Title Page

Title: MARS: A Metamodel Recovery System Using Grammar Inference

Author 1 (corresponding author): Faizan Javed, Department of Computer & Information Sciences, University of Alabama at Birmingham, 1300 University Boulevard, Birmingham, AL 35294-1170, USA

Email: javedf@cis.uab.edu

Telephone: 205-934-5841

Fax : 205-934-5473

Author 2: Marjan Mernik, Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia

Author 3: Jeff Gray, Department of Computer & Information Sciences, University of Alabama at Birmingham, Birmingham, Alabama, USA

Author 4: Barrett R. Bryant, Department of Computer & Information Sciences, University of Alabama at Birmingham, Birmingham, Alabama, USA

Abstract (100 words): Domain-specific modeling (DSM) assists subject matter experts in describing the essential characteristics of a problem in their domain. When a metamodel is lost, repositories of domain models can become orphaned from their defining metamodel. Within the purview of model-driven engineering, the ability to recover the design knowledge in a repository of legacy models is needed.

In this paper we describe MARS, a semi-automatic grammar-centric system that leverages grammar inference techniques to solve the metamodel recovery problem. The paper also contains an applicative case study, as well as experimental results from the recovery of several metamodels in diverse domains.

Keywords: Domain-specific modeling, Metamodeling, Reverse engineering, Grammar Inference

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. For example, a programming language is dependent on a grammar, and a model is defined by a metamodel; as such, both grammars and metamodels represent a schema that defines the syntax of a language. Over time, evolution of the schema definition is often required to address new feature requests (e.g., evolution of a language to provide new language features, or adaptation of a metamodel to accommodate new stakeholder concerns). If the repository artifacts are not transformed to conform to the new schema definition, it is possible that the repository may become stale with obsolete artifacts.

In the realm of programming languages, Lämmel et. al. [1] have motivated the need to fabricate tools for software renovation problems. With more than 500 general-purpose and proprietary programming languages being used in the commercial and public domains, the need for a rapid and reliable renovation tool is very real. Such a tool can be used to solve re-engineering problems, or can be useful in commercial installations where source implementations either need to be recovered or translated to a different language dialect. Lämmel et al. make a strong case for using a *grammar-centric solution* to solve these problems by stating that the dominant factor in producing any renovation tool is building a parser. When a grammar for a language can be obtained, a parser generator can be used to create a parser for the recovered language.

With the proliferation of modeling tools in the commercial and research arenas [2], the number of renovation problems in the modeling community is rising. Fig. 1 categorizes several evolution tasks that require automated assistance to manage the various dependencies among metamodels, domain models, and corresponding source code in Model-Driven Engineering (MDE). Along the top of Fig. 1 is a sequence of intermediate metamodel versions that represent the evolving definition of a specific modeling language in a particular domain. Each new version of the metamodel captures some change in the modeling language (represented by ΔMM). The domain models that are in the middle of Fig. 1 are dependent on the metamodel definition. The problem of metamodel evolution as it relates to updating dependent domain models has already been investigated. As an initial solution to the metamodel schema evolution problem, Sprinkle and Karsai define a visual language for mapping between *old* and *new* metamodels [3]. Their approach uses graph-rewriting techniques to update the domain models accordingly. However, this *schema evolution* approach fails when both the metamodels and the intermediate transformation steps do not exist, or are not accessible. A computationally efficient framework for managing and deploying incremental data transformations between models is proposed in [4]. In [5], Diskin and Dingel propose a formal algebraic framework for efficient model transformations in a generic (metamodel-independent) way. The approach avoids the laborious elementwise specification of source and target models encountered in conventional metamodel transformation methodology by specifying a mapping between source and target metamodels using generic operators. Graaf, Weber and Van Deursen [6] formulate the migration of supervisory machine control systems as a model transformation problem based on the Symphony [7] view-driven software architecture (SA) reconstruction process. CacOphoNy [8] is a generic metamodel-driven software architecture process similar to Symphony. While Symphony is confined to SA reconstruction, CacOphoNy integrates SA with MDE to create an integrated SA reconstruction approach which is more expansive and broader in scope than Symphony.

A correspondence also exists between the domain models and a legacy source code base, as shown in the bottom of Fig. 1. In order to maintain a causal connection between the domain models and the corresponding source code, a technique to synchronize model changes to existing source code is needed. An initial inquiry into model-driven program transformation is presented in [9], which generates program transformation rules from model changes. The generated transformation rules are applied to a transformation engine that parses and modifies the corresponding representation of the legacy source. To keep the illustration simplified, Fig. 1 does not consider reverse engineering of source code into a model representation. This represents another issue outside the scope of this paper, but acknowledged as an area that needs further investigation. The work by Favre et. al [10] on using the Generic Software Exploration Environment (GSEE) platform to reverse engineer large scale object-oriented and component-based systems is an example of reverse engineering source code into models. In GSEE, a metamodel for component models was designed. This metamodel is used to view reverse engineered source code as a model. Furthermore, there are other artifacts that may need to be transformed during metamodel evolution (e.g., OCL constraints, model interpreters, model transformation rules), but are not shown in Fig. 1.

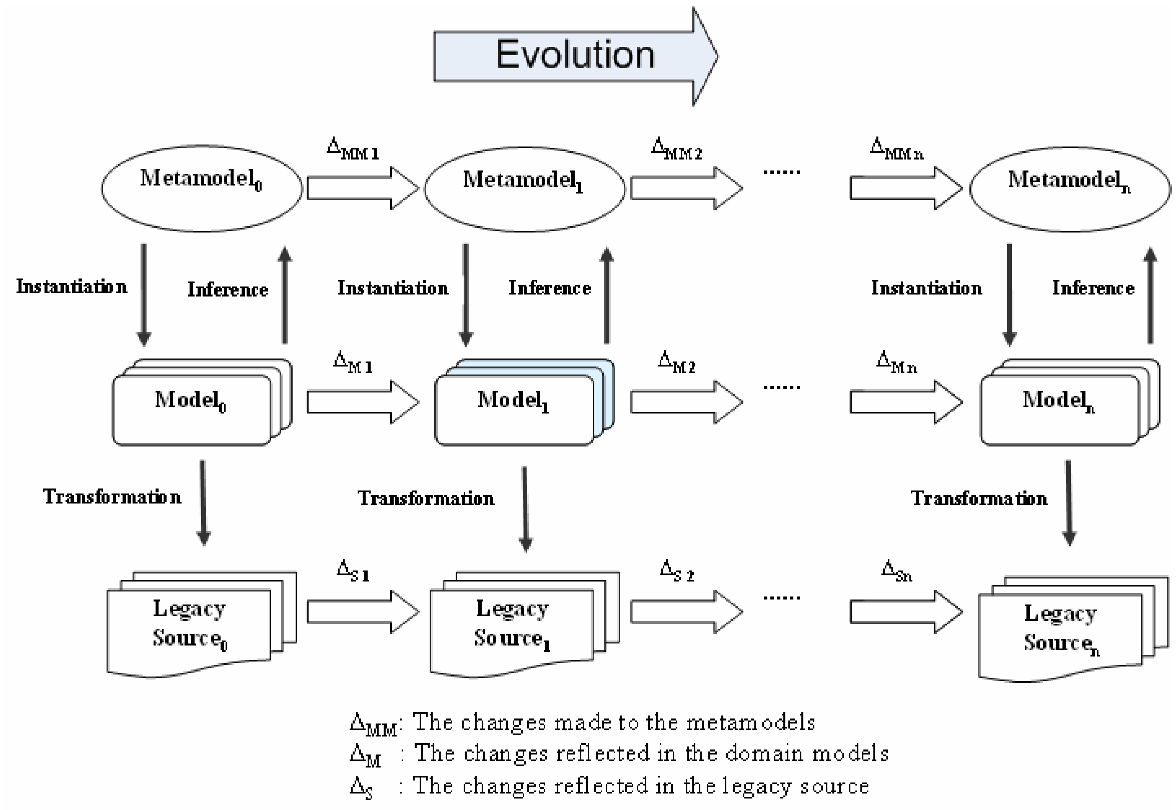


Fig. 1. An Overview of Schema Evolution Dependencies in Model-Driven Engineering

From our experience in model-driven engineering, it is often the case that a metamodel undergoes frequent evolution, which may result in previous model instances being orphaned from the new definition. This has also been observed by others in practice [11], [12]. We call this phenomenon *metamodel drift*. When the metamodel is no longer available, a domain model may not be loaded into the modeling tool (this is similar in concept to a change in a language definition that invalidates prior programs and the associated compiler). However, if a metamodel can be inferred from a set of domain models (as indicated by the "Inference" arrow in Fig. 1), the design knowledge contained in the domain models can be recovered. Some examples of problems that require the need to recover or reverse engineer a metamodel include losing a metamodel definition due to a hard-drive crash, and encountering versioning conflicts when trying to load domain models based on obsolete metamodels. In this paper, we describe a technique to recover the metamodel schema definition from instances that have been separated from their original defining metamodel. The technique is semi-automated and grammar-driven. It uses concepts from the grammar inference domain, and is applicable even when the only accessible resources are the domain models.

The paper is organized as follows: Section 2 introduces a case study used throughout the paper and also highlights the key challenges involved in metamodel inference. The section outlines the solution approach adopted within the MetAmodel Recovery System (MARS). A textual domain-specific language (DSL) is introduced in Section 3. This DSL is used as an intermediate mapping between models and the notation needed for the inference process. The heart of the paper is contained in Section 4, which documents the specific details of the metamodel inference technique used in MARS. Additional case studies are described in Section 5. Related work is compared in Section 6 and a list of current limitations that will drive future work is contained in Section 7. A brief summary serves as a conclusion in Section 8.

2. THE METAMODEL RECOVERY SYSTEM: AN OVERVIEW

Throughout the history of programming languages, abstraction has been 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 [13]. 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 syntax of the domain and the production rules of the instantiation environment.

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 Fig. 2, which represents the metamodel for a simple language for specifying properties of a Finite State Machine (FSM). The metamodel contains FSM concepts (e.g., start state, end state, and state) as well as the valid connections among all entities. An instance of this metamodel is shown in Fig. 3, which illustrates a FSM composed of a start state, a state and two end states.

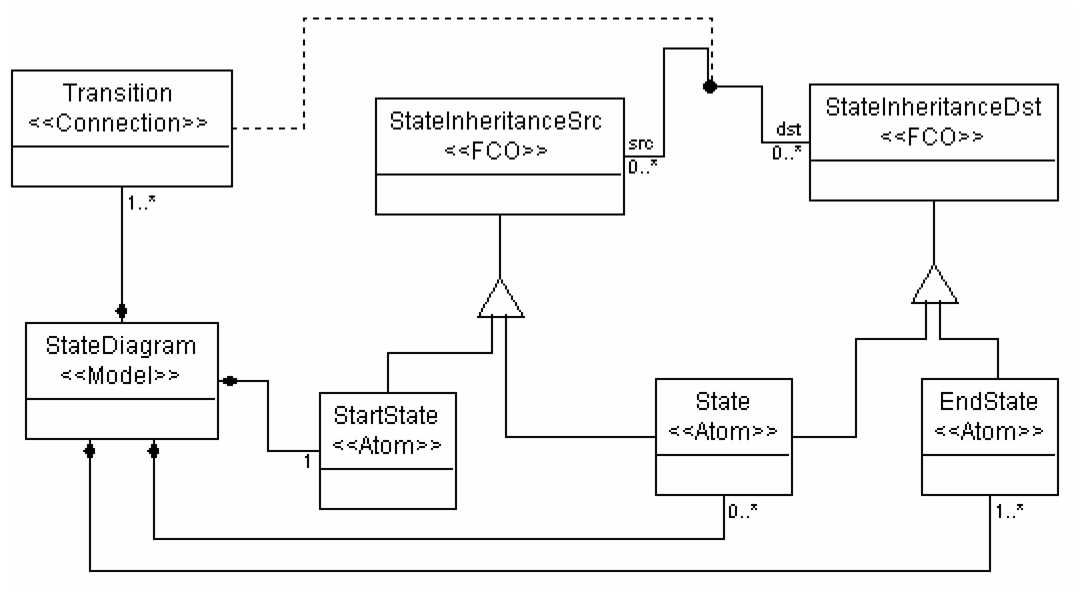


Fig. 2. A Metamodel for Creating Finite State Machines

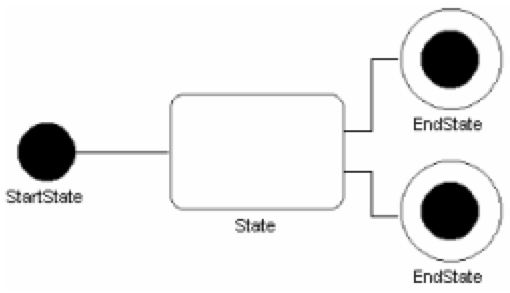


Fig. 3. An Instance of a Finite State Machine

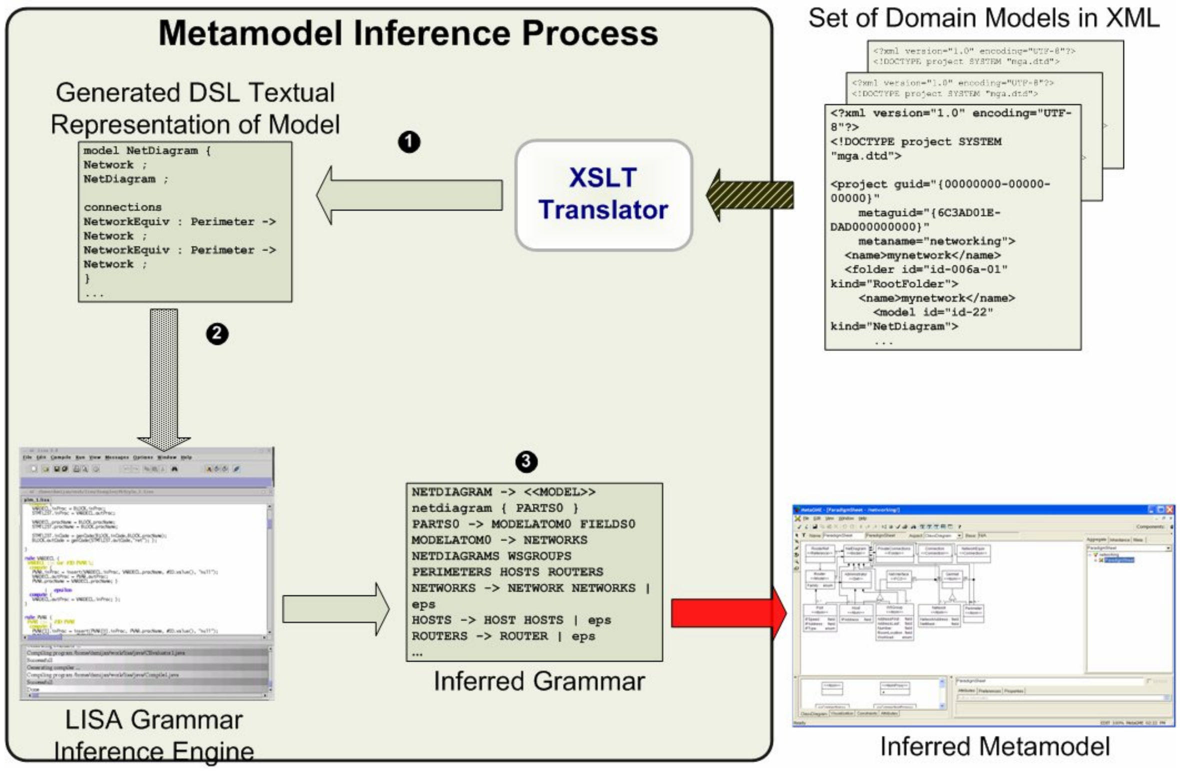
The Generic Modeling Environment (GME) [14] is the modeling tool used in our research. The GME is a metaconfigurable modeling environment that can be configured and adapted from metamodels that describe the domain [15]. When using the GME, an end-user loads a metamodel into the tool to define an environment containing all the modeling elements and valid relationships that can be constructed in a specific domain [16]. Domain models are stored in the GME as objects in a database repository. An API is provided by GME for traversing a model. From the API, it is possible to create model compilers 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) [17]. The metamodel shown in Fig. 2 represents the example to be used throughout this paper to demonstrate metamodel inference from individual instances.

There are several challenges involved in mining a set of domain models in order to recover the metamodel. The key challenges and a summary of the solutions presented in this paper are as follows:

1. *Inference Techniques for Domain Models*: The research literature in the modeling community has not addressed the issue of recovering a metamodel from a set of domain 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. Inductive learning is the process of learning from examples [18]. 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 (i.e., the set of strings belonging to the target language) and negative samples (i.e., 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 proven that exact identification in the limit of any of the four classes of languages in the Chomsky hierarchy is NP-complete [19], and in [20] it is shown that the average case is polynomial. A key contribution of this paper is the use of grammar inference techniques applied 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 instance of a domain model into a textual domain-specific language that can be analyzed by traditional grammar inference tools.

To address these two challenges, we have created MARS, which is a metamodel recovery tool based on the foundational research in grammar inference. An overview of MARS is illustrated in Fig. 4. The metamodel inference process begins with the translation of various domain models into a DSL that filters the accidental complexities of the XML representation in order to capture the essence of the domain models (represented as step 1 in Fig. 4). The inference is performed within the LISA [21] language description environment (step 2 in Fig. 4). LISA 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. LISA was chosen for this project because of its benefits in designing DSLs [22]. Moreover, the LISA system has been used successfully in our evolutionary-based context-free grammar (CFG) inference engine [23]. The result of the inference process is a context-free grammar that is generated concurrently with the XML file containing the metamodel, which can be used to load the domain models into the modeling tool (step 3 in Fig. 4). The next section will introduce the generated DSL from the model instances, as indicated by step 1 in Fig. 4.


Metamodel Inference Process
Generated DSL Textual Representation of Model
1
XSLT Translator
Set of Domain Models in XML
2
3
LISA Grammar Inference Engine
Inferred Grammar
Inferred Metamodel

```
model NetDiagram {
Network ;
NetDiagram ;

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

```
<?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">
      ...
```

```
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. 4. Overview of MARS

3. THE MODEL REPRESENTATION LANGUAGE

A grammar-based system is 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*" [24]. One of the identified benefits of a grammar-based solution is that some problems can be solved simply by converting the representation at hand to a CFG, because appropriate tools and methods for working with CFGs already exist.

A difficulty in inferring metamodels is the mismatch in notations. Each GME domain 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 to make use of already existing inference tools and techniques, a DSL was created called the *Model Representation Language* (MRL). The defining properties of DSLs are that they are small, more focused than General-Purpose Programming Languages (GPLs), and usually declarative [25]. Although DSL development is challenging, and requires domain knowledge and language development expertise, language development toolkits exist to facilitate the DSL implementation process. LISA was used to develop the MRL and supporting tools.

The primary use of the MRL 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 a model, atom, field or connection. 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 (attribute) and connection declarations. The complete grammar for the MRL is given in Table 1. Due to the fact that connections in GME are binary relationships [14], the MRL has only binary relationships. Metamodels are more expressive than a CFG in that they can model references and explicit aggregation and composition relationships [26] [27], while CFGs need additional annotations to be able to express them. Analyzing CFGs along with their annotations to discover such relationships can result in increased processing time. In the case of the MRL, all aggregations and compositions are modeled as compositions. The MRL is able to handle references by utilizing transformation rules that exactly capture such references and which can be seen as annotations to the base grammar. For example, rule 2 in Table 3 allows exact identification of the source and destination of a connection. The binary relationship inherent in the MRL results in a concise grammar that expedites parsing of MRL programs. Thus, MRL is sufficiently succinct in its design to be feasible for the inference process, but also expressive enough to represent the GME metamodels accurately. Adopting MARS within a modeling tool other than the GME would require an MRL grammar definition for that tool, but such a grammar is very simple to produce if the meta-metamodel of the modeling tool is well-understood. The MRL grammar would also need to be adapted to handle n-ary relationships if the tool supports them.

Table 1
Context-Free Grammar for MRL

```
START  ::= GME
GME    ::= MODEL_OR_ATOM  {MODEL_OR_ATOM}
MODEL_OR_ATOM ::= MODEL | ATOM
MODEL  ::= model #Id \{ M_BODY \}
M_BODY ::= [MODELS] FIELDS [connection CONNECT]
MODELS ::= #Id \; {#Id \;}
FIELDS ::= fields {#Id \,} \;
CONNECT::= {#Id \: #Id \-> #Id \;}
ATOM   ::= atom #Id \{ FIELDS \}
```

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) [28]. XSLT is a transformation language that can transform XML documents to any other text-based format. XSLT uses the XML Path Language (XPath) [29] to locate specific nodes in an XML document. XPath is a string-based language of expressions. We use XSLT and XPath to parse the XML-based model files and convert the model information in the XML file into an intermediate but equivalent representation in the form of MRL. The domain models shown in Fig. 5, which are based on the metamodel of Fig. 2, are represented as the MRL programs in Table 2. For example, model 1 is composed of a `StateDiagram` model and two `State` atoms. A `StateDiagram` consists of one `StartState`, `EndState` and a connection named `Transition` that connects both states. The `Start` states in Fig. 5 are atoms with no fields and are transformed to `atom StartState { fields; }`. Section 4.3 describes the transformation between the two representations in more detail. The primary artifact from which a metamodel will be inferred is the intermediate textual representation as translated in the MRL. The MRL serves as input into the next stage of the MARS metamodel recovery process, as described in Section 4.

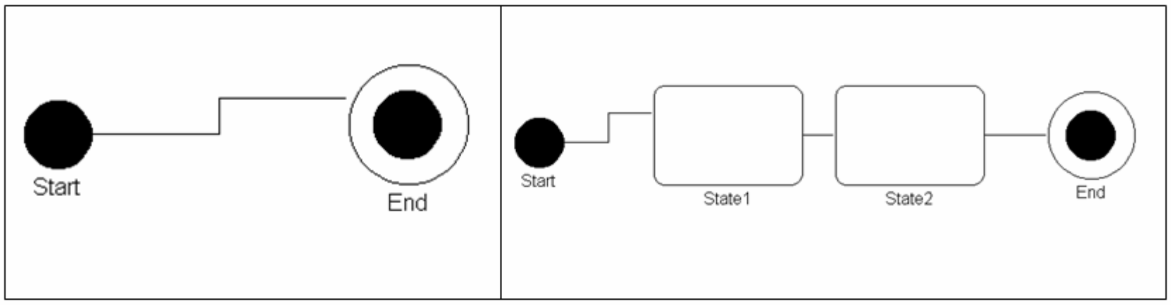


Fig. 5. Two Instances of the FSM Metamodel

Table 2
Two MRL Programs as Representations of Models from Fig. 5

Model 1:	Model 2:
`model StateDiagram {`	`model StateDiagram {`
` StartState;`	` StartState;`
` EndState;`	` EndState;`
` fields;`	` State;`
` connection`	` State;`
`   Transition : StartState → EndState;`	` fields;`
`}`	` connection`
	`   Transition : StartState → State;`
`atom StartState {`	`   Transition : State → State;`
`  fields ;`	`   Transition : State → EndState;`
`}`	`}`
`atom EndState {`	`atom StartState {`
`   fields ;`	`  fields ;`
`}`	`}`
	`atom EndState {`
	`  fields ;`
	`}`
	`atom State {`
	`  fields ;`
	`}`
	`atom State {`
	`  fields ;`
	`}`

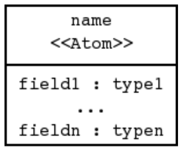
4. METAMODEL INFERENCE ENGINE

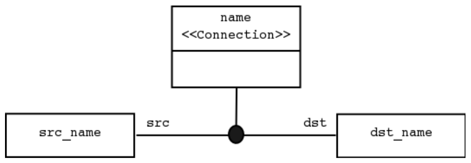
As previously discussed in Section 1, a correspondence exists between a metamodel and its instances, and a programming language and valid programs defined by the language. By considering a metamodel as a representation of a CFG, named G, the corresponding domain models can be delineated as sentences 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.

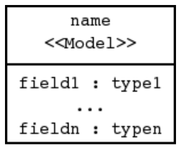
4.1 Metamodels as Context-Free Grammars

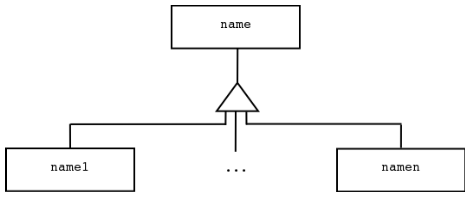
In the next step toward defining a process to infer a metamodel, all of the relationships and transformation procedures between metamodels and CFGs are identified. A key part of the process involves the mapping from the metamodel representation to the non-terminals and terminals of a corresponding grammar. 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 CFGs. This is also true for the GME metamodels [16], 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 3. Note that the type information of fields is not converted into the CFG representation because this information is not available in the domain models. MARS infers all the fields as generic 'field' types. The user can modify the field information manually after loading the inferred metamodel.

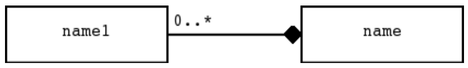
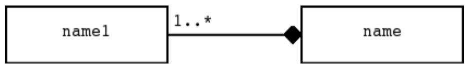
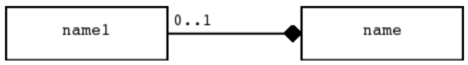
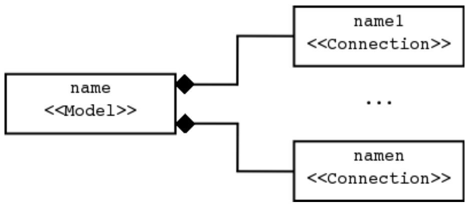
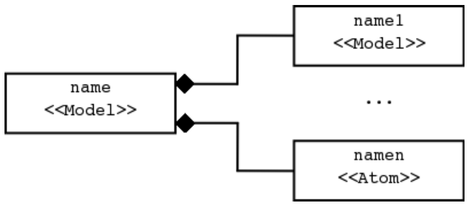
Table 3
Transformation from a Metamodel to a Context-Free Grammar

1.	 name
	<<Atom>>
	field1 : type1
	...
	fieldn : typen
		`NAME   → 'atom' name {FIELDS}`
		`FIELDS → 'fields' field1 ... fieldn`

2.	 name
	<<Connection>>
	src_name
	src
	dst
	dst_name
		`NAME → 'connection' name ':' SRC -> DST;`
		`SRC  → SRC_NAME`
		`DST  → DST_NAME`

3.	 name
	<<Model>>
	field1 : type1
	...
	fieldn : typen
		`NAME        → 'model' name {PARTS}`
		`PARTS       → MODELATOM FIELDS CONNECTIONS`
		`FIELDS      → 'fields' field1 ... fieldn`
		`MODELATOM   →  ...`
		`CONNECTIONS →  ...`
		`(see transformations 8 and 9)`

4.		`FCO → 'fco' NAME`
		`NAME → NAME1 | ... | NAMEn`

5.		NAME → NAME1S NAME1S → NAME1 NAME1S | ε
6.		NAME → NAME1S NAME1S → NAME1 NAME1S | NAME1
7.		NAME → NAME1S NAME1S → NAME1 | ε
8.	 name1 <<Connection>> name <<Model>> ... namen <<Connection>>	CONNECTIONS → NAME1 ... NAMEn (see transformation 3)
9.	 name1 <<Model>> name <<Model>> ... namen <<Atom>>	MODELATOM → NAME1 ... NAMEn (see transformation 3)

As an example application of the transformations in Table 3, the Finite State Machine (FSM) metamodel shown in Fig. 2 is semantically equivalent to the corresponding CFG represented in Table 4. The obtained CFG is a rather intuitive representation of the metamodel in Fig. 2. Productions 1 and 2 state that a `StateDiagram` (or FSM) is a model consisting of models, atoms, fields and connections. Models and atoms (production 3) that can be used in a FSM are `StartState`, `EndState`, and `State`. A FSM has no fields (production 6) and at least one connection called `Transition` (production 8). The source of the connections can be `StartState` or `State` (productions 10 and 12), and the destination can be `EndState`, or `State` (production 11 and 13), which all are atoms without fields (productions 14 – 19). From such a description, a metamodel can be drawn manually using the GME tool. However, this manual process can be automated. In the final step of the MARS inference system, the CFG is transformed to the GME metamodel XML representation that can be loaded by GME. This is accomplished by using transformation rules already presented in Table 3.

Table 4
Context-Free Grammar Representation of the Metamodel in Fig. 2

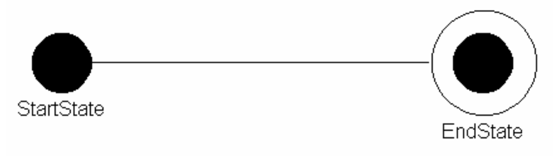
```
1.  STATEDIAGRAM → 'model' StateDiagram { PARTS0 }
2.  PARTS0 → MODELATOM0 FIELDS0 CONNECTIONS0
3.  MODELATOM0 → STARTSTATE STATES ENDSTATES
4.  STATES → STATE STATES | ε
5.  ENDSTATES → ENDSTATE ENDSTATES | ENDSTATE
6.  FIELDS0 →  ε
7.  CONNECTIONS0 → TRANSITIONS
8.  TRANSITIONS → TRANSITION TRANSITIONS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

4.2 Inferring the Metamodel from Domain Models

Before formally explaining the metamodel inference engine in the next section, we first describe the inference process from a less formal viewpoint. The working example is the FSM metamodel introduced earlier in Fig. 2. The metamodel contains FSM concepts (e.g., start state, states, end states) as well as valid connections among all entities. The metamodel also contains two FCO (First Class Object) elements by the names of `StateInheritanceSrc` and `StateInheritanceDst`. The purpose of an FCO element is to serve as an intermediate generalization that helps to better organize the inheritance relationships among model entities. In the case of the FSM metamodel, the `StartState` and `State` atom elements are generalized to the `StateInheritanceSrc` FCO and the `EndState` and `State` atom elements are generalized to a `StateInheritanceDst` FCO. In the GME, model elements can contain other models or atoms, but atoms are primitives that cannot contain other elements. The FSM metamodel states that valid instances of the metamodel must have exactly one start state, zero or more states, one or more end states, and at least one transition. All valid instances of a metamodel have to be compliant with the UML class diagram that defines the metamodel.

4.2.1 Incrementally Inferring a Metamodel

Assume that the metamodel is lost and we have to recover the metamodel only from available instances. The first sample domain model and corresponding MRL program is shown in Fig. 6. The FSM is composed of a start state, end state and a transition going from the start state to the end state.



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

Fig. 6. The First Domain Model and Corresponding MRL Program

Because the domain model exposes only one start state, one end state and one transition, the inferred metamodel (Fig. 7) is only an approximation of the original metamodel (Fig. 2).

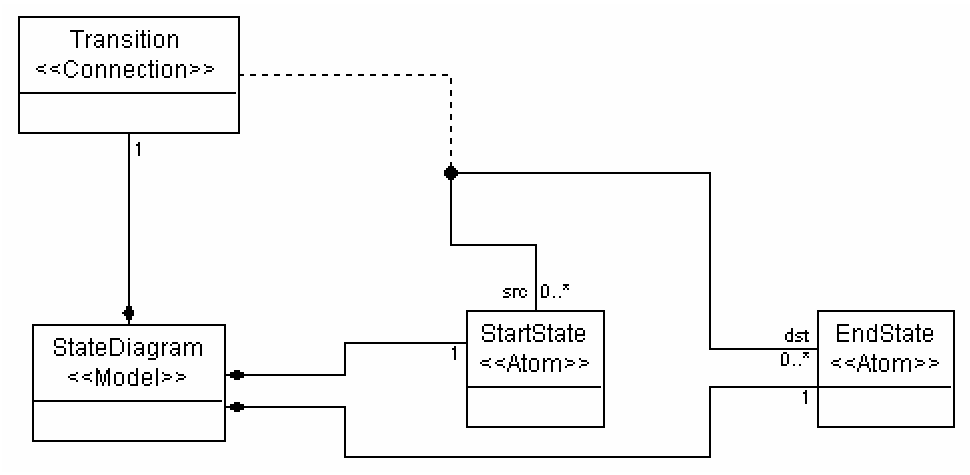
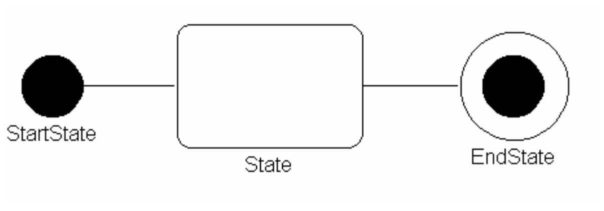


Fig. 7. Inferred Metamodel Based on the First Domain Model

Consider a second sample domain model shown in Fig. 8. In this case, the domain model is richer in detail than the previous domain model. The two transitions indicate that the FSM metamodel can have one or more transitions. This generalization is shown in the metamodel in Fig. 9 as the cardinality 1..* under transition. Moreover, source and destination of transitions can now be inferred more accurately. The domain model from Fig. 8 shows that the source of a transition can be `StartState` or `State`, and the destination can be `State` or `EndState`. All of these additional details allow inference of a more accurate metamodel (Fig. 9). However, both models (Fig. 6 and Fig. 8) are needed to infer the original metamodel accurately. The first domain model indicates the possibility that a FSM can only have a start and end state, and the second domain model shows that intermediate states are allowed. Therefore, a more accurate metamodel can be inferred if both domain models can be used (Fig. 10).



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

Fig. 8. The Second Domain Model and Corresponding MRL Program

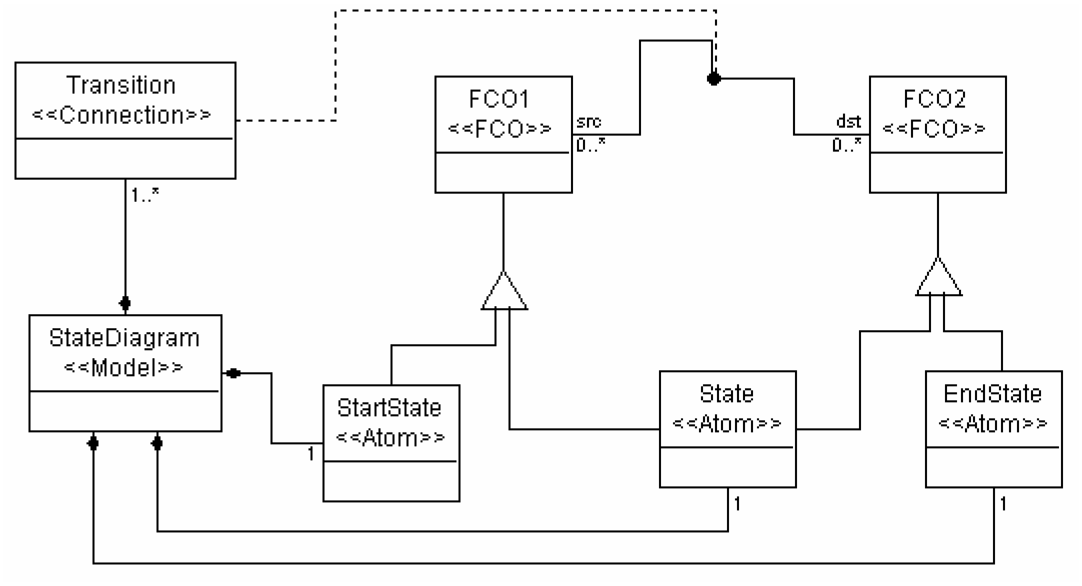


Fig. 9. Inferred Metamodel Based on Second Domain Model

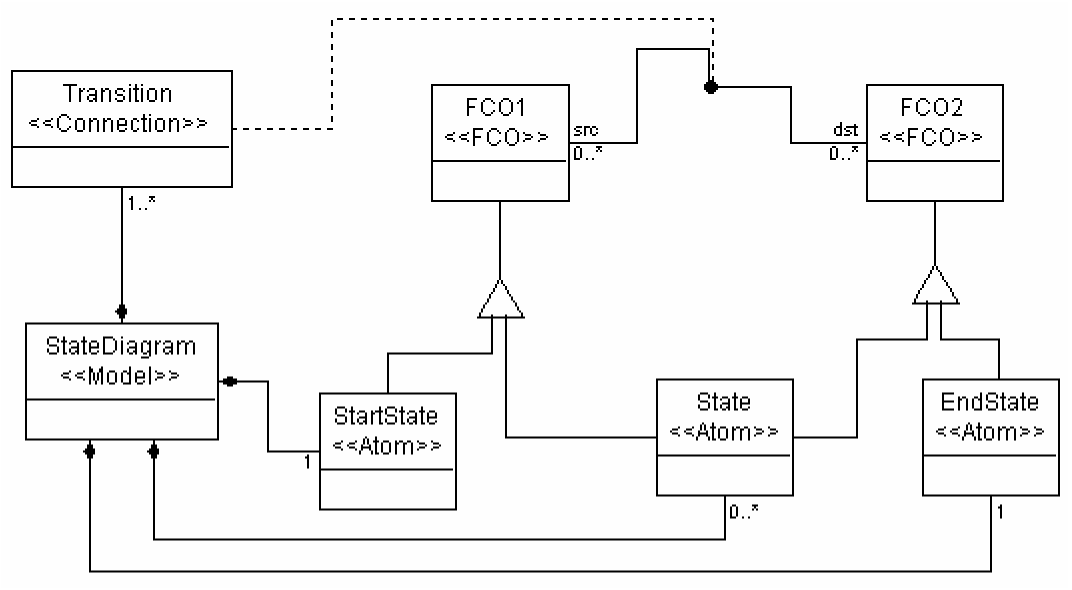
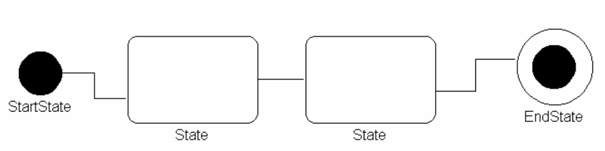


Fig. 10. Inferred Metamodel Based on First and Second Domain Models

To infer the original metamodel, more domain models are needed that exhibit other possible FSMs. For example, the domain model in Fig. 11 is an example of a FSM with many states. The domain model of Fig. 12 is an example of a FSM with many end states.



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

atom StartState { fields ; }
atom State      { fields ; }
atom State      { fields ; }
atom EndState   { fields ; }
```

Fig. 11. The Third Domain Model and Corresponding MRL Program

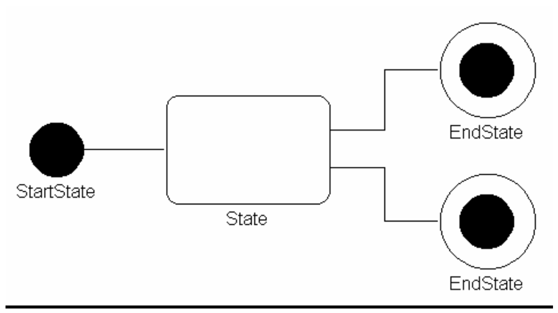
	
	```
	model StateDiagram {
	  StartState ;
	  State ;
	  EndState ;
	  EndState ;
	  fields ;
	  connection
	   Transition : StartState → State ;
	   Transition : State → EndState ;
	   Transition : State → EndState ;
	}
	atom StartState { fields ; }
	atom State      { fields ; }
	atom EndState   { fields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

Fig. 12. The Fourth Domain Model and Corresponding MRL Program

When all four domain models (Fig. 6, Fig. 8, Fig. 11, Fig. 12) are used in the inference process, the inferred metamodel (Fig. 13) and the inferred CFG (Table 5) exploit commonalities (e.g., all domain models have exactly one start state) and variabilities (e.g., zero or more states, one or more end states, one or more transitions) of all domain models. A comparison between the original metamodel (Fig. 2) and the inferred metamodel (Fig. 13) reveals that the inferred metamodel is almost exactly the same as the original metamodel except for the fact that 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.

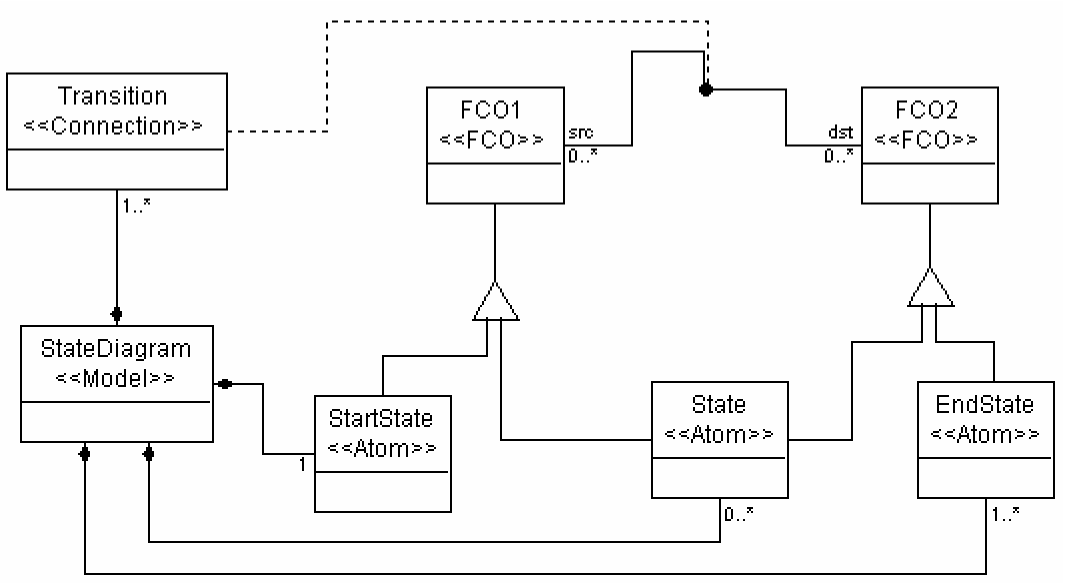


Fig. 13. Inferred Metamodel Based on Four Instances

Table 5
Inferred CFG for the FSM Metamodel

```
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
9.  TRANSITION → transition : SRC0 → DST0 ; TRANSITION | transition : SRC0 →
    DST0 ;
10. SRC0 → 'fco' FCO1
11. FCO1 → STARTSTATE|STATE
12. DST0 → 'fco' FCO2
13. FCO2 → ENDSTATE|STATE
14. STARTSTATE → 'atom' StartState { FIELDS1 }
15. FIELDS1 → ε
16. ENDSTATE → 'atom' EndState { FIELDS2 }
17. FIELDS2 → ε
18. STATE → 'atom' State { FIELDS3 }
19. FIELDS3 → ε
```

The quality of the inferred metamodel depends on the level of detail available in the domain models as well as the total number of domain models used. If the set of domain models used do not make use of all the constituent elements of the original metamodel, then those particular elements cannot be inferred. As already mentioned, we can represent the metamodels as CFGs and various instantiations of this CFG correspond to CFG representations of domain models. For experimental purposes, when the CFG for the original metamodel is available, we can enumerate all possible instantiations of the CFG to obtain all the domain models defined by that CFG. Because the language that the CFG describes could be infinite, it is not feasible to enumerate all the possible CFG instantiations. A key observation here is that to generate all possible domain models, all possible combinations of the productions in the CFG need to be enumerated. If the CFG includes recursive rules, then these rules need to be exercised only once because multiple runs through the rules will not append any new meaning to the domain models. Taking this into account, we can calculate exactly how many different domain models need to be used in the inference process. The calculation is similar to the variability calculation of feature diagrams [30], with additional rules to handle recursion.

4.3 The Induction Process

Because existing CFG inference algorithms are successful only at inducing small programming languages, the MRL 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. Because we were previously successful in inferring DSLs of a similar size using the evolutionary-based CFG inference engine [23], the same inference system was applied to induce the appropriate CFGs for the MRL. 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 an evolutionary cycle is completed, mutation, crossover and heuristic operators are applied on randomly chosen survived CFGs. Through many generations, CFGs evolve in such a manner that they correctly recognize all positive sentences and reject all negative sentences. 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 CFG inferred by the evolutionary-based inference engine was greatly generalized. The inferred grammar had 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 [19]. However, this limitation can be overcome in our case because additional knowledge about the format of GME metamodels is used. For MARS, a new inference algorithm resembling type inference algorithms (e.g., the Hindley-Milner algorithm, which always infers the most general type [31]) was invented. In a similar manner to type inference algorithms, grammar productions can be inferred from MRL descriptions of models. The algorithm has heuristics (discussed below) that allow it to generalize beyond the information contained in the domain models (a key characteristic of grammar inference algorithms). MRL programs consist of atom and model descriptions. An atom is described in MRL as:

```
atom name {
   fields f1, …, fm;
}
```

For every atom description in MRL, the following production is inferred (if a production with a left-hand non-terminal NAME does not already exist in grammar G):

```
NAME    → 'atom' name { FIELDS }
FIELDS  → 'fields' f1 f2 … fm
```

If a production with a left-hand non-terminal NAME exists in grammar G, then no changes in grammar G need to be made. Note that fields in the same atom cannot change.

A model is described in MRL as:

```
model name {
   name1 ; … ; namek ;
   fields f1, …, fm;
   connection
   con-name1: src1 → dst1 ;
   …
   con-namen: srcn → dstn ;
}
```

Note that the name of fields f_1, …, f_m must be distinct. For every model description in MRL, the following production is inferred (if a production with a left-hand non-terminal NAME does not already exist in grammar G):

```
NAME         → 'model' name { PARTS }
```

In this case, the following productions are also added to grammar G:

```
PARTS        → MODELATOM FIELDS CONNECTIONS
MODELATOM    → NAME1 NAME2 … NAMEk
FIELDS       → 'fields' f1 f2 … fm
CONNECTIONS  → 'connection' CON-NAME1 CON-NAME2 … CON-NAMEn
CON-NAMEi    → con-namei ':' SRCi '->' DSTi ';'
```

The preceding examples assumed that a connection has only one source and destination. This assumption will be relaxed later when needed. However, in this case more productions (FCO elements) will be generated that are not needed in the case of a single source or destination. In the MODELATOM production above, it can be assumed that $NAME_1$, …, $NAME_k$ are distinct. If some of the names are not distinct, then repetitions of these non-terminals will be inferred. If it is the case that NAME$_1$ = NAME$_k$, then the following productions replace the previously inferred productions:

```
MODELATOM    → NAMES_1 NAME_2 … NAME_{k-1}
NAMES_1      → NAME_1 NAMES_1 | NAME_1  //repetition of one or more
```

Moreover, the same is true for connections CON-NAME$_1$, …, CON-NAME$_n$. In the case where CON-NAME$_1$ = CON-NAME$_n$, the following productions replace previously inferred productions:

```
CONNECTIONS → 'connection' CON-NAMES_1 CON-NAME_2 … CON-NAME_{n-1}
CON-NAMES_1 → CON-NAME_1 CON-NAMES_1 | CON-NAME_1  //repetition
CON-NAME_1  → con-name_1 ':' SRC-FCO '->' DST-FCO ';'
SRC-FCO     → 'fco' SRC
DST-FCO     → 'fco' DST
SRC         → SRC_1 | SRC_n
DST         → DST_1 | DST_n
```

Although the name of the connections is the same, the source and destination of these connections can be different. To handle this, alternative productions for source and destination are inferred.

If a production with a left-hand non-terminal NAME exists in grammar G, such as:

```
NAME → 'model' name { PARTS }
```

then this situation denotes the case where several possible different instances of a metamodel exist. The difference between the previous and the new instance needs to be identified and can be in the MODELATOM part (e.g., a model or atom is missing or added) or in the CONNECTIONS part (e.g., a connection is missing or added, or the source or destination is not the same as in the previous instance).

As just described, for all models or atoms represented by non-terminals NAME$_i$ that do not appear in both instances, the inference process is different. Assume that NAME$_1$ is such a non-terminal; two cases are possible:

a) A non-terminal that represents a model or an atom is optional in the model description

```
MODELATOM    → NAMEOPT_1 NAME_2 … NAME_k
NAMEOPT_1    → NAME_1 | ε        //option
```

b) A non-terminal that represents a model or an atom can appear zero or more times in the model description

```
MODELATOM    → NAMES_1 NAME_2 … NAME_k
NAMES_1      → NAME_1 NAMES_1 | ε  //repetition of zero or more
```

The inference process is different for the case where connections represented by non-terminals CON-NAME$_i$ do not appear in both instances. Consider the following two cases (assuming that CON-NAME1 is such a non-terminal):

a) A non-terminal that represents a connection is optional in the model description:

```
CONNECTIONS    → 'connection' CON-NAMEOPT_1 CON-NAME_2 … CON-NAME_{n-1}
```

```
CON-NAMEOPT1    → CON-NAME1 | ε   //option
```

b) A non-terminal that represents a connection can appear zero or more times in the model description:

```
CONNECTIONS    → 'connection' CON-NAMES1 CON-NAME2 … CON-NAMEn-1
CON-NAMES1     → CON-NAME1 CON-NAMES1 | ε //repetition of zero or more
```

Finally, the last difference between instances is when the connection source or destination is not the same as in the previous description. To handle this case, all differences in source and destination are identified and the following productions are inferred:

```
CON-NAMEi    → con-namei ':' SRC-FCO '->' DST-FCO ';'
SRC-FCO      → 'fco' SRC
DST-FCO      → 'fco' DST
SRC          → SRC1 | … | SRCp
DST          → DST1 | … | DSTp
```

For example, in the MRL description of “Model 2” in Table 2 there are three connections, as shown below:

```
Transition : StartState → State ;
Transition : State → State ;
Transition : State → EndState ;
```

All of these connections have the same name (`Transition`). However, the source and destinations are different and the following productions are inferred:

```
CONNECTIONS → 'connection' TRANSITIONS
TRANSITIONS → TRANSITION TRANSITIONS | TRANSITION
TRANSITION  → transition ':' SRC-FCO '->' DST-FCO ';'
SRC-FCO     → 'fco' SRC
DST-FCO     → 'fco' DST
SRC         → STARTSTATE | STATE
DST         → ENDSTATE | STATE
```

Note that cardinality information can be inferred if enough model examples are available. It is not possible to generalize the cardinality value from a collection of domain 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...*).

Table 6

The Metamodel Inference Algorithm

```
Input: Vectors containing contents of domain models
Output: The inferred metamodel in CFG and XML formats

1.  for all models in Model_Vector
2.      process_ model
3.      for all model subitems Model_Subitem_Vector
4.          for all vectors in Cardinality_Vector
5.              for all subitems in Cardinality_Vector_items
6.                  compute_cardinality_of_subitem
7.          process subitem

8.      if Model_Fields_Vector is not empty
9.          for all fields in Model_Fields_Vector
10.             process model_fields

11.     if Model_Connections_Vector is not empty
12.         for all connections in Model_Connections_Vector
13.             for all connections in Connections_Cardinality_Vector
14.                 compute_cardinality_of_connection_name
15.             process_connection_name
16.             if connection has more than 1 source
17.                 check_for_fco
18.                 if fco exists
19.                     process_source_with_fco
20.                 else
21.                     process_source_with_new_fco
22.             else
23.                 process_source

24.             if connection has more than 1 destination
25.                 check_for_fco
26.                 if fco exists
27.                     process_destination_with_fco
28.                 else
29.                     process_destination_with_new_fco
30.             else
31.                 process_destination

32. for all atoms in Atom_Vector
33.     process_ atom
34.         if Atom_Fields_Vector is not empty
35.             for all fields in Atom_Fields_Vector
36.                 process atom_fields
```

The grammar inference process is performed during the parsing of MRL models using LISA. The inferred CFG for the FSM example is shown in Table 5, which is quite similar to the manually created CFG shown previously in Table 4. Table 6 details the metamodel inference algorithm and we now show that the algorithm achieves (worst case) polynomial time in complexity. The input to the algorithm are vectors that contain the constituent elements (models, atoms, connections, fields) of the domain models and are bounded by N, the total number of elements in the instance set. The *process* action in lines 2, 7, 10, 15, 19, 21, 23, 27, 29, 31, 33 and 36 represents a group of statements that output the corresponding model element in CFG and XML format and runs in linear time. Lines 1-31 compute model definitions and are responsible for the bulk of the computational complexity of the algorithm. Lines 3-7 compute subitems (child atoms and models) of models by iterating over the cardinality vectors. The cardinality computation step in line 6 uses conditional tests for specific cardinality conditions, which takes linear time. Thus, the complexity of inferring subitems of a model is $O(N^3)$. The time required to infer the fields of a model (lines 8-10) is $O(N)$. The connections of a model are handled in lines 11-31. Like the cardinality computation step of model subitems in line 6, the cardinality computation of connections is composed of conditional statements, which takes linear time. The *check_for_fco* action in lines 17 and 25 checks whether an FCO generalization exists that can be used as the source or destination for the connection under consideration instead of creating a new FCO generalization. This check is done using a nested for loop and runs in $O(N^2)$ time. The total computation time for inferring connections of a model is bounded by $O(N * (N + 2N^2))$. Lines 32-36 infer atoms and take $O(N^2)$ time. Thus the overall upper bound of the algorithm's complexity is $O(N * (N^3 + N + (N * (N + 2N^2)) + O(N^2) \approx O(N^4)$.

5. EXPERIMENTAL STUDIES OF METAMODEL INFERENCE

We conducted experimental studies to test MARS on various example domains and analyzed its performance under a variety of metamodel recovery situations that might be encountered in practice. Specifically, we wanted to test the following hypothesis: Is it possible to infer a metamodel from one model only? Is it possible to infer a metamodel from models that do not exhibit all the properties of the metamodel? To test these hypotheses, we designed experiments to infer a metamodel from multiple instances (or a single instance), which contain complete information about the metamodel, and a metamodel from instances which contain incomplete information. We applied the system to the following diverse domains which are a mix of small examples that have been created as pedagogical case studies used to teach domain-specific modeling: Finite State Machine (FSM – case study of this paper), Network Diagram Modeling Language[1], AudioVideo System Modeling Language[2], and the Petri Net Modeling Language. The model instances were created by hand to explore the metamodel recovery situations mentioned above.

5.1 Inferring the Network Diagram Metamodel

The Network metamodel is larger and more elaborate than the FSM model. Fig. 14 shows the metamodel for a simple 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. An instance of this metamodel is shown in Fig. 15, which illustrates different company configurations that are connected on the network.

We used domain models corresponding to the following configurations to infer the metamodel for this domain:

1) A `NetDiagram` containing multiple `Host` and `WSGroup` elements.
2) A `NetDiagram` containing multiple `Network` and `Perimeter` elements.
3) A `NetDiagram` containing multiple instances of the following configuration: connections from `Host` to `Network` and `Perimeter`.
4) A `NetDiagram` containing multiple instances of the following configuration: connections from `WSGroup` to `Network` and `Perimeter`.
5) A `NetDiagram` containing multiple instances of the following configuration: connections from `Port` to `Network` and `Perimeter`.

[1] The Network Diagram Modeling Language is the case study used within the tutorial that is installed by the GME.

[2] The Audio/Video metamodel was created by Jonathan Sprinkle, Greg Nordstrom, and James Davis as a pedagogical tool for the Model-Integrated Computing graduate course at Vanderbilt University.

6) A `NetDiagram` containing multiple instances of the following configuration: connections from `Perimeter` to `Network` and `Perimeter`.

7) A `NetDiagram` containing multiple `Routers`, with some of those routers containing multiple `Ports`.

8) A `NetDiagram` containing other `NetDiagrams`.

Fig. 16 shows the inferred metamodel for the network domain.

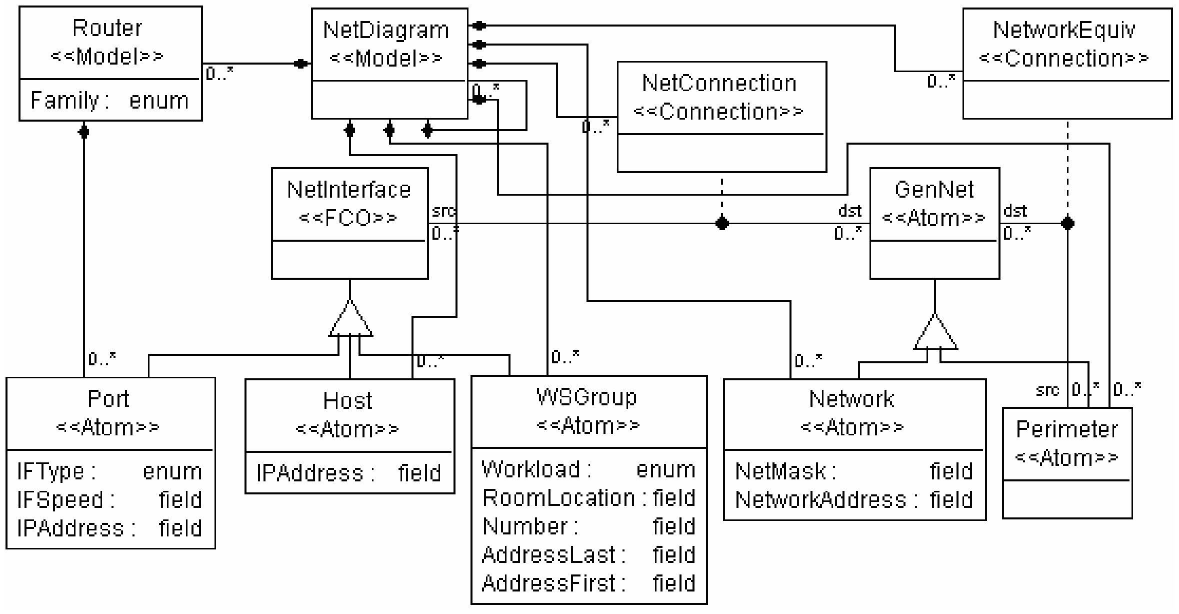


Fig. 14. Original Metamodel for Creating Network Diagrams

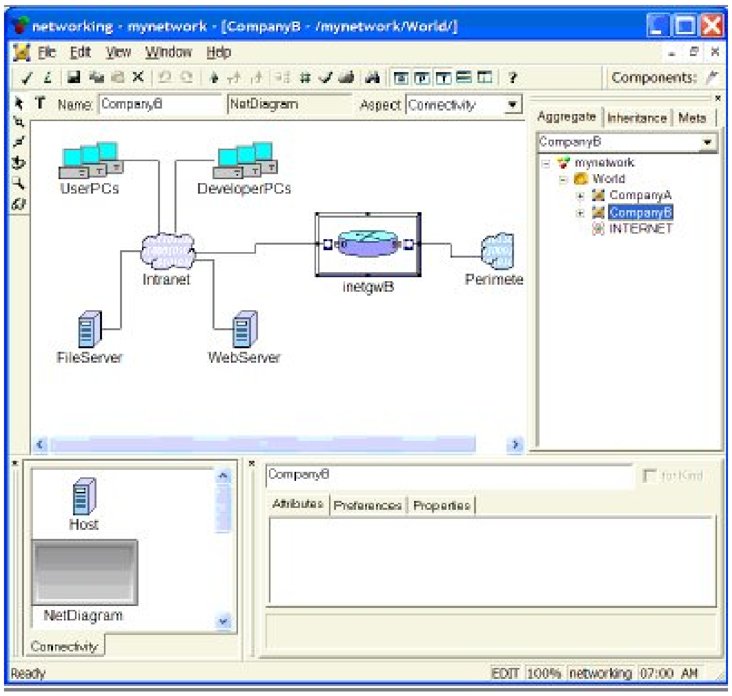


Fig. 15. An Instance of a Network

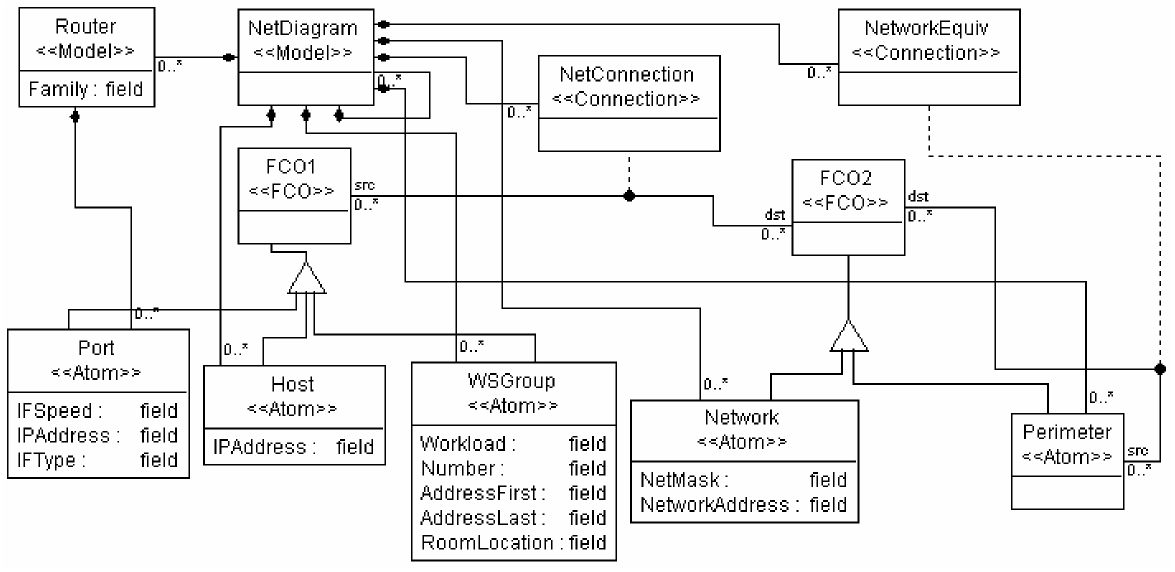


Fig. 16. Inferred Metamodel for the Network Domain

Comparing the original metamodel for the Network domain in Fig. 14 to the inferred metamodel in Fig. 16, we observe that the `NetInterface` and `GenNet` generalizations are inferred as `FCO1` and `FCO2` generalizations, respectively. The reason for this is that the generalization hierarchy is not explicitly defined in the domain models. Also, the fields `Family`, `IFType`, and `Workload`, originally of type 'enum,' are inferred as the more general type 'field.' This is because MARS currently infers all field attributes as the more general type 'field.'

5.2 Inferring the Petri Net Metamodel

The inferred metamodels from the FSM example were determined from artificially created domain models. In practice, a metamodel is inferred using the domain models available at hand, which can vary in number as well as in their information content. We demonstrate this scenario using a Petri Net [32] modeling language, which consists of the elements `Place` and `Transition`, as well as the connections between them. A `Place` can also hold a certain number of tokens. Fig. 17 shows the original metamodel for this domain.

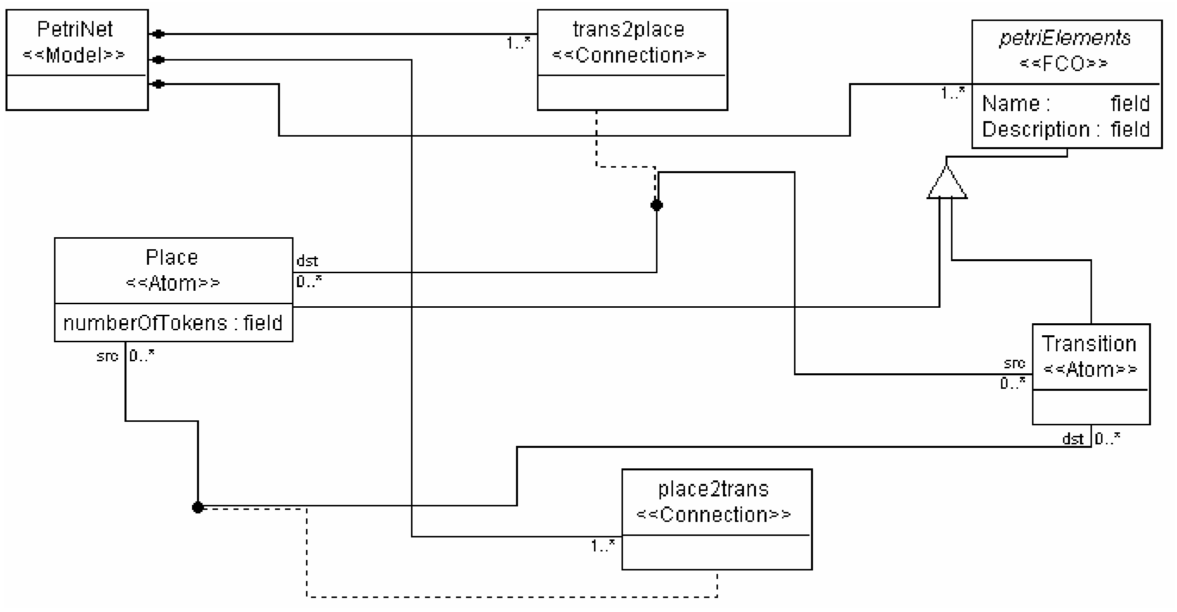


Fig. 17. Original Metamodel for the Petri Net Domain

An instance of this metamodel is shown in Fig. 18, which depicts a model of the dining philosophers problem. The visualization icons for various elements of the Petri Net metamodel are the standard icons like a circle for a `place` and a horizontal bar for a `transition`. GME also allows domain models to be reconfigured to display user-specified icons; the domain model in Fig. 18 has been configured to use different icons to better illustrate the dining philosophers problem. The Petri Net metamodel can be inferred using this sole domain model, which makes use of all the elements and connections of the original metamodel and allows the complete inference of an accurate metamodel. Fig. 19 shows the inferred metamodel for the Petri Net domain. The only difference is that the `petriElements` FCO generalization hierarchy in the original metamodel is missing from the inferred metamodel. The reason for this is that the `petriElements` generalization in the original metamodel was designed using the domain knowledge that `Place` and `Transition` are "Petri Net Elements" of various kinds. The inference process does not have this kind of built-in domain knowledge; thus, it has to perform the inference with the amount of information available in the domain models under consideration. A consequence of this is that the attributes of the `petriElements` FCO (`Name` and `Description`) are inferred as attributes for `Place` and `Transition` in the inferred metamodel.

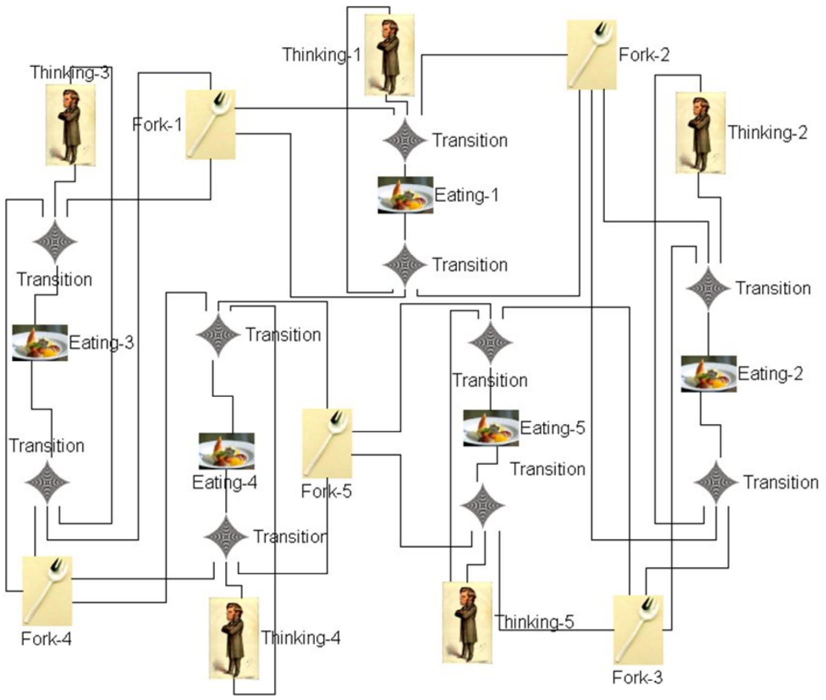


Fig. 18. Dining Philosophers: An Instance of a Petri Net

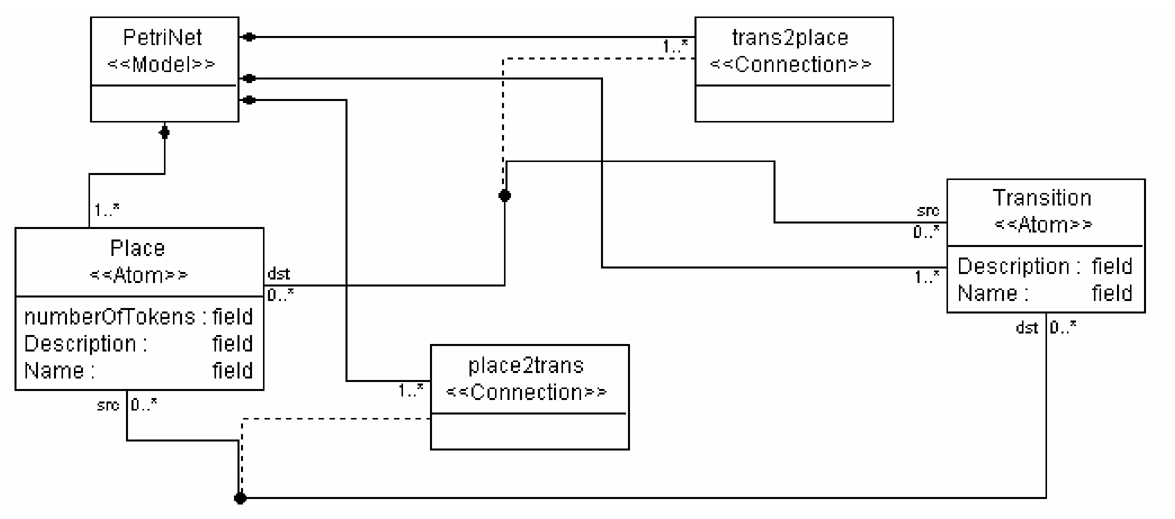


Fig. 19. Inferred Metamodel for the Petri Net Domain

5.3 Inferring the AudioVideo System Metamodel

The AudioVideo System is a domain example that illustrates the scenario of inferring a partial metamodel when the available examples do not contain all possible element and connection possibilities. The AudioVideo System metamodel describes a language for configuring a home entertainment system consisting of domain elements (e.g., VCR, DVD, LD, MiniDisc, and other assorted electronics) and the connections between them. Fig. 20 and fig. 21 show the original metamodel and an instance of the metamodel, respectively. Note that in the domain model, SONYSTR-DB1070 is an instance of an `AudioProcessor` model and houses further metamodel component configurations. Fig. 22 shows the inferred metamodel after using three available instances.

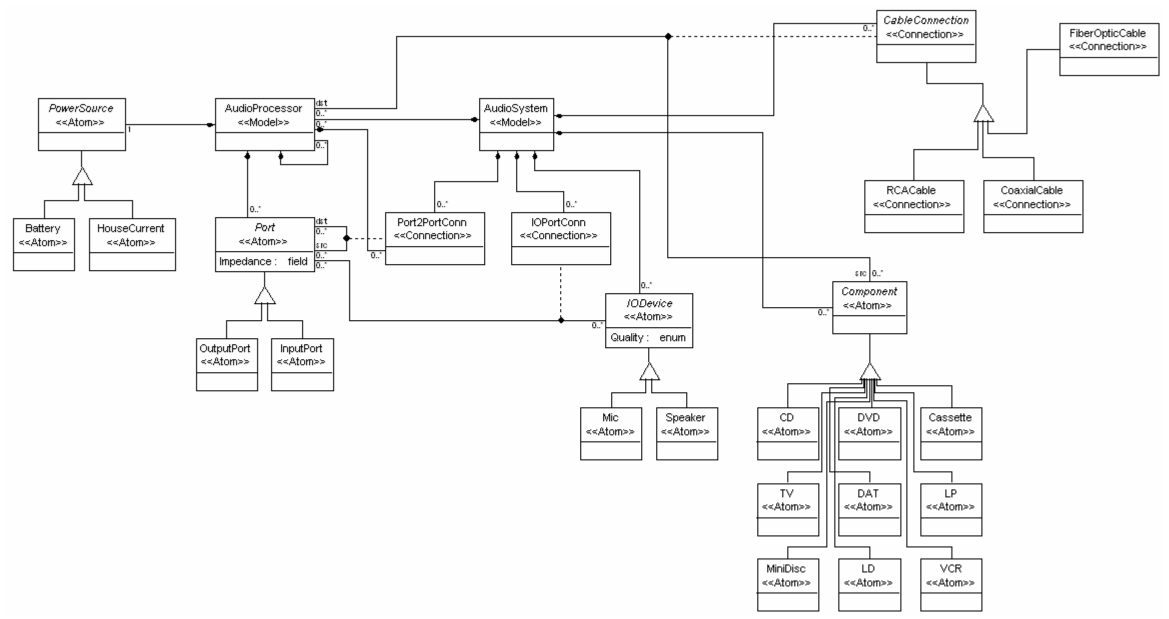


Fig. 20. Original Metamodel for the AudioVideo System Domain

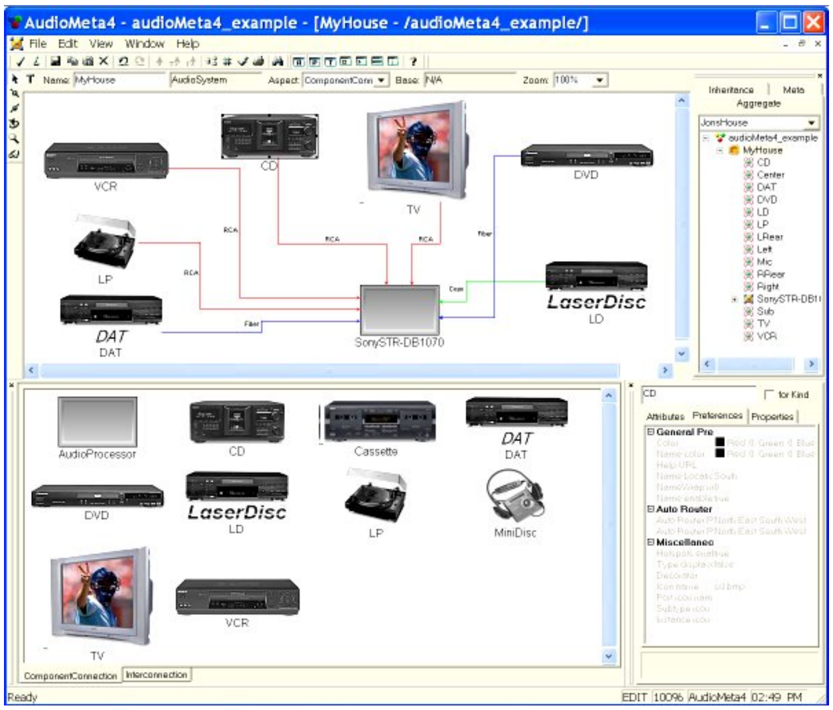


Fig. 21. An Instance of the AudioVideo System Domain

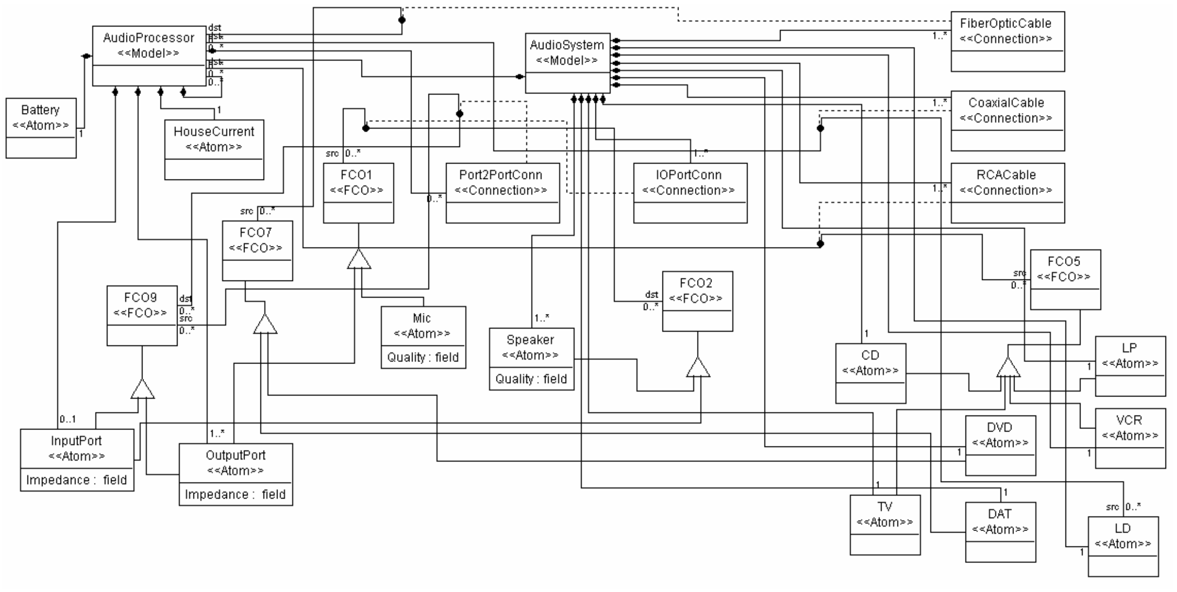


Fig. 22. Inferred Metamodel for the AudioVideo System Domain

When compared to the original metamodel, the following are the differences observed in the inferred AudioVideo System metamodel:

- Some of the atom elements (e.g., `Cassette` and `MiniDisc`) from the original metamodel are missing; this is because these elements were not present in the example domain models.
- Specializations of the various generalizations (e.g., `CableConnection`, `PowerSource`, and `IODevice`) in the original metamodels, instead of the generalizations themselves, are used in the inferred metamodel. For example, `Battery` is a direct part of `AudioProcessor`. The reason for this is the lack of domain knowledge available to the inference process. We propose some initial ideas on how to solve this problem. For example, elements with the same attributes (e.g., `InputPort` and `OutputPort`) can be considered as potential candidates for generalization, although this might not be valid for all cases. In the case of a `connection` generalization, if the elements have the same source and destination connections, they can be potential generalization candidates. For example, in the inferred metamodel the connections `FiberOpticCable,CoaxialCable` and `RCACable` all have `AudioProcessor` as the destination and `Component` specialization as the source.
- In addition to the missing elements, some connections between elements are absent. For example, in the original metamodel, the connection `RCACable` is a connection between the `Component` elements (e.g., DVD, CD, LP) as the source and the `AudioProcessor` model as a destination. In the inferred metamodel, the source for the `RCACable` connection is the element `FCO5`, which is actually the inferred metamodel equivalent to the original metamodel's `Component` generalization hierarchy (albeit with a few missing elements), and the destination is the `AudioProcessor` model. In the original metamodel, the `IOPortConn` connection defines a connection between the `Port` and `IODevice` generalizations, either of which can be source or destination. In the inferred metamodel, this is translated to `FCO1` as source (e.g., `MIC` and `OutputPort`) and `FCO2` as destination (e.g., `InputPort` and `Speaker`) because these are the possible set of connections between elements exhibited in the domain models. Similarly, `Port2Conn` in the original metamodel is part of both `AudioProcessor` and `AudioSystem` in the original metamodel, but only `AudioProcessor` in the inferred metamodel.

- There are also differences in the cardinalities. All the specializations of `Component` (e.g., `LD`, `CD`, `DVD`) have a cardinality of 1 in the inferred metamodel. This is because only one instance of these components is used in the example domain models.
- The attribute `Quality` is of type 'field' in the original metamodel, but 'enum' in the inferred metamodel.
- `Mic` and `Speaker` have `Quality` attributes in the inferred metamodel, but this attribute belongs to `IODevice` (i.e., their generalization) in the original metamodel.

5.4 Evaluation and Results

We have also applied MARS on the large multi-tiered Embedded Systems Modeling Language (ESML), which represents a domain-specific graphical modeling language developed for modeling Real-Time Mission Computing Embedded Avionics applications [33]. The ESML contains domain models representing mission computing avionics scenarios. The ESML is a large domain, with models arranged in multiple folders. MARS was able to infer an ESML metamodel which had over 80 elements. However, this inferred metamodel was incomplete because it was single-tiered and did not contain information on metamodels in other folders. MARS is currently limited to inferring single folder metamodel domains. We plan to extend this capability to infer multi-tiered metamodels.

Table 7 summarizes the results of the metamodel inference experiments. The complexity of a metamodel is measured by counting the total number of elements, aggregations, generalizations and connections in the metamodel. All the metamodels (including the large ESML metamodel) were inferred in less than 3 seconds on a modern notebook (Pentium 1.6GHz, 1GB RAM) indicating that MARS is capable of scaling well to large single-tiered metamodels.

The MARS project website [34] contains the results of experimental runs of these domains, as well as all the artifacts of the inference process (e.g., XSLT rules, sample metamodels, domain models and grammars) for all of the domains discussed in this paper.

Table 7
Summary of the Inference Experiments

Domain	Instances Used	N: number of elements in instance set	Original Metamodel Complexity	Inferred Metamodel Complexity	Observations
Finite State Machine	4	22	7 elements, 2 generalizations, 4 aggregations, 1 connection. Sum = 14	7 elements, 2 generalizations, 4 aggregations, 1 connection. Sum = 14	Some generalization FCO objects are renamed to FCO1 and FCO2.
Network	8	205	11 elements, 2 generalizations, 9 aggregations, 2 connections. Sum = 24	11 elements, 2 generalizations, 9 aggregations, 2 connections. Sum = 24	Some generalization FCO objects are renamed to FCO1 and FCO2.
Petri Net	1	175	6 elements, 1 generalization, 3 aggregations, 2 connections. Sum = 12	5 elements, 0 generalizations, 4 aggregates, 2 connections. Sum = 11	`The petriElements` generalization in the original metamodel is not inferred since it was designed by using the domain knowledge that `Place` and `Transition`

					are "Petrinet Elements" of various kinds
Audio Video System	3	310	27 elements, 5 generalizations, 10 aggregations, 3 connections. Sum = 45	25 elements, 5 generalizations, 19 aggregations, 5 connections. Sum = 54	Some model elements, generalizations and connection configurations are not inferred.

6. RELATED WORK

In this section, we compare MARS to some of the work in the area of programming language grammar recovery, component-based software reverse engineering and domain model evolution. We also compare our work to two systems that take a grammatical inference approach to Document Type Definition (DTD) [35] and XML schema [36] extraction.

6.1 Grammar Stealing, Software Reverse Engineering and Model Evolution

As mentioned in Section 1, the work of Lämmel et. al. [1] focused on recovering a programming language grammar for renovation tool construction. More specifically, they discussed the concept of *grammar stealing:* a technique that uses either compiler sources or language reference manuals to obtain a grammar, from which renovation tools are constructed. In the absence of reference manuals or compiler sources (a more problematic scenario), grammar stealing will not be very successful. However, recent advances [23] have allowed the application of grammar inference techniques to the renovation tool construction problem. Our technique can recover a grammar (albeit for small DSLs like MRL) from source code. The GSEE software exploration environment is used in [10] to reverse engineer component-based software systems. Unlike MARS, GSEE operates at the level of source code, but MARS infers models at a domain-specific modeling level. To the best of our knowledge, MARS represents the first effort that applies grammar inference techniques to the model recovery problem.

Sprinkle and Karsai [3] presented a schema evolution solution for DSM. In their approach, a visual language to allow mappings between metamodels is introduced and graph-rewriting techniques are exploited to map domain models from one metamodel schema to another. In other words, as the metamodel evolves, the domain models can be transformed accordingly to conform to the new metamodel. However, the mapping between different metamodels requires a distinct algorithm for each new schema evolution. The user has to manually observe the differences between the two metamodels, and then using the defined visual language, create an algorithm that suitably maps one metamodel to the other. By comparison, our approach uses the same algorithm and steps regardless of the specific metamodel. Also, the work in [3] is only applicable when the user is migrating domain models in an environment where a metamodel versioning system is maintained.

Graff, Weber and van Deursen propose a generic model transformation based approach for migration of supervisory machine control architectures [6]. Similar to Sprinkle and Karsai's approach [3], this model transformation based migration technique specifies mappings from source metamodels to target metamodels. In comparison, MARS is a metamodel *recovery* technique not a metamodel *transformation* technique. MARS offers a solution to the more extreme case of recovering metamodels and is also applicable in model evolution scenarios.

CacOphoNy [8] integrates software architecture and MDE to reverse engineer software architectures. It shares many similarities with the GSEE environment [10] (a research tool from the same group) and can be viewed as executing GSEE at a higher layer of abstraction. CacOphoNy has six major steps, one of which deals with metamodel recovery. The metamodel recovery capabilities of CacOphoNy appear to be manual. Although it is suggested by the authors that attempts be made by software maintenance engineers to go beyond the directly accessible information in the instances and extract interesting information, no concrete inference algorithm or heuristics to do so are discussed. The recovery relies on the software maintenance engineers' analysis of the facts contained in the source repository. In contract, MARS is a semi-automatic approach for recovering metamodels and uses an iterative algorithm with heuristics to generalize beyond the information contained in domain models.

6.2 Grammar Inference Applied to XML Schema Extraction

Grammar inference has also been applied to DTD 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 have been proposed to replace DTD. The XTRACT [37] 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 Minimum Description Length (MDL) [38] principle to choose the best DTD from a group of candidate DTDs. In [39], an Extended Context-Free Grammar (ECFG) based system is proposed to allow extraction of more complex XML schemas. Our induction system parallels these works in the sense that it is XML based, makes use of grammar inference techniques, and extracts the metamodel (analogous to DTD or other forms of XML schemas) of a specific domain model (analogous to the XML document). As intermediate representations of the XML document, the XTRACT system uses regular expressions and the ECFG-based system represents XML documents as examples of an unknown ECFG (and then tries to infer the ECFG). In comparison, our approach in MARS represents the XML document as a DSL (with a corresponding parser written as LISA specifications), resulting in a higher level of abstraction. Regarding content generalization comparisons (i.e., the process of being able to infer a concise hypothesis regarding a set of elements), the XTRACT system chooses the DTDs with the MDL score. The ECFG-system merges 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 in that containment information regarding models and atoms in a domain model XML file is used to compute the cardinality of the constituent elements of the models and the connections.

7. LIMITATIONS AND FUTURE WORK

There are several limitations to the current investigation. Each limitation can be eased by considering additional input from a model engineer. The complete metamodel inference process can be described as a semi-automatic technique that derives much of the required metamodel from the mined domain 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:

- *Data Type Inference*: For each field (attribute) of the model elements, it is not always possible to infer the attribute type from the representative XML of the model instances. For example, a string value associated with an attribute in a domain model could correspond to a string or an enumeration value. In addition to the domain models there are other artifacts that can be mined in the modeling repository. For example, model compilers can traverse the internal representation of a model and generate source code. They may also contain type information that cannot be inferred from the domain 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. To address this problem, we are currently investigating the use of a program transformation system to parse the code of the model compiler and recover the type information of metamodel entities. More specifically, the program transformation system uses a powerful term rewriting engine to allow sophisticated program analysis and transformation tasks to be performed on the code of the model compiler to extract the type information from the model compiler.

- *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 domain models, the inferred metamodel contains generic icons. To complete the visualization of the inferred metamodel, a domain expert needs to be consulted to associate the inferred 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 a domain model is created. Any violation of a constraint raises an error to inform the model engineer that some aspect of the model is incorrect with respect to the metamodel. MARS is currently unable to infer OCL constraints from domain models. The constraint definitions do not appear in the domain 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 domain 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 domain 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.
- *Inferring the Modularization of Large Metamodels:* Metamodels for large domains such as ESML are multi-tiered and are arranged in multiple folders. Currently, MARS is limited to inferring metamodels which are contained in single folders. When applied to domain models of multi-tiered domains like ESML, MARS only infers the elements and relationships contained in the primary folder. Our future work involves extending MARS to infer multi-tiered metamodels. This would possibly involve a search for all elements and their relationships which might be arranged according to folders in the domain model XML files, and may require modifying the MRL to handle scoping rules.

Although the application of MARS has focused specifically on recovering metamodels for the GME, the same process can be applied to other metamodeling tools, such as MetaCase's metaEdit+ (http://www.metacase.com), Microsoft's DSL tools (http://msdn.microsoft.com/vstudio/dsltools/), the Eclipse Modeling Project (http://www.eclipse.org/modeling/), 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. We are also investigating the use of our approach to situations where intermediate metamodels leading to the evolved metamodel are available. Using the information contained in old metamodels, more accurate metamodels can be recovered (possibly with the need of fewer domain models) when compared to the situation when no metamodels are available.

8. CONCLUSION

In most metamodeling environments, the domain models cannot be loaded properly into the modeling tool without a corresponding metamodel. The goal of the MARS project is to infer a metamodel from a collection of domain models. The motivating problem was to address the issue of metamodel drift, which occurs when domain models in a repository are separated from their defining metamodel. A growing number of research and industrial projects exhibiting the metamodel drift problem were observed which served as further motivation for this work. Although previous works have addressed the problem of metamodel evolution using model transformation techniques by specifying mappings between source and target metamodels, they do not address the more extreme situation where a metamodel needs to be recovered from models. Some research efforts like CacOphoNy do have a metamodel recovery component, but they lack precise inference heuristics and algorithms and rely on the user's analysis of information contained in the source repository.

The key contribution of the paper is a discussion of MARS, which is a semi-automatic inference system for recovering a metamodel that correctly defines the mined domain models through application of grammar inference techniques. MARS has a three-step process that uses a host of technologies such as XSLT, LISA and an inference algorithm to recover metamodels. MARS also leverages the fact that 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. MRL bridges the gap between the input expected by grammar inference engines and XML representations of the domain models. It is comprehensive enough to express all of the GME concepts that we encountered, yet concise in its design to assist as an intermediate representation for the inference process. The paper also discusses the induction techniques used to infer metamodels and gives a complexity analysis of the algorithm that runs in polynomial time. MARS has been applied to several case studies and scenarios that reflect metamodel recovery situations encountered in practice. The results successfully demonstrate the feasibility and scalability of the approach. MARS can, within a few seconds, infer single-tiered metamodels composed of a large number of elements but is currently not able to infer multi-tiered metamodels. The future work involves extending MARS to infer metamodel element types and multi-tiered metamodels. All of the extended listings (e.g., XSLT rules, sample metamodels, domain models and grammars) are available at the MARS website [34].

9. REFERENCES

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

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

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

[4] S. Johann, A. Egyed, Instant and incremental transformation of models, Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004), IEEE Computer Society Press, Los Alamitos CA, 2004, pp. 362-365.

[5] Z. Diskin, J. Dingel, A metamodel independent framework for model transformation : Towards generic model management patterns in reverse engineering, Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies for Reverse Engineering (ATEM 2006), Genoa, Italy, 2006.

[6] B. Graaf, S. Weber, A. van Deursen, Model-driven migration of supervisory machine control architectures, Journal of Systems and Software, to appear, 2007.

[7] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, C. Riva, Symphony: view-driven software architecture reconstruction, Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04), IEEE Computer Society, 2004, pp. 122-134.

[8] J-M. Favre, CacOphoNy: Metamodel driven architecture reconstruction, Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004), IEEE Computer Society, Delft, The Netherlands, 2004, pp. 204-213.

[9] J. Gray, J. Zhang, Y. Lin, H. Wu, S. Roychoudhury, R. Sudarsan, A. Gokhale, S. Neema , F. Shi, T. Bapty, Model-driven program transformation of a large avionics framework, Proceedings of Generative Programming and Component Engineering (GPCE 2004), Springer: Heidelberg, Germany, 2004, pp. 361-378.

[10] J-M. Favre, F. Duclos, J. Estublier, R. Sanlaville, J-J. Aufrette, Reverse engineering a large component-based software product, Proceedings of the Fifth Conference on Software Maintenance and Reengineering (CSMR 2001), IEEE Computer Society, Lisbon, Portugal, 2001, pp. 95-104.

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

[12] GME Users Mailing List, http://list.isis.vanderbilt.edu/pipermail/gme-users/2005-May/000776.html [14 February 2006].

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

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

[15] K. Balasubramaniam, A. Gokhale, G. Karsai, J. Sztipanovits, S. Neema, Developing applications using model-driven design environments, IEEE Computer 39(2) (2006) 33-40.

[16] G. Karsai, M. Maroti, A. Lédeczi, J. Gray, J. Sztipanovits, Composition and cloning in modeling and metamodeling, IEEE Transactions on Control System Technology 12(2) (2004) 263-278.

[17] J. Warmer, A. Kleppe, The Object Constraint Language, Addison-Wesley, Reading MA, 2003.
[18] M. Pazzani, D. Kibler, The utility of knowledge in inductive learning, Machine Learning , 9(1) (1992) 57-94.
[19] E. M. Gold, Language identification in the limit, Information and Control 10(5) (1967) 447-474.
[20] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. Shapire, L. Sellie, Efficient learning of typical finite automata from random walks, Proceedings of the Twenty-Fifth Annual Symposium on Theory of Computing, ACM Press, New York, NY, 1993, pp. 315-324.
[21] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, LISA: An interactive environment for programming language development, Proceedings of the 11th International Conference on Compiler Construction. Springer: Heidelberg, Germany, 2002, pp. 1-4.
[22] M. Mernik, V. Žumer, Incremental programming language development, Computer Languages, Systems and Structures 31(1) (2005), 1-16.
[23] M. Črepinšek, M. Mernik, F. Javed, B. R. Bryant, A. Sprague, Extracting grammar from programs: An evolutionary approach, ACM SIGPLAN Notices 40(4) (2005) 39-46.
[24] M. Mernik, M. Črepinšek, T. Kosar, D. Rebernak, V. Žumer, Grammar-based systems: Definition and examples, Informatica 28(3) (2004) 245-255.
[25] M. Mernik, J. Heering, T. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys, 37(4) (2005) 316-344.
[26] M. Wimmer, G. Kramler, Bridging grammarware and modelware, Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005), LNCS 3844, 2005, pp. 159-168.
[27] M. Alaanen, I. Porres, A relation between context-free grammars and meta object facility metamodels, Technical Report 606, TUCS, March 2004.
[28] J. Clark, XSL Transformations (XSLT) (Version 1). W3C Technical Report, November 1999, http://www.w3.org/TR/1999/REC-xslt-19991116 [14 February 2006].
[29] J. Clark, S. DeRose, XML path language (XPath) (Version 1.0). W3C Technical Report, November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116 [14 February 2006].
[30] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical Report, CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 1990.
[31] S. Peyton-Jones, Implementation of Functional Programming Languages, Prentice-Hall International, London, England, 1997.
[32] J. Peterson, Petri nets, ACM Computing Surveys, 9 (3) (1977) 223-252.
[33] D. Sharp, Component-based product line development of avionics software, Proceedings of the First Software Product Lines Conference (SPLC-1), Kluwer International, Boston MA, 2000, 353-359.
[34] The MetAModel Recovery System Project: http://www.cis.uab.edu/softcom/GenParse/mars.htm
[35] 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]
[36] J. Siméon, P. Wadler, The essence of XML, Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages, ACM Press, New York NY, 2003, pp. 1-13.
[37] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, K. Shim, XTRACT: A system for extracting document type descriptors from XML documents, Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '00), ACM Press, New York NY, 2000, pp. 165-176.
[38] J. Rissanen, Hypothesis selection and testing by the MDL principle, The Computer Journal 42(4) (1999) 260-269.
[39] B. Chidlovskii, Schema extraction from XML data: A grammatical inference approach, Proceedings of the Eighth International Workshop on Knowledge Representation meets Databases (KRDB 2001), CEUR Workshop Proceedings, Rome, Italy, 2001.