

# Metamodel Recovery from Multi-Tiered Domains Using Extended MARS

Qichao Liu, Barrett R. Bryant  
Department of Computer and Information Sciences  
University of Alabama at Birmingham  
Birmingham, Alabama, USA  
qichao@cis.uab.edu, bryant@cis.uab.edu

Marjan Mernik  
Institute of Computer Science  
University of Maribor  
Maribor, Slovenia  
marjan.mernik@uni-mb.si

**Abstract**—With the rapid development of model-driven engineering (MDE), domain-specific modeling has become a widely used software development technique. In MDE, metamodels represent a schema definition of the syntax and static semantics to which an instance model conforms (i.e., a model conforms to its metamodel in a similar manner to how a program conforms to a grammar). However, in order to address new feature requests of the domain and language, the metamodel often undergoes frequent evolution that may result in the inability of users to load and view previous model instances. MARS is a metamodel recovery system to address the problems of metamodel evolution. This paper presents our extensions to MARS to infer models for multi-tiered domains. A new XSLT translator has been developed to generate a domain-specific language (DSL) called MRL (model representation language) for the XML representation of domain instances. The metamodel inference engine has been revised to translate the MRL back into a metamodel.

*Keywords*-domain-specific modeling; grammar inference; metamodel; model-driven engineering

## I. INTRODUCTION

In software engineering, model-driven engineering (MDE) represents an alternative to classical code based development. According to Schmidt [1], “model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively.”

Compared with a programming language that is dependent on a grammar, a model is defined by a metamodel. A model conforms to its metamodel in a similar way to how a computer program conforms to the grammar of the programming language in which it is written. Both grammars and metamodels represent a schema that defines the syntax of a language. It is common for language definitions to undergo evolution to address new feature requests. Traditionally, schema evolution has been associated with the problem of evolving a database schema to adapt to a change in the modeled reality. For a

programming or modeling language, schema evolution provides new language features or adaption of the metamodel to accommodate new concerns. Lämmel and Verhoef [2] [3] addressed the schema evolution problem in the area of programming languages, including the need to renovate software to cover emerging languages or dialects. This work led to an approach to recover grammars by extracting them from language references, compilers, and other grammar-related artifacts. Once a grammar for a language is recovered, a parser generator is used to create a parser for the recovered language.

Modeling language designers often need to make changes to the metamodel even if there are many existing instance models that depend on a former version of the metamodel. After a metamodel is changed, those model instances that depend on the previous version of the metamodel may not be accessible. With the prevalence of modeling tools, this problem often arises in the modeling community. In [4], Sprinkle and Karsai proposed a domain-specific visual language (DSVL) for a mapping from the existing metamodels to the evolved metamodels. They used graph-rewriting (GR) techniques to update the domain models accordingly. However, this approach works only when both the metamodels and the intermediate transformation steps exist.

The motivation of our work is to address the metamodel inference problem in the modeling community. An initial solution to the more general problem of metamodel recovery where existing metamodels might be even missing was investigated in the MARS (MetAmodel Recovery System) project using grammar inference [5]. However, the application area of MARS is limited because large models, such as those described by the ESCHER Research Institute [6], may be hard to be used in the inference process. The main contribution of this paper is to describe an extended MARS for metamodel recovery from large models.

The paper is structured as follows. Section II gives an overview of MARS and motivates the benefits of multi-tiered domains. The extended MARS is introduced in Section III: subsection A describes the new domain specific

language (DSL) used in our extension work; subsection B introduces the revised metamodel inference engine. A case study of incrementally inferring a metamodel is presented in Section IV to illustrate how the extended MARS works to infer a metamodel and what improvement has been made to MARS. Section V briefly introduces the related work. The paper concludes with a section on future work.

## II. MARS

This section briefly introduces MARS [7] and the concept of multi-tiered domains. The overview provides references and links to additional details that describe specific features and use of the system. With the limitations of MARS as motivation, this paper extends the concepts in this section to provide an extended metamodel recovery system.

### A. Overview of MARS

An overview of MARS is illustrated in Figure 1. The activities in the box contain three major steps representing the key generators and intermediate results of the metamodel inference process. The two artifacts out of the box correspond to the required input and final output of the process. XSLT translation rules, sample metamodels, domain instances and grammars are available at the MARS website [5]. MARS works with GME (Generic Modeling Environment) [8], which is a metaconfigurable modeling tool.

Most modeling tools provide a capability to persist the model instance using XML. The metamodel inference process first reads a set of instance models in XML as input and then translates them into a DSL that captures the essence of the instance models and filters the accidental complexities of the XML model representation (step 1 in Figure 1). The DSL is then loaded into the LISA [9] language description environment (step 2 in Figure 1), which is an interactive environment for programming language development [10]. As a result of the inference process, an inferred metamodel in XML file is generated using formal transformation rules (step 3 in Figure 1). The inferred metamodel can then be loaded back in GME and used to load the previous instance models into the modeling tool.

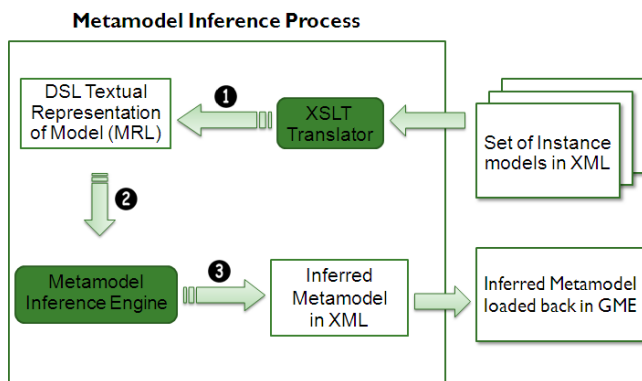


Figure 1. Overview of MARS

### B. Limitations of MARS

In order to test the effectiveness of MARS, we applied MARS to various domains (e.g., Finite State Machines and Petri Nets [11]). Although the initial version of MARS could infer a metamodel that had a close correspondence to the original metamodels, the instance models used in the inference were all from single-tiered domains that are simple and have a small number of elements.

In the second phase of evaluation, we applied MARS to handle large and complex domains. These new domains are more widely used in model-driven engineering and able to represent larger models that enable users to capture multiple viewpoints; as a result, we call such domains multi-tiered. One characteristic of the multi-tiered domain distinguishing from the single-tiered is the introduction of more modeling concepts in GME metamodels [12]. Single-tiered domains use the basic modeling concepts, such as models, atoms and connections. These modeling concepts are expressed as stereotypes of specific classes and are directly supported by the modeling environment. Atoms are classes of objects containing no other objects; models are container classes; connections are associations that are visualized with a line between objects (i.e., model, atom). Multi-tiered domain uses more concepts: folders, sets and references. Folders are containers that help organize models just like folders in operating systems help manage files. Sets can be used to specify a relationship among a group of objects. References are similar to pointers in object oriented programming (OOP) languages, and could be treated like a connection between two objects located in two different models. When an instance of a multi-tiered domain is created in GME, each viewpoint of the domain is inserted as a separate folder and placed under the “Root Folder”. ESML (Embedded Systems Modeling Language), a domain-specific graphical modeling language for modeling embedded systems [13] was chosen as our multi-tiered domain for the case study in this paper. As a multi-tiered domain, ESML has 7 different viewpoints, “Component”, “Configuration”, “Event”, “Interaction”, “Interface”, “Parameter” and “TNA”. Using the ESML metamodel, we created an instance as our running example through this paper, which captured 5 out of 7 viewpoints. Correspondingly, we inserted 5 folders into the “Root Folder” which are of type “Component”, “Configuration”, “Interaction”, “Interface” and “Parameter” separately. Each viewpoint has its own responsibility in the construction of an embedded system and viewpoints cooperate with each other to make the system work like a whole. Each viewpoint has a folder to store its own elements to fulfill its job. For example, the viewpoint “Component” could define models for different purposes or uses in its folder, (e.g., “Navigation Component”, “Sensor Component”) while the viewpoint “Interaction” specifies the event/data interactions among those defined components in its folder.

After applying MARS to this instance, we could infer a metamodel with some elements contained in the instance model missing such as sets and references. Those correctly

inferred elements were placed together in the “Root Folder”. As mentioned previously, the instance model captured 5 viewpoints and there is a folder hierarchy under the “Root Folder”. We expected to infer a metamodel having inferred elements arranged to accommodate all of these folders.

From the test result of our second phase of evaluation, MARS is limited to inferring single-tiered domains. In order to remove this limitation, we should incorporate the multi-tiered folder information contained in the instance at the beginning of the inference process and then recover this information as viewpoints in the inferred metamodel. This idea was introduced briefly in [14]. With this motivation we made extensions to MARS and details are discussed in the following section.

### III. EXTENDED MARS

In this section, we describe our extension work to MARS and illustrate how it works to avoid the limitation mentioned in Section II.A. For the extended system, the metamodel inference process remains the same as the original shown in Figure 1 but we revised the XSLT translator with a new XSLT translation and upgraded the metamodel inference engine, which are presented in subsection A and B, respectively.

#### A. XSLT Translator

The XSL (Extensible Stylesheet) Transformation [15] is a visual-to-textual representation transformation process. As illustrated in Figure 1, our XSLT Translator is responsible for reading in a set of instance models in XML and translating them into a domain-specific language (DSL). DSLs are computer languages designed for specific application domains [16]. The DSL we created is called the Model Representation Language (MRL) that accurately represents the visual GME domain models contained in XML files. Although the inference process could be applied directly to these instances in XML, the mismatch between the XML representation of an instance and the syntax expected by the grammar inference process increases the cost with greater complexity.

1) *New XSLT Translation*: The design purpose of XSLT translation was to translate the XML representation of domain instances into MRL. Figure 2 illustrates an XML fragment from a single-tiered domain instance while Figure 3 shows an XML fragment from our running example of the ESML domain instance (multi-tiered domain). For simplicity, the XML fragments shown here are pruned and only contain the core parts we are most interested in. In the XML representation of a GME model, the modeling concepts (e.g., folder, model and atom) are described as nodes. XSLT uses the XML Path Language (XPath) [17] to locate specific nodes in an XML document and retrieve values of interest. In Figure 2, we have 2 models named “Test\_model1”, “Test\_model2” under “RootFolder”. “Test\_model1” has two atoms “Test\_atom1” and “Test\_atom2”; it also has a connection named “Conn” linking “Test\_atom1” to “Test\_atom2”.

In Figure 3, we have five subfolders under the “Root Folder” which captured 5 out of 7 viewpoints of ESML domain and we will discuss “ConfigurationFolder” further in detail. There is a model named “ConfigurationModel” in “ConfigurationFolder” and it contains one sub-model named “ProcessorModel”, which has two sub-models “ProcessModel1” and “ProcessModel2”. “ProcessModel1” contains nothing while “ProcessModel2” has two atoms named “PortAtom1” and “PortAtom2”, a reference named “ComponentRef” referring to another model in the folder “Components”, and a set named “Set” with “PortAtom1”, “PortAtom2” and “ComponentRef” as the set members. “ProcessModel2 also has two connections named “Conn1” and “Conn2” linking “PortAtom1” and “PortAtom2” to “ComponentRef” separately.

```

01<folder "RootFolder">
02 <model "Test_model1">
03   <atom "Test_atom1"> ... </atom>
04   <atom "Test_atom2"> ... </atom>
05   <connection "Conn">
06     <connpoint "src"=Test_atom1>
07     <connpoint "dst"=Test_atom2>
08   </connection>
09 </model>
10 <model "Test_model2"> ... </model>
11</folder> //end "RootFolder"

```

Figure 2. XML fragment from a single-tiered domain instance

```

01<folder "RootFolder">
02 <folder "ComponentFolder"> ... </folder>
03 <folder "ConfigurationFolder">
04   <model "ConfigurationModel">
05     <model "ProcessorModel"> ... </model>
06     <model "ProcessModel1"> ... </model>
07     <model "ProcessModel2">
08       <atom "PortAtom1"> ... </atom>
09       <atom "PortAtom2"> ... </atom>
10       <reference "ComponentRef"> ... </reference>
11       <set "Set"> ... </set>
12       <connection "Conn1">
13         <connpoint "src"=PortAtom1>
14         <connpoint "dst"=ComponentRef>
15       </connection>
16       <connection "Conn2">
17         <connpoint "src"=PortAtom2>
18         <connpoint "dst"=ComponentRef>
19       </connection>
20     </model>
21   </model> //end "ProcessorModel"
22 </model> //end "ConfigurationModel"
23 </folder> //end "ConfigurationFolder"
24 <folder "InteractionFolder"> ... </folder>
25 <folder "InterfaceFolder"> ... </folder>
26 <folder "ParameterFolder"> ... </folder>
27</folder> //end "RootFolder"

```

Figure 3. XML fragment from an ESML domain instance

Compared with the single-tiered instance, ESML instance has modeling concepts like “set” and “reference” which are also established as nodes in XML. In order to retrieve nodes values we need a new XSLT translation to locate the exact position of each node; in other words, a path leading to the location of a desired node (e.g., with “RootFolder → ConfigurationFolder → ConfigurationModel” we could find the “ConfigurationModel” in “ConfigurationFolder”). Without taking this information into account, MARS mixed inferred elements and placed them together directly under “RootFolder.” The new translation is illustrated in Figure 4. For each folder under “RootFolder”, lines 05-08 will extract and store the kind of folder in the variable “folderkind”. The variable is then passed down to each model under the folder. Lines 10-23 describe a template processing models. In this template, we first store the model kind value in the variable “modelkind” and then create a basic structure of model in MRL as follows:

```

model folderkind::modelkind
{
  submodels submodel1, submodel2 .....;
  fields field1, field2 .....;
  connections
  conn1: src1 → dst1;
  conn2: src2 → dst2;
  .....
}

```

As the above structure shows, the folder information of each model is recorded in front of it. And we also record the sub-models, fields (attributes) and any connections in the model.

Lines 25-33 describe the template processing atoms. Similar to the models template, we first store the kind of atom in the variable “atomkind” and then create a basic structure for each atom as follows:

```

atom modelkind::atomkind
{
  fields field1, field2 .....;
}

```

Compared with the model structure, we placed “modelkind” instead of “folderkind” in front of each atom. Because an atom is defined in a specific model so the location of the model determines the folder that the atom comes from. It is the same case with sets and references and they have very similar structures with atoms. Lines 35-43 and 45-55 define the template of sets and references separately and their structures are described as below:

```

set modelkind::setkind
{
  fields field1, field2 .....;
  members member1, member2 .....;
}

```

```

01<xsl:template match="project">
02 <xsl:for-each select="folder"></xsl:for-each>
03</xsl:template>
04
05<xsl:template match="folder">
06 <xsl:variable name = "folderkind" select = "@kind" />
07 <xsl:for-each select="model"></xsl:for-each>
08</xsl:template>
09
10<xsl:template match="model">
11 <xsl:variable name = "modelkind" select = "@kind" />
12
13 model <xsl:value-of select="$folderkind"/>::
    <xsl:value-of select="$modelkind"/>
14 {
15 submodels process submodels;
16 fields process fields;
17 connections process connections;
18 }
19
20 <xsl:for-each select="atom"></xsl:for-each>
21 <xsl:for-each select="set"></xsl:for-each>
22 <xsl:for-each select="reference"></xsl:for-each>
23 </xsl:template>
24
25<xsl:template match="atom">
26 <xsl:variable name = "atomkind" select = "@kind" />
27
28 atom <xsl:value-of select="$modelkind"/>::
    <xsl:value-of select="$atomkind"/>
29 {
30 fields process fields;
31 }
32
33 </xsl:template>
34
35<xsl:template match="set">
36 <xsl:variable name = "setkind" select = "@kind" />
37 <xsl:variable name = "setmembers" select = "@members" />
38
39 set <xsl:value-of select="$modelkind"/>::
    <xsl:value-of select="$setkind"/>
40 {
41 fields process fields;
42 members process members;
43 }
44
45</xsl:template>
46
47<xsl:template match="reference">
48 <xsl:variable name = "refkind" select = "@kind" />
49 <xsl:variable name = "referto" select = "@referred" />
50
51 reference <xsl:value-of select="$modelkind"/>::
    <xsl:value-of select="$refkind"/>
52 {
53 fields process fields;
54 referto process referto;
55 }
56
57</xsl:template>

```

Figure 4. New XSLT translation

```

reference modelkind::referencekind
{
  fields field1, field2 .....;
  referto referred;
}

```

Each set has a series of “members” to record the group of objects that it specifies. Each reference has a “referto” to indicate the object it actually refers to.

After applying the new XSLT translation, our ESML domain instance in XML is translated into a new MRL composed of structures introduced above. Further detail will be discussed in the next section.

2) *New MRL*: As the output of the XSLT translator, MRL (Model Representation Language) describes the components of the domain instances in a specific form that can be used by the metamodel inference process. As such MRL only accentuates the most useful information contained in the domain instances (i.e., the kind or type value). Thus, an MRL program is a collection of <kind/type, identifier> implicit bindings.

After applying the new XSLT translation to our ESML domain instance in XML illustrated in Figure 3, we created a new MRL and part of the program is shown in Figure 5. Each node in the XML fragment has been translated into an equivalent MRL description. For example, folder “ConfigurationFolder” is inferred as “Configuration” kind and model “ProcessModel1” is inferred as “Process” kind. From Figure 5 we can tell that the new MRL incorporates the modeling concepts (i.e., set, reference) and stores folder information (e.g., “Configuration”) used in the instance in front of each model in the MRL.

### B. Extended Metamodel Inference Engine

1) *Mapping from MRL to Metamodel*: Section A discussed the need for MRL to establish all essential mappings from domain instances in XML to MRL. Within our metamodel inference engine, this process is reversed and a metamodel in XML file can be inferred from MRL programs.

Based on the atom structure described in the new MRL we could infer the corresponding XML representation for each atom as follows:

```

<atom id="xxx" kind="atomkind" role="atomkind">
  <name>atomkind</name>
  <attribute>
    <value>field1</value>
  </attribute>
  <attribute>
    <value>field2</value>
  </attribute>
  .....
</atom>

```

A similar process could be applied to sets and references in MRL programs with just a minor change to the above XML fragment. For each model structure in the new MRL, an XML representation like follows could be inferred:

```

... ..
model RootFolder::Configuration::Configurations
{
  submodels Processor;
  fields;
  connections;
}

model RootFolder::Configuration::Processor
{
  submodels Process, Process;
  fields;
  connections;
}

model RootFolder::Configuration::Process
{
  submodels;
  fields;
  connections;
}

model RootFolder::Configuration::Process
{
  submodels;
  fields;
  connections
  ComponentConnection : PublishPort → ComponentRef;
  ComponentConnection : SubscribePort → ComponentRef;
}

reference Process::ComponentRef
{
  fields;
  referto ComponentType;
}

atom Process::ConnectionPort
{
  fields Period, PortID;
}

set Process::EnableComponents
{
  fields GroupID, ItemID;
  members ComponentRef, ConnectionPort;
}
... ..

```

Figure 5. Part of New MRL for ESML domain instance

```

<model id="xxx" kind="modelkind" role="modelkind">
  <name>modelkind</name>
  <attribute>
    <value>field1</value>
  </attribute>
  <attribute>
    <value>field2</value>
  </attribute>
  .....
  <atom> ... </atom>
  <set> ... </set>
  <reference> ... </reference>
  <connection> ... </connection>
</model>

```

For nodes like atoms, sets and references we just need to insert their inferred structures separately into the XML representation for model.

In the new MRL we have two models of type “Process” under the model “Processor”. That means one “Processor” could contain one or many “Process”. The cardinality of the relation between model “Process” and “Processor” is inferred as “1..\*” in the metamodel. We also have two

atoms “PublishPort” and “SubscribePort” under the model “Process” and they are both connected to “ComponentRef” as sources. We introduce an element called FCO (First Class Object), which serves as an intermediate generalization that helps to better organize the inheritance relationships. In the metamodel, both “PublishPort” and “Subscribe” inherit from FCO which then connects to “ComponentRef” as the source of connection.

Table 1 illustrates a mapping between MRL and metamodel with all of the relationships and transformations specified for our ESML domain instance.

2) *Metamodel Transformation*: With the MRL program and mapping from MRL to XML, we developed a metamodel transformation algorithm depicted in Figure 6. The input to the algorithm is a set of vectors that contain the constituent elements (e.g., models, atoms) of the domain instance. The preprocessing function extracts and stores every folder (viewpoint) that was recorded in front of each model in MRL into a separate vector and then allocate a vector for each folder to store sub-items with a structure like the following:

```

RootFolder:
  Viewpoint1 {models1, atoms1, ...}
  Viewpoint2 {models2, atoms2, ...}
  ...
  
```

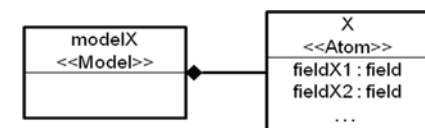
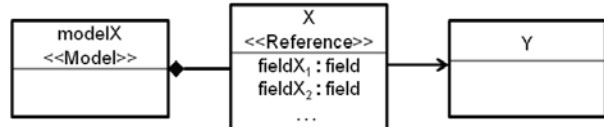
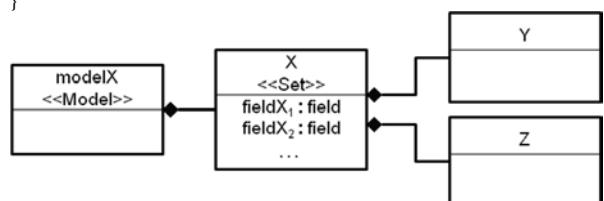
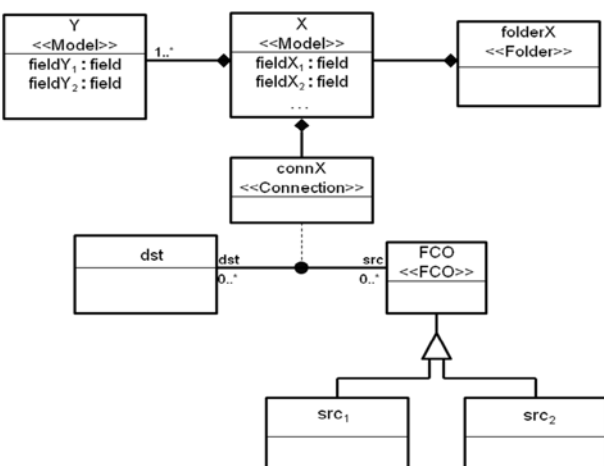
Each element is now placed under the viewpoint it came from instead of being mixed with other items directly under “RootFolder.” After applying the transformation algorithm a metamodel can now not only be drawn manually using the GME tool but be automatically recovered.

#### IV. CASE STUDY: INFERRING A METAMODEL USING EXTENDED MARS

Figure 7 illustrates the “Configuration” viewpoint of the original ESML metamodel. The metamodel in GME relies on the use of Unified Modeling Language (UML) class diagrams [18]. The UML class diagram specifies a “grammar” that represents all the possible models created in the “language”; it also specifies the static semantics of the modeling language through expressing restrictions (e.g., cardinality of objects appearing in relations). As Figure 7 indicates, the ESML metamodel has 7 viewpoints and each one could be treated as a separate metamodel.

Assume that the original metamodel is lost and we have to recover the metamodel only from domain instances. We created some instances that are similar to our running example but exhibiting different properties of the “Configuration” viewpoint. With those instances, a metamodel illustrated in Figure 8 was inferred by the extended MARS. Compared with the original metamodel, the inferred one is almost exactly the same as the original except for the fact that some item or connection was missing. For example, the reference “FailedProcessor” is missing in the inferred metamodel. The quality of the inferred metamodel depends on the level of details available in the domain instances and also the number of

Table 1. Mapping between MRL and Metamodel

|   |  |
|---|--|
| 1 | <pre> atom modelX::X {   fields fieldX<sub>1</sub>, fieldX<sub>2</sub> ... ; }           </pre>    |
| 2 | <pre> reference modelX::X {   fields fieldX<sub>1</sub>, fieldX<sub>2</sub> ... ;   referto Y; }           </pre>    |
| 3 | <pre> set modelX::X {   fields fieldX<sub>1</sub>, fieldX<sub>2</sub> ... ;   members Y, Z ... ; }           </pre>   |
| 4 | <pre> model folderX::X {   submodels Y, Y, ... ;   fields fieldX<sub>1</sub>, fieldX<sub>2</sub> ... ;   connections   connX : src<sub>1</sub> -&gt; dst;   connX : src<sub>2</sub> -&gt; dst;   ... }  model folderX::X::Y {   submodels;   fields fieldY<sub>1</sub>, fieldY<sub>2</sub> ... ;   connections; }  model folderX::X::Y {   submodels;   fields fieldY<sub>1</sub>, fieldY<sub>2</sub> ... ;   connections; }           </pre>  |

Input: Vectors containing contents of MRL  
Output: The inferred metamodel in XML file

```

fun1 contents_Preprocessing()
  for each model M in Model_Vector
    if Folder(M) not exists
      Folder_Vector.add(Folder(M));
      Folder(M).subitemVector.add(M);

    if Model_Fields_Vector is not empty
      for all fields in Model_Fields_Vector
        process model_fields

    if Model_Connections_Vector is not empty
      for all connections in Model_Connections_Vector
        process connections

        if connection has more than 1 source
          check_for_fco
          if fco exists
            process_source_with_fco
          else
            process_source_with_new_fco
        else
          process source

        if connection has more than 1 destination
          check_for_fco
          if fco exists
            process_destination_with_fco
          else
            process_destination_with_new_fco
        else
          process destination

  for each atom A in Atom_Vector
    Model(A).subitemVector.add(A);
    if Atom_Fields_Vector is not empty
      for all fields in Atom_Fields_Vector
        process atom_fields

  for each set S in Set_Vector
    Model(S).subitemVector.add(S);
    if Set_Fields_Vector is not empty
      for all fields in Set_Fields_Vector
        process set_fields

  for each reference R in Ref_Vector
    Model(R).subitemVector.add(R);
    if Ref_Fields_Vector is not empty
      for all fields in Ref_Fields_Vector
        process ref_fields

fun2 MRL_XML_translation
  for each folder F in Folder_Vector
    for each model M in F.subitems
      construct XML representation

```

Figure 6. Metamodel Transformation Algorithm

domain instances used. If the set of domain instances used in the inference did not make use of all the constituent elements of the original metamodel, then those elements cannot be recovered in the inferred metamodel. That is the reason why some items are missing. Other than that, almost all properties addressed in the instances were used properly in the metamodel inference and the cardinalities of connections appeared in the original were also correctly computed. For example, the cardinality of the relation between the model “Process” and the model “Processor” is “1..\*” in both the original and the inferred metamodel.

Figure 9 illustrates an inferred metamodel applying the original MARS based on the same instances used by the extended MARS. As mentioned previously, MARS simply placed all inferred elements (from different viewpoints) directly under the “Root Folder” without distinguishing different viewpoint of each element. And if we compare the inferred elements in this metamodel with those of the original we could find that lots of elements were not recovered.

From this case study we tested the effectiveness of our extended MARS on the multi-tiered domain (i.e., ESML domain).

## V. RELATED WORK

Selonen and Kettunen presented metamodel-based inference in [19], which is a flexible approach for inferring inter-model correspondence, a relationship established between model elements representing the same concepts. MARS is a metamodel recovery system focusing on metamodel inference using model instances instead of model correspondence inference.

As mentioned in Section I, Lämmel and Verhoef focused on recovering a programming language grammar for renovation tool construction in [2] [3]. However, without the language reference manuals or compiler sources this work will not be very successful. Our technique could recover a metamodel from domain instances. GSEE (Generic Software Exploration Environment) used in [20] can reverse engineer component-based software systems. GSEE operates on source code but MARS infers models at the domain-specific modeling level.

CacOphoNy, a generic metamodel-driven process integrating software architecture and MDE to reconstruct software architectures was presented in [21]. MDE was chosen because it had a broad scope and the potential to cover the whole spectrum of software engineering processes. Although one step of CacOphoNy deals with metamodel recovery, this capability appears to be manual. MARS incorporated model-driven engineering and represents the effort on model recovery problem. MARS is a semi-automatic approach for metamodel recovery beyond the information contained in model instances.

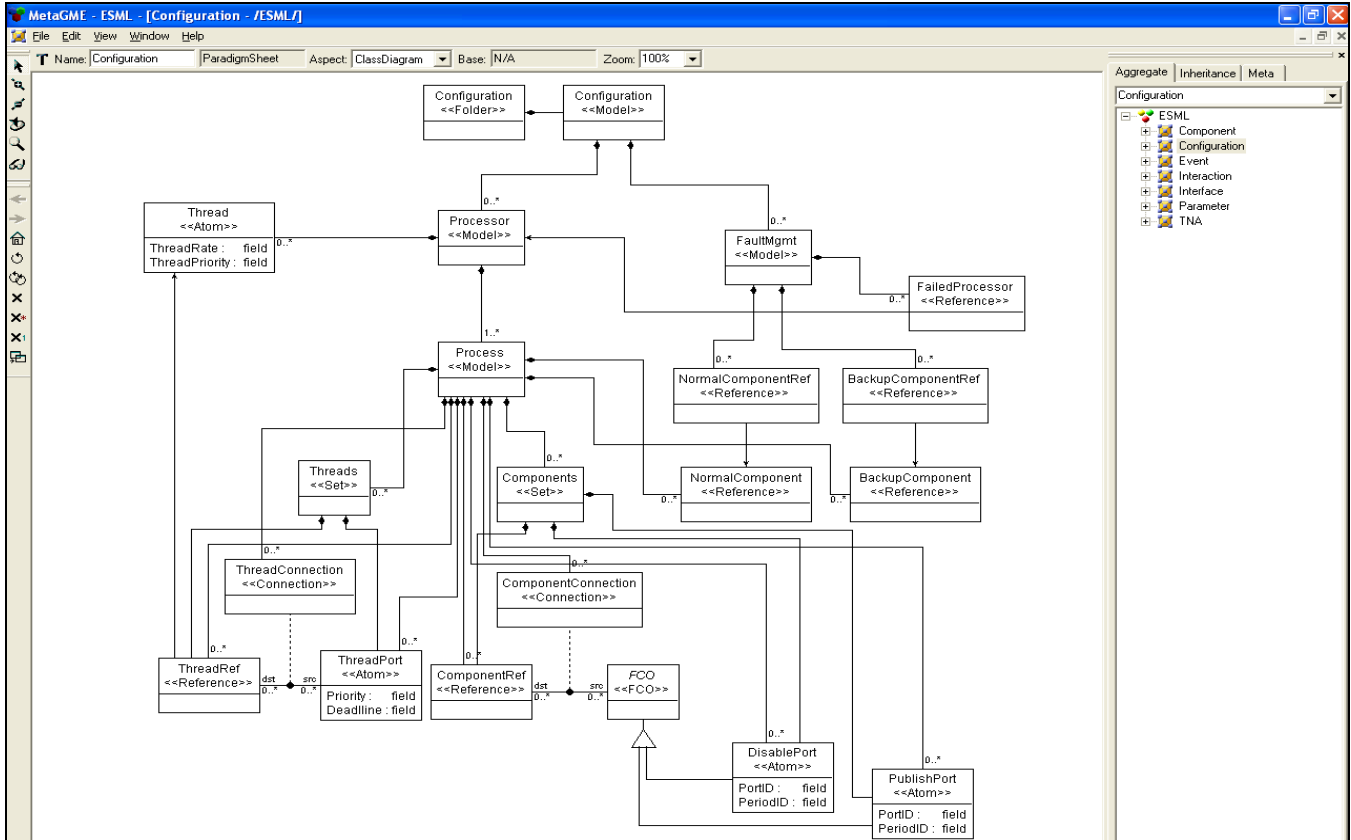


Figure 7. Original ESML Metamodel "Configuration" Viewpoint

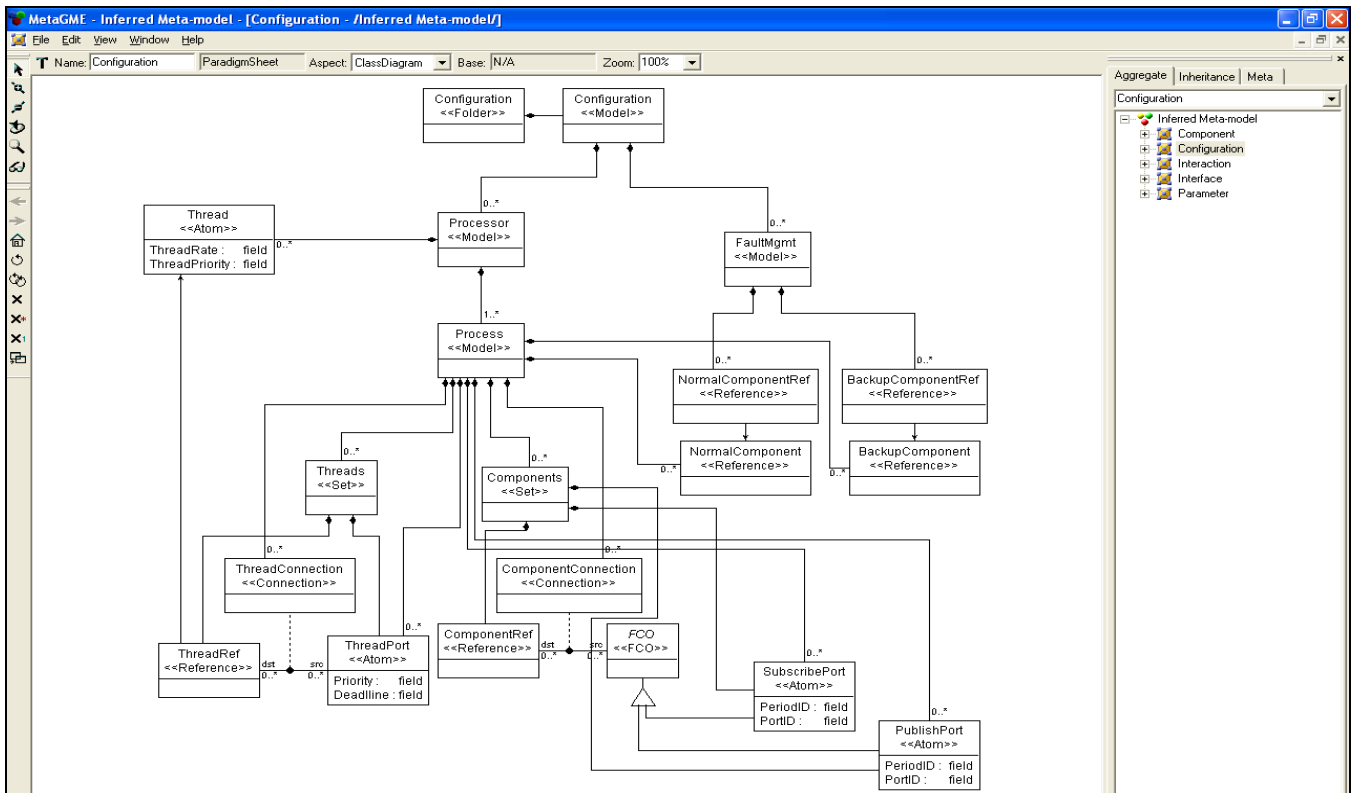


Figure 8. Inferred ESML Metamodel "Configuration" Viewpoint using Extended MARS



- [3] R. Lämmel, C. Verhoef, "Cracking the 500 language problem", *IEEE Software*, November/December 2001, 18(6), pp.78-88.
- [4] J. Sprinkle, G. Karsai, "A domain-specific visual language for domain model evolution", *Journal of Visual Languages and Computing*, 2004, 15(3-4), pp. 291-307.
- [5] The MetAmodel Recovery System Project:  
<http://www.cis.uab.edu/softcom/GrammarInference/>
- [6] The ESCHER Research Institute  
<http://repository.escherinstitute.org/Plone>
- [7] F. Javed, M. Mernik, J. Gray, B. Bryant, "MARS: A metamodel recovery system using grammar inference", *Information and Software Technology*, August 2008, 50(9-10), pp.948-968.
- [8] The Generic Modeling Environment,  
<http://www.isis.vanderbilt.edu>.
- [9] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, "LISA: An interactive environment for programming language development", 11<sup>th</sup> Int'l. Conf. Compiler Construction (CC), 2002, pp. 1-4.
- [10] M. Mernik, V. Žumer, "Incremental programming language development", *Computer Languages, Systems and Structures*, April 2005, 31(1), pp.1-16.
- [11] J. Peterson, "Petri nets", *ACM Computing Surveys*, September 1977, 9(3), pp. 223-252.
- [12] G. Karsai, M. Maroti, A. Lédeczi, J. Gray, J. Sztipanovits, "Composition and cloning in modeling and metamodeling", *IEEE Transactions on Control System Technology*, 2004, 12(2), pp. 263-278.
- [13] G. Karsai, S. Neema, D. Sharp, "Model-driven architecture for embedded software: A synopsis and an example", *Science of Computer Programming*, September 2008, 73(1), pp. 26-38.
- [14] Q. Liu, F. Javed, M. Mernik, B. R. Bryant, J. Gray, A. Sprague, D. Hrnčić, "MARS: Metamodel Recovery from Multi-tiered Models Using Grammar Inference", Third IEEE Int'l. Symp. Theoretical Aspects of Software Engineering (TASE), 2009, pp. 325-326.
- [15] J. Clark, *XSL Transformations (XSLT) (Version 1)*. W3C Technical Report, November 1999,  
<http://www.w3.org/TR/1999/REC-xslt-19991116>.
- [16] M. Mernik, T. Sloane, J. Heering, "When and how to develop domain-specific languages", *ACM Computing Surveys*, January 2005, 37(4), pp.316-344.
- [17] 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>.
- [18] G. Booch, I. Jacobson, J. Rumbaugh, "The Unified Modeling Language User Guide", Reading, MA: Addison-Wesley, 1998.
- [19] P. Selonen, M.Kettunen, "Metamodel-Based Inference of Inter-Model Correspondence", Eleventh Conference on Software Maintenance and Reengineering (CSMR), 2007, pp. 71-80.
- [20] J-M. Favre, F. Duclos, J. Estublier, R. Sanlaville, J-J. Aufrette, "Reverse engineering a large component-based software product", Fifth Conference on Software Maintenance and Reengineering (CSMR), 2001, pp. 95-104.
- [21] J-M. Favre, "CacOphoNy: Metamodel driven architecture reconstruction", 11<sup>th</sup> Working Conf. Reverse Engineering (WCRE), 2004, pp. 204-213.
- [22] The Generic Eclipse Modeling System (GEMS)  
<http://www.eclipse.org/gmt/gems/>
- [23] Domain-Specific Modeling with MetaEdit+  
<http://www.metacase.com/>
- [24] Microsoft Visual Studio  
<http://msdn.microsoft.com/vstudio/dsltools/>