Creating Visual Domain-Specific Modeling Languages from End-User Demonstration

Hyun Cho, Jeff Gray, and Eugene Syriani

Department of Computer Science
University of Alabama
Tuscaloosa, AL USA
hcho7@crimson.ua.edu, gray@cs.ua.edu, esyriani@cs.ua.edu

***Abstract*—Domain-Specific Modeling Languages (DSMLs) have received recent interest due to their conciseness and rich expressiveness for modeling a specific domain. However, DSML adoption has several challenges because development of a new DSML requires both domain knowledge and language development expertise (e.g., defining abstract/concrete syntax and specifying semantics). Abstract syntax is generally defined in the form of a metamodel, with semantics associated to the metamodel. Thus, designing a metamodel is a core DSML development activity. Furthermore, DSMLs are often developed incrementally by iterating across complex language development tasks. An iterative and incremental approach is often preferred because the approach encourages end-user involvement to assist with verifying the DSML correctness and feedback on new requirements. However, if there is no tool support, iterative and incremental DSML development can be mundane and error-prone work. To resolve issues related to DSML development, we introduce a new approach to create DSMLs from a set of domain model examples provided by an end-user. The approach focuses on (1) the identification of concrete syntax, (2) inducing abstract syntax in the form of a metamodel, and (3) inferring static semantics from a set of domain model examples. In order to generate a DSML from user-supplied examples, our approach uses graph theory and metamodel design patterns.**

Keywords; **Domain-Specific Modeling Languages, Metamodel, Metamodel Inference, Semantic Inference, Graph Theory, Metamodel Design Patterns**

I. INTRODUCTION

Domain-Specific Modeling Languages (DSMLs) raise the level of abstraction while at the same time keep the design space narrowed to a specific domain. Due to the increased abstraction, it has been observed that end-users can learn a DSML in a short amount of time because they perceive that they are working directly with domain notions [8]. Moreover, it has been shown that DSMLs help to produce quality domain models [10].

However, DSML development is not an easy task because it requires both domain knowledge and language development expertise. In addition, DSMLs often are developed incrementally by iterating over complex tasks. As shown in Figure 1, DSML development begins by eliciting the DSML requirements, where domain experts describe what capability is required and how that is represented in their domain. Based on the DSML requirements, language development experts then identify the concrete syntax, design the abstract syntax, and specify semantics. Finally, domain experts verify the DSML and give formative feedback that may lead to an iteration requiring more development. The design of the concrete and abstract syntax, and the specification of semantics are difficult even for language development experts. In addition, an iterative and incremental DSML development process can be mundane and error-prone if there is no tool support.

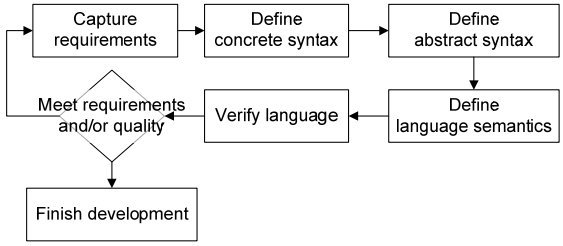


Figure 1. DSML Development Process

To lessen the burden of DSML development, we have investigated a new approach for creating DSMLs, especially visual DSMLs, based on a set of model examples that are provided by a domain expert. Graph theory is used in our approach to analyze each model instance representation independently and to infer the metamodel and static semantics based on the provided examples. Our approach allows the concept of end-user model "sketching" on white boards to be captured in a modeling tool that understands the domain abstractions and static semantics.

This paper is organized as follows. Section 2 describes the overall process for inferring a metamodel and semantics from a set of domain model examples. We introduce in Section 3 the approach that uses graph theory to handle a set of domain model examples. Section 4 describes how to induce a metamodel and static semantics from the graph representation of the end-user provided model examples. Section 5 describes related work, and Section 6 concludes with future work based on current limitations.

II. MLCBD: FRAMEWORK FOR DSML CREATION

Conventionally, a visual DSML is developed using metamodeling environments such as GME [16] and MetaCase [17]. Although visual DSML development tasks are aided by such metamodeling tools, most tasks (e.g., defining abstract/concrete syntax and specifying semantics) are generally done manually by language development experts who may not have deep understanding or experience in the domains in which a visual DSML is needed. To address these challenges of visual DSML development, we have created a framework named MLCBD, which can assist in creating a visual DSML in a semi-automated manner. MLCBD addresses the issues and challenges of DSML development described in [3]. The overall process of MLCBD is illustrated in Figure 2.

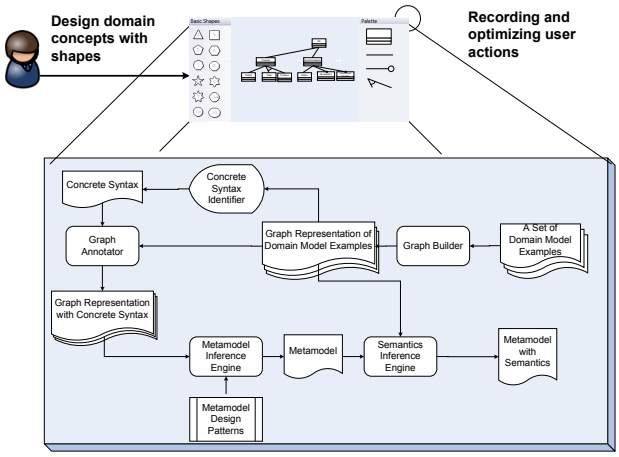


Figure 2. Process of MLCBD for Creating Visual DSML

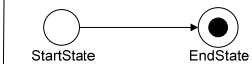
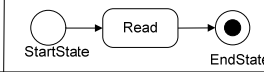
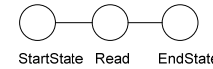
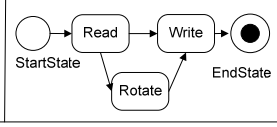
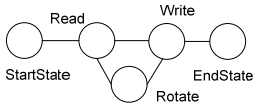
The process for creating a visual DSML begins with the demonstration of a set of model examples by end users. A Modeling Canvas, which has features similar to diagramming tools (e.g., Microsoft Visio), is provided to domain experts who then model the concepts in their domain. While domain experts demonstrate the notions of a domain, the sequences of user actions are captured and optimized to indentify candidate concrete syntax.

After a set of domain model examples is created, those examples are transformed into graph representations. Graph transformation is widely used in the domain modeling community and is also a proven approach for representing a high-level programming language formally, which can then be used to generate other types of outputs by applying production/replacement rules through graph rewriting [2][14]. Transforming platform-independent models into platform-specific models is a common example of graph transformation [11]. In our approach, graph transformations are used in two places: the Graph Builder and Graph Annotator in Figure 2.

The Graph Builder transforms a set of domain model examples into a set of graphs. The goal of the Graph Builder is to generate a representation-independent model from a set of domain model examples provided by an end-user. Because DSMLs can be developed in various languages, domain models can be described with different representations. For example, to define the syntax of a DSML and maintain model instance data, a visual DSML may use different file representations such as XML, text, and binary forms. Although DSMLs may use the same file representation, the schema representing the metamodel for the visual DSML can be structured differently for each visual DSML. Thus, the Graph Builder reads a set of domain model examples and transforms them into the corresponding internal graph representation prior to inferring the metamodel.

Table I shows examples of FSM instances that model a simple photo view application (left side), and the corresponding graph representations (right side). The first FSM instance, shown in Table I (a), represents that the application terminates without doing anything after startup. In Table I (b), the application reads an image to display. The last FSM instance in Table I (c) describes the state transition for copying an image from one media to another, or saving an image after rotation. The graph representation of each FSM model is shown in the right side of Table I. As the Graph Builder transforms a set of model examples based on simply modeling elements and their links, it generates a set of undirected graphs.

TABLE I. INSTACES OF FSM AND GRAPH REPRESENTATION

	Finite State Machine	Graph Representation
(a)		
(b)		
(c)		

After a set of domain model examples are transformed into the internal graph representations, the Concrete Syntax Identifier defines the concrete syntax of the visual DSML in a semi-automated manner. First, the Concrete Syntax Identifier searches for candidate concrete syntax from a set of graph representations of the domain model examples. Because the concrete syntax describes how modeling concepts (or abstract syntax) are rendered with visual and/or textual elements, the Concrete Syntax Identifier finds unique modeling elements, which are modeled as nodes in a graph, by traveling each graph representation. In our approach, we focus only on visual modeling elements. The identified unique modeling elements become the candidate concrete syntax. Then, the candidate concrete syntax is reviewed and annotated by the end-user who is a domain expert. Initially, the candidate concrete syntax is selected with respect to the uniqueness of modeling elements (e.g., shapes and styles) such that different names can be associated to each modeling element. Thus, the framework requires interaction with the domain expert to review the candidate concrete syntax and annotate each unique modeling element with a generalized name that can represent the notion of each modeling element precisely and clearly.

During the review and annotation of candidate concrete syntax, users may be asked to assign additional information for association links between example model elements, especially directional information. A link is used to connect two or more classifiers and provides a static semantic relationship between connected classifiers. The direction of a link adds constraints between connected classifiers such as direction of data or control flow. For example, a Dependency relationship in UML is used to represent how a change in a model element may affect the semantics of dependent modeling elements at the modeling level. The arrow of a dependency specifies the direction of a relationship between connected modeling elements.

Table II shows the identified concrete syntax from Table I and the results of the annotation. As shown in Table II (a), four symbols are identified as candidate concrete syntax from Table I, and their instance names are associated to the symbol for annotation. Domain experts review each symbol and associated attributes, in this case only instance names, and then finalize concrete syntax identification by naming each symbol with the term, which is generally accepted in the domain. The assignment of required attributes is shown in Table II (b). If a symbol is classified as a relationship type (e.g., association, aggregate, or inheritance), domain experts are asked whether the relationship is directional.

TABLE II. CONCRETE SYNTAX IDENTIFICATION

	Name
	StartState
	EndState
	Read, Write, Rotate

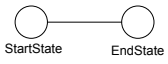
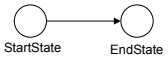
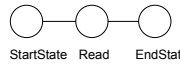
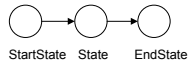
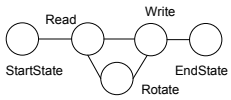
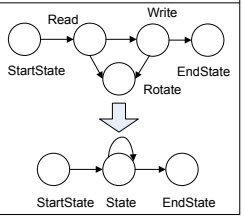
(a) Identified concrete syntax

	Name	Attribute
	StartState	Type=Classifier
	EndState	Type=Classifier
	State	Type=Classifier
		Type=Association Directional=Yes

(b) Output of Concrete Syntax Identifier

After the concrete syntax is specified, the Graph Annotator rewrites the graph representations generated by the Graph Builder with the names of concrete syntax. To generate a graph representation with concrete syntax, the Graph Annotator takes the following three steps. First, all nodes and edges are renamed with the matched concrete syntax. The name of the nodes and edges are initially assigned with instance names and the instance names are provided to the domain experts when they review and annotate the candidate concrete syntax. After the concrete syntax is defined, the arbitrary names need to be renamed with their corresponding concrete syntax name. Second, the Graph Annotator checks the link information, which is also added when the domain experts annotate the concrete syntax, and determines whether to change the graph into a directed graph. Initially, the graph representation is constructed using an undirected graph because only limited link information (e.g., name and participated classifiers) is provided when transforming the domain model examples into a graph representation by the Graph Builder. However, additional pieces of information about links (including the direction of a link) are added at the concrete syntax identification phase. Finally, the Graph Annotator completes the graph generation by optimizing a graph structure to reduce the complexity. Table III shows the result of the Graph Annotator. The first two graphs are simply transformed into the directed graph from the undirected graph as domain experts annotate links between modeling elements that are association types with direction. However, the last graph is not just transformed into the directed graph, but optimized by merging nodes, especially State nodes, which have the same name and attribute. The node State is used multiple times between node StartState and EndState to represent state Read, Write, and Rotate, and forms a cyclic loop. The Graph Annotator optimizes the graph by pruning redundant nodes and edges, and redirecting links to the remaining node.

TABLE III. OUTPUT OF GRAPH ANNOTATOR

	Graph Representation By Graph Builder	Output of Graph Annotator
(a)		
(b)		
(c)		

Based on the graph representation with the concrete syntax, the Metamodel Inference Engine infers an abstract syntax and produces its output in the form of a metamodel. Generally, metamodel inference can be considered a special case of inductive learning, which induces output by learning from examples [7][13]. To induce quality output, the inference engine requires a large set of training data, which contains positive (i.e., a set of data that belongs to target) and negative (i.e., a set of data that not belongs to target), but preparing such training data is not easy in practice [3]. To address the issue of preparing training data for inductive learning, we introduce the notion of metamodel design patterns.

Metamodel design patterns extend the notion of design patterns onto metamodel design in order to provide solutions for recurring problems when designing a metamodel. In previous work [4], we analyzed the commonality and variability of publicly available DSMLs and elicited common features (i.e., classifier and relationship) and variable features (e.g., style, containment, and nesting). Based on the commonality and variability analysis, we defined three initial metamodel design patterns, which can be commonly applied to the design of a metamodel. From the idea that a set of domain model examples can be instantiated from a metamodel design pattern, the set of metamodel design patterns can be used as training or reference data for the inference engine. Thus, the Metamodel Inference Engine is designed to find the maximum-likelihood for an inference by comparing the graph representation of domain model examples with the set of metamodel design patterns.

The final task of the MLCBD framework is inferring static semantics from a set of domain model examples. An explanation about metamodel and semantic inference is described in the next section.

III. INFERRING METAMODEL AND SEMANTICS

This section provides an overview of the process for inferring a metamodel and its associated static semantics, especially static constraints, from the graph representation.

A. Metamodel Inference

After a set of domain model examples are transformed into a set of graph representations with concrete syntax by the Graph Annotator, the Metamodel Inference Engine infers a metamodel based on the graph representation. To induce a metamodel, the Metamodel Inference Engine loads each graph representation, and then merges them in order to generate a single graph that represents the entire set of domain model examples. When combining each domain model example, the type of each edge (e.g., mandatory and optional) is specified based on the appearance of a node.

For instance, node StartState and EndState are specified as mandatory because they are used in every domain model example. However, node State is optional because it is not present in some domain model examples, such as those in Table III (a). The result of the graph combination is shown in Figure 3. The merged graph is similar to the graph in Table III (c), but a link between the StartState to EndState is added in order to cover the FSM model that has only start and end states, such as Table III (a). In addition, a dotted line is used between StartState/EndState and State to represent that a State can be used optionally in FSM models.

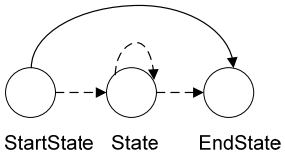


Figure 3. Combined Graph Representation

Besides combining graph representations, the Metamodel Inference Engine determines the cardinality between nodes as well as the dependency of the nodes in the graph. The identified cardinality and dependency are represented by using a matrix, as shown in Table IV. In both matrices, the first column represents the source node and the first row lists destination nodes. Table IV (a) shows the cardinality of FSM, shown in Figure 3, assuming that nodes are connected with a directed link. As links are directional, only certain nodes can be source nodes, such as StartState and State, because those links initiate a connection to End State or to State. The table is initially set to 0 for the minimum and maximum value of each cell. When the first domain model example from Table III (a) is processed, both the minimum and maximum cardinality between StartState and EndState is set to 1. When the second domain model example is introduced, the cardinality between StartState and State, and State and EndState, are updated. At this time, the minimum cardinality between StartState and State remains 0 because State does not appear in the first model instance. The table is finally completed by filling the cardinality between States when the last domain model example is processed.

Generating the dependency matrix is simpler than that of the cardinality matrix because it only checks nodes, which are involved in linking, and directional information. Table IV (b) shows the dependency matrix for the graph shown in Figure 3. The combined graph representation and cardinality/dependency matrix are used to infer the metamodel as well as the semantics of the DSML (i.e., static constraints) that will be described in the next section.

TABLE IV. CARDINALITY MATRIX AND DEPENDENCY MATRIX

	StartState	EndState	State
StartState	0	1,1	0, 1
EndState	0	0	0
State	0	1,1	0,2

(a) The Cardinality Matrix

	StartState	EndState	State
StartState	0	1	1
EndState	0	0	0
State	0	1	1

(b) The Dependency Matrix

After the Metamodel Inference Engine creates a combined graph representation and its corresponding cardinality and dependency matrix, it performs a graph and subgraph isomorphism test to infer the metamodel. For the (sub)graph isomorphism test, the combined graph representations are used as an input graph and tested over a set of graphs that are instantiated from metamodel design patterns. Metamodel design patterns can provide solutions for recurring problems when designing a metamodel. In previous work [4], we proposed three different metamodel design patterns after analyzing the commonality and variability of publicly available DSMLs. Figure 4 shows one of the metamodel design patterns, which shows the base metamodel design pattern and possible graph instances of the pattern. The base metamodel design pattern is commonly used when designing a metamodel for simple DSMLs that consist of simple classifiers and relationships. The top part of Figure 4 (b) represents two different classifiers that are linked with a directed relationship, and the middle part describes two or more classifiers that are linked using a circular and/or directed relationship.

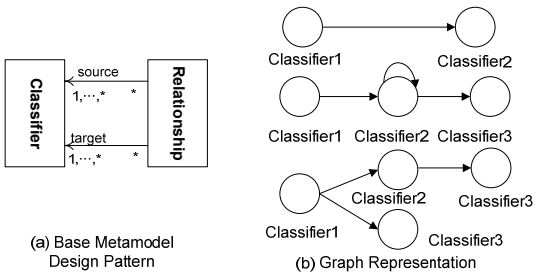


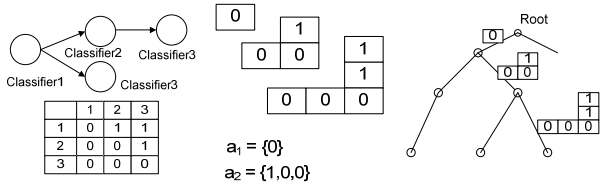
Figure 4. Base Metamodel Design Pattern

```
Let G be the combined graph representation, and
    M be the dependency matrix of G and
        represented as M = ( a1, a2, ..., am)
        where m = |V|, size of vertex in the graph G,
            and ai is the row-column element that is attached
                to node i.

Let GTD = (GTD1, GTD2, ..., GTDk) be a set of graph template
    of metamodel design patterns, and k is the number of
    metamodel design patterns

Measure size of vertex, m = |V|, of the combined graph

Instantiate_Compile_Tree(GTD, m )
For i =1 to k
   GM += Merge_Tree( Node Root, GDi)
next for

Let N = Root of GM
For i=1 to m
   Look up row-column elements that is attached to node N,
   and find an entry aN such that aN = ak
   if no matched element is found,
      the graph G is not isomorphic with GM, and
          exit with failure
   else N = NS and mark node N matched
next for

Traverse GM to find the marked nodes and then transform the
node with corresponding metamodel design pattern.

Instantiate_ Graph (GTD, m )
   Instantiate GD = (GD1, GD2, ..., GDk) with size of m
      from GTD and GDi = (GDi1, GDi2,..., GDil ) is l different
      types of graph instances of l^th metamodel design pattern
      with vertex size m
   Create the root node Root of the decision tree
      if it does not yet exist.

Merge_ Graph(Node Root, GDi)
   Generate dependency matrix of GDi,
      MDi = (MDi1, MDi2,..., MDil,), where dimension of MDi
      is m, and MDik = (aik1, aik2,..., aikm), where aikl is
      row-column elements that is attached to node l

   Let N = Root
   For l = 1 to m
      If there is successor node NS of N for which aikl = aNs,
          and N = NS
      Else Generate a new node NS for aikl and
          make NS a direct successor of N.
      end if
   next for
```

Figure 5. The Algorithm for Metamodel Inference

The algorithm for inferring a metamodel is shown in Figure 5. To infer a metamodel, a combined graph representation is tested over a set of graphs, which is instantiated from metamodel design patterns in order to check (sub) graph isomorphism. Thus, the algorithm needs to solve the one-to-many (sub)graph isomorphism problem: one combined graph representation and many graphs instantiated from the metamodel design patterns. To solve the issue, the algorithm extended and modified the algorithm proposed by Messmer and Bunke [12], whose algorithm tests one-to-many (sub)graph isomorphism in quadratic time. In order to test (sub)graph isomorphism, the algorithm transforms a set of reference graphs into a decision tree and then tests the input graph over the decision tree.

The algorithm shown in Figure 5 consists of three parts; Instantiate_Graph(), Merge_Graph(), and main. Instantiate_Graph() generates a set of graphs from metamodel design patterns with size of m, where m is the size of the vertex of the combined graph representation. As shown in Figure 4, each metamodel design pattern is used as a template to instantiate a set of graphs. Merge_Tree() merges graphs instantiated from Instantiate_Graph() in order to make a decision tree. When graphs instantiated from metamodel design patterns are passed into Merge_Graph(), a dependency matrix is constructed and its row-column representation for every graph is passed into Merge_Graph(). Figure 6 (a) and (b) show the dependency matrix and its corresponding row-column representation of the graph at the top of Figure 6 (a). As shown in Figure 6 (b), the row-column representation has different dimensions depending on the node type.



(a) Graph and Dependency Matrix (b) Row-Column Representation (c) Decision Tree

Figure 6. An Example of Row-Column Representation and Decision Tree

After the dependency matrix and row-column elements are identified, Merge_Tree() builds a new decision tree if it does not yet exist, or merges graphs into the existing decision tree. Each branch, which connects from one node to another, is labeled with a row-column element. The row-column element associated to the branch is used to determine whether the algorithm needs to traverse further down nodes, or whether the input graph is isomorphic. The main part controls the overall process for the (sub)graph isomorphism test and generates a complete metamodel by traversing the decision tree.

B. Semantics Inference

Model semantics are often used to impose structural and behavioral properties, as well as govern the syntax and semantics of concrete syntax, and the value of the properties. In general, DSMLs require two types of semantics: static and dynamic semantics. Dynamic semantics describe the meaning of a model and are related to the abstract syntax. Static semantics are well-formed rules for the model and are used to constrain the concrete syntax.

As mentioned earlier, the Metamodel Inference Engine generates a single graph that is created by combining individual graph representations of domain model examples. Based on the graph representation and the cardinality tables, the MLCBD framework induces the following static constraints as fundamental constraints for visual DSMLs.

- *Identification of concrete syntax*: Some calssifiers are mandatory for every model instance, while others are optional. In our FSM example, StartState and EndState are mandatory modeling elements that should be included in all FSM domain model examples. But, State can be used optionally as shown in Table I (b) and (c). Initially, this constraint can be identified when the Metamodel Inference Engine produces a single graph representation by combining individual domain model examples. The Semantics Inference Engine verifies the concrete syntax by comparing it with the cardinality table.
- *Identification of boundedness*: In our FSM examples, all modeling elements are bounded to each other through transitions. In some DSMLs, such as the chemical structure modeling language [1], modeling elements can have an open link in one end as long as the other end is bounded to other modeling elements. If modeling elements are not bounded to other modeling elements, they can be represented as isolated nodes, forming cyclic paths, or have an infinite number of branches [6]. Thus, the Semantics Inference Engine needs to search for those cases, and cardinality is used as additional information for the boundedness check. For example, a cyclic path around State may be marked as unbounded. But, it can be determined as bounded because the cardinality between StartState/EndState and State is set to (1, 1).
- *Identification of relationship constraint*: Some DSMLs may constrain the use of a relationship. For example, in a UseCase diagram, relationship types known as "extend" and "include" are only applicable between UseCases. When those relationships are used to link between UseCases and an Actor, most tools warn about a violation. The Semantics Inference Engine infers all possible relationship constraints and provides the constraints for the end-users to select and apply according to their needs.
- *Identification of participation constraints for each modeling element*: Some modeling elements can be used only once in a model. StartState and EndState are examples of this constraint. To derive this constraint, the inference engine looks at the cardinality table and finds cells that are set to 1 for both minimum and maximum cardinality.

IV. Related Work

The core of the MLCBD framework is an inference engine for inducing a metamodel and its static constraints from a set of domain model examples. Metamodel inference has been researched in the past as an instance of grammar inference in order to recover the lost metamodel and architecture from example instance models.

Javed et al. [9] introduced a grammar inference system, named MARS, for metamodel recovery. Their goal was to induce a metamodel from a set of domain model examples, which are the legacy of previous modeling tools when additional information such as syntax and a language manual are lost. To induce a metamodel, the MARS system translates a set of domain model examples into MRL (Model Representation Language), which is a small domain-specific language. The MRL is used to translate domain model examples into a context-free grammar, and then that grammar is used to infer a metamodel. The MRL is introduced as an intermediate representation between a set of domain model examples and the MARS inference engine. In our approach, a graph is used to resolve the representation mismatch issue instead of introducing a new DSL such as the MRL. In addition to this difference in representation and inference, the purpose of MARS and MLCBD is also different. MARS is focused on recovering a lost metamodel from a set of legacy instances, whereas MLCBD assumes that domain experts are available when creating a brand new modeling language for a new domain, and a visual DSML is built based on interaction with the user in the form of supplied examples of the language that they desire. Thus, in MARS, the underling metamodel has already been defined in the past, but MLCBD allows new DSMLs to be elaborated from domain expert interaction.

CacOphoNy [5] proposed a metamodel-driven process for reconstructing software architecture based on a metamodel. The approach defined six major steps, which are performed iteratively and incrementally to reconstruct software architecture. To reconstruct software architecture, the approach collects domain-specific knowledge through interviewing domain experts and storing domain information in a database. From this information, a metamodel is designed by searching and integrating appropriate metamodel components from the inventory. As the approach focuses on a methodological guide, no technical descriptions about the algorithm, implementation, and automation are discussed. However, the notion of metamodel integration needs to be considered even in our approach when a set of domain model examples are matched to more than one metamodel design pattern.

To ease the semantics specification, Qattous et al. [15] adopted a By-Example technique for specifying constraints for modeling languages. As specifying constraints requires language development expertise and is a mundane and error-prone task, especially for domain experts who do not have language development expertise, they implemented a Specification By-Example technique in a metamodeling tool and compared the approach with a wizard-based approach. The metamodel-based approach showed better results compared to a wizard-based approach with respect to accuracy of specification, elapsed time for completion, and user satisfaction. However, the approach needs intervention of language experts if users do not understand a series of constraint specification tasks. Similar to the Qattuos approach, our approach infers the semantics of the visual

DSMLs (in particular, the static constraints of the visual DSMLs), which can be applicable for most visual DSMLs, by the Semantics Inference Engine. However, unlike the Qattuos approach, our approach infers the static semantics from the domain model examples, which are created by the By-demonstration technique. Thus, our approach can specify (static) semantics without intervention of a language expert.

V. CONCULUSION AND FUTURE WORK

In this paper, we introduce a systematic and (semi)automated approach to create visual DSMLs from a set of domain model examples, while resolving issues of visual DSML development from an end-user perspective. The approach identifies concrete syntax from a set of domain model examples and then infers abstract syntax in the form of a metamodel. To identify the concrete syntax and infer a metamodel, a set of domain model examples provided by an end-user are transformed into a graph representation. By transforming a set of domain model examples into a graph representation, the framework can resolve representation mismatch issues. In addition, incremental and iterative metamodel inference can reduce computation complexity.

We illustrated our approach with simple FSM instances. But, in practice, visual DSMLs often have complex language constructs such as types, containment, nesting and composite structures. We will consider how to represent such complex language constructs with a graph representation and to minimize computation complexity of metamodel inference.

Another issue is the absence of negative domain model examples. The approach induces a metamodel only with a set of positive domain model examples. The generated DSML may allow domain experts to design domain models that do not belong to the domain. To prevent this problem, the framework needs an extra verification step that can generate positive and negative samples from an induced metamodel and then reflect negative model instance information from user feedback (i.e., from an inferred metamodel, generate all possible model instances allowed by the metamodel and ask the user to confirm correctness).

Although we can infer static constraints, the most significant limitation of our approach is the need to infer dynamic semantics of the visual DSML. Because semantics add domain-specific knowledge and help in reasoning about the desired properties of modeling elements, we need to list and identify the categories of semantics that can be applied generally to all visual DSMLs. Our future work is focused on inferring and expressing these semantics formally and declaratively in a manner that continues to support the idea of an end-user by-example approach.

ACKNOWLEDGMENT

This work is supported by NSF CAREER award CCF-1052616.

REFERENCES

[1] J. L. Bentley, L. W. Jelinski, and B. W. Kernighan, “Chem-a program for phototypesetting chemical structure diagrams,” Computers & Chemistry, vol. 11, no. 4, pp. 281-297.

[2] D. Blostein and A. Schürr, “Computing with Graphs and Graph Transformations,” Software Practice and Experience, vol. 29, no. 3, pp. 197-217, 1999.

[3] H. Cho, Y. Sun, J. Gray, and J. White, “Key Challenges for Modeling Language Creation By Demonstration,” ICSE 2011 Workshop on Flexible Modeling Tools, Honolulu HI, May 2011.

[4] H. Cho, and J. Gray, “Design Patterns in Metamodels,” The 11th Workshop on Domain-Specific Modeling, Portland OR, Oct. 2011.

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

[6] M. Hagenbuchner, A. Sperduti, and A. C. Tsoi, “Graph self-organizing maps for cyclic and unbounded graphs,” Neurocomputing, vol. 72, no. 7-9, pp.1419-1430, Mar. 2009.

[7] C. De La Higuera, “A bibliographical study of grammatical inference,” Pattern Recognition, vol. 38, no. 9, pp. 1332-1348, Sep. 2005.

[8] P. Hudak, “Building domain-specific embedded languages,” ACM Comput. Surv., vol. 28, no. 4es, Dec. 1996.

[9] F. Javed, M. Mernik, J. Gray, and B. R. Bryant, “MARS: A metamodel recovery system using grammar inference,” Information and Software Technology, vol. 50, no. 9-10, pp. 948-968, Aug. 2008.

[10] S. Kelly and J-P. Tolvanen, Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Computer Society Press, 2008.

[11] T. Mens, and P. V. Gorp, “A Taxonomy of Model Transformation,” Electronic Notes in Theoretical Computer Science, vol. 152, pp. 125-142, Mar. 2006.

[12] B. T. Messmer and H. Bunke, "A decision tree approach to graph and subgraph isomorphism detection," Pattern Recognition, vol. 32, no. 12, pp. 1979-1998, Dec. 1999.

[13] R. S. Michalski, “A theory and methodology of inductive learning,” Artificial Intelligence, vol. 20, no. 2, pp. 111-161, Feb. 1983.

[14] G. Rozenberg (ed.). Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, vol.1-2. World Scientific, Singapore, 1997.

[15] H. Qattous, P. Gray, and R. Welland, “An empirical study of specification by example in a software engineering tool,” In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10), Sep. 2010, Bolzano-Bozen, Italy.

[16] GME. http://www.isis.vanderbilt.edu/Projects/gme/

[17] MetaCase. http://www.metacase.com/