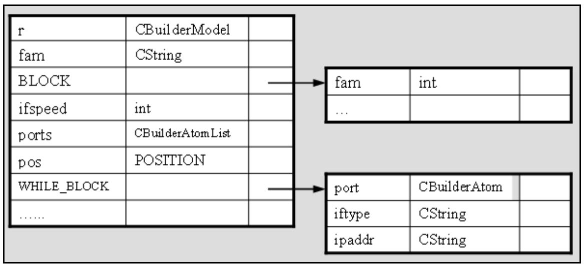


Fig. 6 An instance of a Network

```
void CComponent::ProcessRouter(CBuilderModel *r) {
  ASSERT(r->GetKindName() == "Router");
  CString fam;
  {
    int fam;
    ……
  }
  r->GetAttribute("Router_Family", fam);
  int ifspeed;
  const CBuilderAtomList *ports = r->GetAtoms("Port");
  POSITION pos = ports->GetHeadPosition();
  while(pos) {
    CBuilderAtom *port = ports->GetNext(pos);
    CString iftype, ipaddr;
    port->GetAttribute("Port_IFType", iftype);
    port->GetAttribute("Port_IFSpeed", ifspeed);
    port->GetAttribute("Port_IPAddress", ipaddr);
      ……. } }
```

Tab. 1 An excerpt from the model compiler for processing routers in the Network domain



r	CBuilderModel
fam	CString
BLOCK	
ifspeed	int
ports	CBuilderAtomList
pos	POSITION
WHILE_BLOCK	
……	

fam	int
…	

port	CBuilderAtom
iftype	CString
ipaddr	CString

Fig. 7 Symbol table for the Process Router method

After the symbol table is constructed, it can be used to discover 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 Table 2 searches the attribute variables in the model compiler 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 3) will be generated to serve as an input for step 3 of MARS.

```
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 ?))
                      …… }
```

Tab. 2 PARLANSE code fragment to determine attribute types

4. EXPERIMENTAL STUDY

In this section, we discuss the results of applying the type recovery technique to diverse domains. Due to space constraints, we only show the results of applying the technique to the Petri Net [16] modeling language and the Network modeling language introduced in Section 3. The original metamodel for the Petri Net domain is shown in Figure 8 and consists of the elements *Place* and *Transition*, as well as the connections between them. A *Place* can also hold a certain number of tokens (attribute *numberOfTokens*) and the *petriElements* have *Name* and *Description* attributes.

Figure 9 shows the inferred metamodel for the Petri Net domain which was inferred from a single instance model that was rich in information content (i.e., it uses all the elements and connections of the original metamodel). The only difference between the original and the inferred metamodels is that the *petriElements* FCO generalization hierarchy in the original metamodel is missing from the inferred metamodel. This is because generalization information is not available in instance models. Consequently, the attributes of the *petriElements FCO* (*Name* and *Description*) are inferred as attributes for *Place* and *Transition* in the inferred metamodel. The inferred metamodel for the Network domain (Figure 10) is almost the same as the original metamodel in Figure 5 except that the *NetInterface* and *GenNet* generalization names are inferred as *FCO1* and *FCO2*, respectively. This is because the names of the generalizations are not contained in instances.

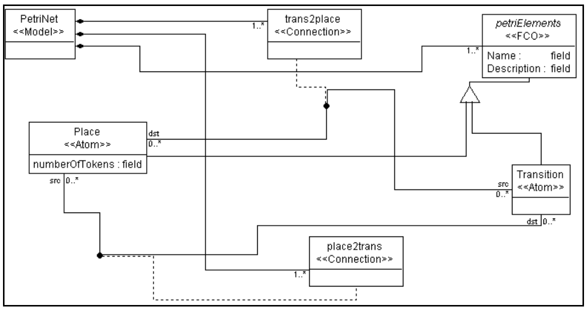


Fig. 8 Original Metamodel for the Petri Net Domain

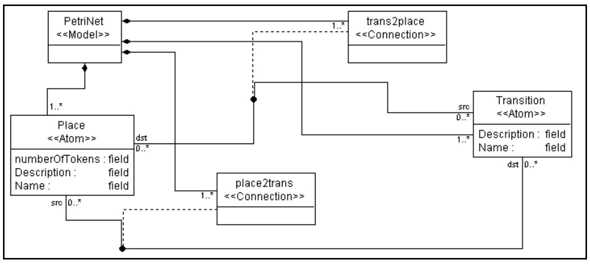


Fig. 9 Inferred metamodel for the Petri Net domain

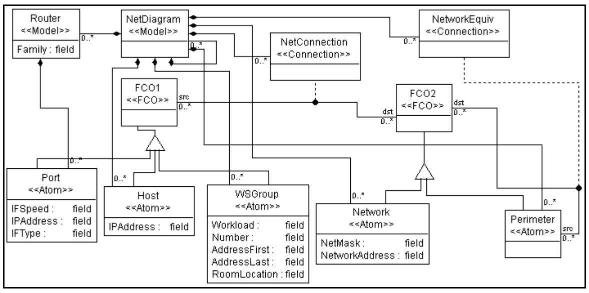


Fig. 10 Inferred metamodel for the Network domain

Results of the type inference experiments for the Petri Net and Network domains are detailed in Table 3 and Table 4, respectively. The types *Integer* and *String* are inferred as *Int* and *CString* because these are the corresponding equivalent types used by the GME model compiler. Apart from this, the only other difference is that the *Enum* type in Table 4 is inferred as a *CString* type. The reason for this is that 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 solely determined from an instance model and must involve human input. Future work will extend the type inference technique such that a user is asked to categorize a string type as a true string or as an enumeration type.

Attribute Name	Original Type	Inferred Type
numberOfTokens	Integer	Int
Name	String	CString
Description	String	CString

Tab. 3 Original and inferred types for the Petri Net domain

Attribute Name	Original Type	Inferred Type
Family	Enum	CString
IFType	Enum	CString
Workload	Enum	CString
IPAddress	String	CString
Number	Integer	Int
Netmask	String	CString
AddressFirst	String	CString
AddressLast	String	CString
NetworkAddress	String	CString
RoomLocation	String	CString

Tab. 4 Original and Inferred types for the Network domain

5. RELATED WORK

The type inference technique described in this paper is different than type inference for functional programming languages, which aims to increase programmer productivity by freeing the programmer from the task of adding type annotations while maintaining type safety. This is accomplished by algorithms that use inference rules and are partially or fully able to infer the type of a variable or an expression lacking an explicit type annotation [17]. By comparison, type inference for MARS infers (or recovers) types of model fields from a repository of model compiler source code using a program transformation engine instead of inference rules.

Our approach is more related to work on Document Type Definition (DTD) [18] and XML Schema [19] extraction. A DTD uses regular expressions to define the internal structure of an XML document. XML Schema is a grammar-based XML schema language that affords increased syntax and expressive power than DTDs and along with a host of other XML schema languages has been proposed to replace DTDs. The Microsoft XSD Inference tool [20] infers an XML Schema from well formed XML instance documents. The tool uses inference rules to infer data types as follows: the most restrictive unsigned type is inferred for attribute values when they are first encountered. If a new value is encountered that does not match the currently inferred type, a type promotion mechanism promotes the inferred type to a new type that applies to both the currently inferred type and the new value. In [21], XML Schemas are modeled as Extended Context-Free Grammars (ECFGs) and a schema extraction algorithm based on grammar inference principles is used to infer XML Schemas.

The technique initially marks all simple elements in the instances as a generic data type *Any* to simplify the inference process. After the ECFG is inferred, the simple elements are revisited and an XML Schema language data type coverage subsumption graph is used to contrain the types for each element. The XTRACT [22] system uses a regular grammar inference induction engine to infer DTDs from XML documents. The method first induces equivalent regular expressions from DTD patterns and then uses the Minimum Description Length (MDL) [23] to choose the best DTD from a group of candidate DTDs. XTRACT does not attempt to infer element types.

The work reported in [23] describes a method for extracting a logical structure from HTML files. This approach, like MARS, can be seen as a special case of grammar inference. After logical structure has been extracted, an equivalent XML file is generated. This is accomplished using three phases: visual grouping, element identification, and logical grouping. An important step in the element identification process is use of a document model, which is a kind of ontology for particular document types (e.g., personal home pages). A document model is manually prepared beforehand by careful examination of the general characteristics of such kind of documents. The main difference between MARS and this logical structure extraction technique, besides the application domain (model engineering vs. web documents), is the use of document models (defined as a grammar) for representing the knowledge of a document type. In MARS, a metamodel and the skeleton of the hierarchical structure do not exist and need to be inferred solely from examples of usage (models). Unlike MARS, this technique does not produce a grammar (in a form of XML schema) of the generated XML document.

6. CONCLUSION

MARS is a semi-automatic grammar inference based technique that addresses the metamodel drift problem [7]. The main contribution of this paper is the application of DMS, a powerful program transformation engine, to address the problem of type inference in MARS. More specifically, DMS is used to parse model compilers to recover the type information of metamodel fields. An experimental study is conducted on various metamodel domains and it is shown that the proposed type inference technique is successfully able to infer all but enum types. To overcome this limitation, the use of human intervention to disambiguate between string and enum types is proposed.

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 compilers) are available at the MARS website, which can be found at: http://www.cis.uab.edu/softcom/GenParse/mars.htm

ACKNOWLEDGEMENT

This work was supported in part by an NSF CAREER award (CCF-0643725).

REFERENCES

[1] J. Gray, J-P. Tolvanen, S. Kelly, A. Gokhale, S. Neema and J. Sprinkle. Domain-specific modeling, *Handbook on Dynamic System Modeling*, CRC Press, Boca Raton, FL, 2007, Chapter 7.

[2] R. Lämmel and C. Verhoef. Cracking the 500 language problem. *IEEE Software*, 18(6):78-88, 2001.

[3] D. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25-31, 2006.

[4] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291-307, 2004.

[5] GME Users Mailing List, http://list.isis.vanderbilt.edu/pipermail/gme-users/2005-March/000697.html [14 February 2006]

[6] GME Users Mailing List, http://list.isis.vanderbilt.edu/pipermail/gme-users/2005-March/000697.html [14 February 2006]

[7] F. Javed, M. Mernik, J. Gray, and B. R. Bryant. MARS: A MetaModel Recovery System Using Grammar Inference. Accepted for publication in *Information and Software Technology*, http://www.cis.uab.edu/softcom/GenParse/mars.htm, 2007.

[8] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[9] A. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44-51, 2001.

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

[11] J. Clark, XSL Transformations (XSLT) (Version 1). W3C Technical Report, November 1999, http://www.w3.org/TR/1999/REC-xslt-19991116 [14 February 2006].

[12] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316-344, December 2005.

[13] M. Mernik, M. Lenič, E. Avdičaušević and V.

Žumer. LISA: An interactive environment for programming language development. *The 11th International Conference on Compiler Construction*, pp. 1-4, Springer: Heidelberg, Germany, 2002.

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

[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)*, pp. 361-378, Vancouver, Canada, October 2004.

[16] J. Peterson. Petri nets. *ACM Computer Surveys*, 9 (3):223-252, 1977.

[17] B. C. Pierce. *Types and Programming Languages*, MIT Press, 2002.

[18] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Technical Report, February 2004, http://www.w3.org/TR/2004/REC-xml-20040204 [14 February 2006]

[19] J. Siméon, P. Wadler. The essence of XML. *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 1-13, ACM Press, New York NY, 2003.

[20] Microsoft XML Schema Definition Tool: http://msdn2.microsoft.com/en-us/library/x6c1kb0s(VS.80).aspx

[21] B. Chidlovskii. Schema extraction from XML data: A grammatical inference approach. *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB 2001), CEUR Workshop Proceedings*, Rome, Italy, 2001.

[22] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, K. Shim. XTRACT: Learning document type descriptors from XML document collections. *Data Mining and Knowledge Discovery*, 7(1): 23-56, 2003.

[23] M.-H. Lee, Y.-S. Kim, and K.-H. Lee. Logical structure analysis: From HTML to XML. *Computer Standards & Interfaces*, 29:109-124, 2007.

BIOGRAPHY

Faizan Javed is a Ph.D. candidate in the Department of Computer and Information Sciences at the University of Alabama at Birmingham. His research interests include grammatical inference algorithms and applications, software engineering and model-driven engineering. Faizan received an M.S. in computer science with a specialization in Bioinformatics from UAB. He is a student member of the ACM and the IEEE.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently an associate professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also an adjunct associate professor at the University of Alabama at Birmingham, Department of Computer and Information Sciences. His research interests include programming languages, compilers, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Jeff Gray is an Assistant Professor in the Department of Computer and Information Sciences at the University of Alabama at Birmingham (UAB). He received the Ph.D. in May 2002 from the Electrical Engineering and Computer Science department at Vanderbilt University. His research interests include model-driven engineering, generative programming, and aspect-oriented software development. Jeff is a member of the ACM and Senior Member of IEEE.

Jing Zhang is a research scientist at Motorola Labs, where she is responsible for conducting research on Autonomic Network Management. Jing is also a part-time PhD student in the Department of Computer and Information Sciences at the University of Alabama at Birmingham (UAB). Her PhD research is focused on techniques that combine model transformation and program transformation in order to assist in evolving large software systems. Jing obtained an M.S. in Computer Science from UAB.

Barrett R. Bryant is a Professor and Associate Chair of Computer and Information Sciences at the University of Alabama at Birmingham. He joined UAB after completing his Ph. D. in computer science at Northwestern University. His primary research areas are the theory and implementation of programming languages, formal specification and modeling, and component-based software engineering. Barrett is a member of ACM, IEEE (Senior Member), EAPLS, and the Alabama Academy of Science. He is an ACM Distinguished Lecturer and Chair of the ACM Special Interest Group on Applied Computing (SIGAPP).

Suman Roychoudhury is a Ph.D. candidate in the Computer and Information Sciences (CIS) Department at the University of Alabama at Birmingham (UAB). His research interests include aspect-oriented software development and program transformation techniques as applied to evolving large legacy systems.