Proactive Modeling: Auto-Generating Models From Their Semantics and Constraints*

Tanumoy Pati, Dennis C. Feiock, and James H. Hill
Dept. of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
tpati@cs.iupui.edu, dfeiock@iupui.edu, hillj@cs.iupui.edu

ABSTRACT

This paper discusses how DSML semantics and constraints enable *proactive modeling*—a form of model intelligence that foresees model transformations, automatically executes them, and prompts the modeler for assistance when necessary. This paper also shows how we integrated proactive modeling into the Generic Modeling Environment (GME). Our experience using proactive modeling shows that it can reduce modeling effort by both automatically generating required model elements, and guiding modelers to select what actions should be executed on the model.

Keywords

proactive modeling, model intelligence, domain-specific modeling language, model-driven engineering

1. INTRODUCTION

Model-Driven Engineering (MDE) [5] powered by domain-specific modeling languages (DSMLs) [3] allows developers to define the abstractions and semantics of a given domain using intuitive graphical representations, and define constraints that govern interactions of the abstractions. The DSMLs are then used by modelers to model concepts for the target domain. Lastly, model interpreters transform constructed models into concrete artifacts .

Traditionally, the process of using DSMLs to create models is primarily a manual process. This means that it is the responsibility of the modeler to manually craft and manage their models, such as adding and deleting model elements, setting attributes, and ensuring constraints are not violated. Because creating a model can be a tedious and time-consuming process—especially when with dealing with complex DSMLs and large models—model intelligence techniques (*e.g.*, constraint solvers [1, 6, 7] and model guidance [2, 8]) have emerged as an approach to alleviate this concern. For example, modelers can manually create a *partial model* and use constraint solvers to automatically generate a complete solution. Likewise, modelers can select a model element and model guidance engines will highlight valid associations (*e.g.*, connections and references), or how to resolve violated constraints after the model has been created.

Although model intelligence is improving the usability of DSMLs, it is still plagued by manual processes. As highlighted above, it is the modeler's responsibility to manually create a partial model before invoking constraint solvers. Likewise, model guidance techniques engage the modeler *after* they make a selection. It, however, can be *hard* for the modeler to know what actions can occur next—especially if the modeler is not familiar with the DSML. Likewise, "fixing" a model implies the modeler has to first create a model. This is typically a manual process through trial-and-error, even with current state-of-the-art model guidance techniques. Finally, it is the modeler's responsibility to manage model consistency and correctness above and beyond manually, or automatically, evaluating constraints after completing actions (*i.e.*, reactive constraint checking).

Because of the current challenges discussed above, there is need for improved model intelligence techniques that better assists modelers in the modeling process. Based on this understanding, the main contributions of this paper are as follows:

- It introduces proactive modeling, which is a form of model intelligence that foresees plausible model transformations and executes them automatically, and prompts the modeler for assistance when needed; and
- It shows how proactive modeling is implemented in GME as a GME add-on (*i.e.*, a domain-independent event handler) named the *Proactive Modeling Engine (PME)*.

Finally, experience from applying PME to a simple DSML and other DSMLs (not discussed in this paper) show that it can significantly reduce modeling effort. It, however, is necessary to provide mechanisms that allow modelers control how engaged proactive modeling is with the model and modeler.

Paper organization. The remainder of this paper is organized as follows: Section 2 introduces an example DSML that motivates the need for proactive modeling; Section 3 presents an overview of proactive modeling; Section 4 discusses the design and implementation of the PME; Section 5 compares our work on proactive modeling with other related works; and Section 6 provides concluding remarks and lessons learned.

*This work was sponsored in part by the Maritime Operations Division (MOD) of the Australian Defense Science and Technology Organization (DSTO).

2. THE LIBRARY MANAGEMENT SYSTEM

The section introduces the Library Management System (LMS) example, which is a system that helps librarians track their book inventory, and patrons who have borrowed books from the library.

Modeling elements. There are many different ways to compose a metamodel for the LMS. Figure 1 shows one simple example metamodel for the LMS. As shown in this figure, the root element is a Library model element. The Library model element contains five basic model elements: Book, Patron, Librarian, HRStaff and Shelf; two connection elements: Borrows (representing a patron borrowing the connected book from the library), and Employees (representing a librarian hired by the connected HR staff); one reference element Patronref that refer to patrons belonging to other libraries.

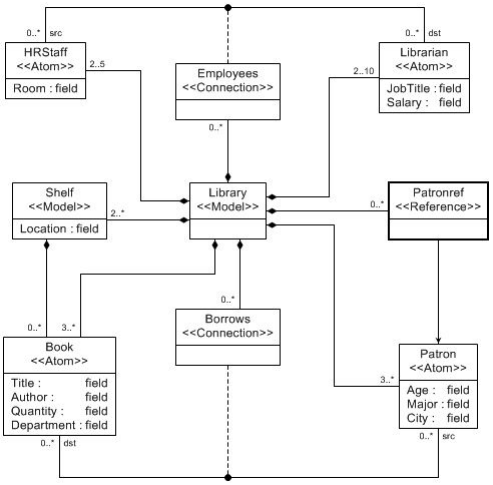

Figure 1: An example GME metamodel for the Library Management System.

Constraints. The LMS has several constraints that govern its model. Because the LMS is created in GME, the Object Constraint Language (OCL) [4] is used to express the LMS's constraints. Some of the constraints for the LMS are as follows:

- **Required city.** As shown in Listing 1, this constraints checks that all patrons who are a member of the library are from Indianapolis.

```
1  self.City = "Indianapolis"
```

Listing 1: OCL constraint showing the required city.

- **Book borrowing condition.** As shown in Listing 2, this constraint validates that a patron can only borrow books that are relevant to his/her field. For example, a Computer Science student can only borrow books that are relevant to the field of Computer Science.

```
1  self.connectedFCOs(Borrows)->
2      forAll(p:Book | self.Major = p.Department)
```

Listing 2: OCL constraint showing the book borrowing condition.

- **Patron referencing condition.** As shown in Listing 3, this constraint checks that the reference model element refers to a patron that belongs to another library.

```
1  self.refersTo().parent() <> self.parent()
```

Listing 3: OCL constraint showing patron referencing condition.

An example model. Figure 2 shows an example model for the LMS created using the metamodel shown in Figure 1.

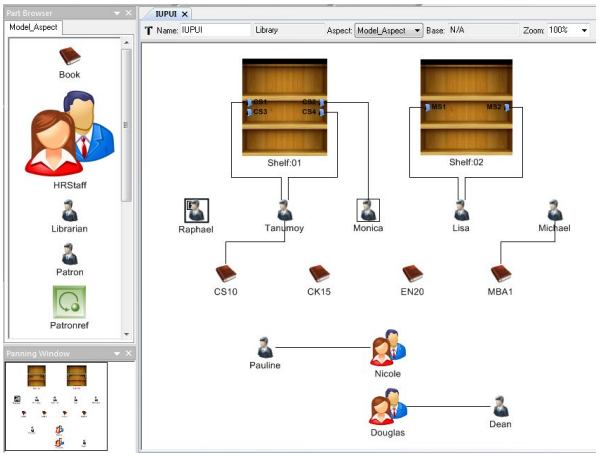

Figure 2: An example model for the Library Management System.

3. OVERVIEW OF PROACTIVE MODELING

This section provides a detailed overview of proactive modeling in DSMLs.

3.1 The Goal of Proactive Modeling

The term *proactive modeling* translates directly to foreseeing modeling. The main goal of proactive modeling therefore is to automate—as much as possible—the modeling process by foreseeing valid model transformations (*i.e.*, those that must be executed manually by a modeler), and automatically executing them. If there are optional model transformations, then proactive modeling queries the modeler for what model transformation to execute, and executes the selected model transformation (similar to model guidance).

With this in mind, proactive modeling focuses on automating the following aspects of the modeling process:

- **Automated model creation.** This aspect of proactive modeling involves automatically creating different model elements when a related model element is first created. For example, when a Library model element is added to the model, all its child model elements (*e.g.*, Book, Patron, and Librarian) should be automatically added to the Library model element up to the required quantity. This is different from current model intelligence techniques in that the current techniques do not support auto-generating elements in the required quantity, or they do not support auto-generation at all, unless the modeler manually creates a partial model.
- **Decision-making.** This form of proactive modeling involves presenting the modeler with a list of valid model transformations (or actions) (*e.g.*, create a connection, adding a reference, and adding a new model element) that can occur

based on the current state of the model. After selecting an action, proactive modeling executes the action. For example, when a modeler wants to add a Patronref model element, proactive modeling presents the modeler with a list of all the possible Patrons that the Patronref model element can reference. Upon selecting a Patron model element, the reference is auto-generated. This form of automation, which requires human-intervention, is different from current model intelligence techniques in that it is triggered automatically when automated modeling reaches a stopping point.

Figure 3 provides a high-level overview of the proactive modeling process. As shown in this figure, proactive modeling resides between the modeler and the model. The proactive modeling engine (1) automatically adds modeling elements to the model. When the proactive modeling engine reaches a stopping point, (2) it then interacts with the modeler to select what transformations to apply to the model. In the end, the modeler does not interact directly with the model. Instead, the modeler interacts with the model through the proactive modeling engine.

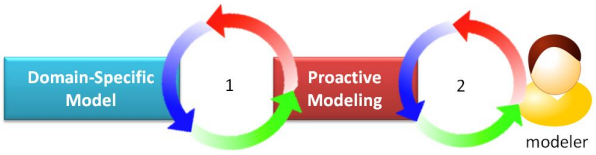

Figure 3: Overview of the proactive modeling process.

3.2 Insights for Realizing Proactive Modeling

In order for proactive modeling to function it must get its insight from somewhere. Because a DSML is well-defined, it is possible for proactive modeling to gain insight from analyzing a DSML as follows:

- **Semantic analysis.** Semantic analysis is the process of analyzing a DSML's metamodel at runtime to discover information about its model elements. For example, when adding a Patronref element to the model, semantic analysis of the LMS metamodel (see Figure 1) will identify that a Patronref model element can reference a Patron model element. By performing semantic analysis, proactive modeling can collect any type of information that is relevant to a model element without being bound to the target DSML.
- **Constraint analysis.** Constraint analysis is the process of parsing and analyzing a DSML's constraints collected during the semantic analysis process. For example, semantic analysis of Patronref returns the constraint shown in Listing 3, which is then parsed and evaluated to generate the list of possible Patrons that can be referenced. By performing constraint analysis, proactive modeling can not only evaluate constraints, but also use them to provide modeling guidance and auto-generate model elements.

There can be other types of analysis integrated into proactive modeling, such as layout analysis where actions are performed based on the layout of modeling elements, and user-intent analysis where actions are performed based on past knowledge of how a modeler creates a model, but we have scoped the work to these two forms of analysis. This is because semantic and constraint analysis is based on static, well-defined information.

3.3 Mutable vs. Immutable Constraints

As explained above, it is possible to analyze a DSML's constraints and determine what elements should be added to the model, or a list of valid modeling actions. For example, saying the number of Patrons must equal 3 means that proactive modeling can automatically ensure the number of patrons is always 3 since 3 does not change. On the other hand, saying that the number of patrons must equal the number of books means that proactive modeling needs modeler intervention because both the number of patrons and books can be modified.

Based on the two examples above, constraints can be classified as either *mutable* or *immutable*. A mutable constraint is a constraint that evaluates two variable expressions. An immutable constraint is a constraint that evaluates a variable expression and a constant expression. Because an immutable constraint has constant values, it is possible to automatically execute actions that transform the model towards the constant value. Mutable constraints, however, require modeler intervention because one model element must act as the constant value in the constraint evaluation.

4. THE DESIGN AND IMPLEMENTATION OF PROACTIVE MODELING IN GME

This section discusses how proactive modeling is realized in GME via the *Proactive Modeling Engine*.

4.1 Mapping Proactive Modeling into GME

Both semantic and constraint analysis can be integrated into GME at run-time without being bound to a specific DSML. Before going into the implementation details of proactive modeling in GME, it is first necessary to understand how the analysis maps into GME. Based on the functionality of GME, we have classified constraints into the following four categories:

- **Containment constraints.** Modelers can define *multiplicity* specification (also known as cardinality) on containment relationships. The multiplicity specification determines the acceptable number of containment relationships allowed between a parent model element and a child element. For example in Figure 1, the containment relationship between Book model element and Library model element has a multiplicity of `3..*`. This means that a Library model element should contain at least 3 Book model elements at all times.
- **Attribute constraints.** Modelers can define constraints that validate attribute values with respect to expected values or other elements. For example, Listing 1 illustrates that the expected city for a patron is "Indianapolis".
- **Association constraints.** Modelers can define association relationships between two model elements using a Connection model element. Modelers can refine association relationships using constraints and reduce the possible destination model elements of a connection. For example, Listing 2 shows an association constraint imposed on Patron that governs the Borrows connection between a Book and Patron model element.
- **Reference constraints.** Modelers can define aliases (or pointers) to other model elements by using the Reference model element. For example, Figure 1 shows how the LMS metamodel defines a Patronref model element that refers to Patron model elements. Modelers can also impose constraints on references, which validate that the referenced model element meets a condition. For example, Listing 3 shows a reference

constraint imposed on Patronref, which specifies that a Patronref can only reference Patrons from another library.

4.2 The Proactive Modeling Engine

Figure 4 provides an overview of the Proactive Modeling Engine (PME), which is a GME add-on that implements proactive modeling. A GME add-on is a domain-independent event handler that receives events dictating what model actions have occurred (*i.e.*, model element creation and selection). It is worth noting that if a GME add-on modifies the model, then the event that corresponds to the modification is sent to all loaded add-ons—including the add-on that modified the model. Lastly, all GME add-ons are stateful.

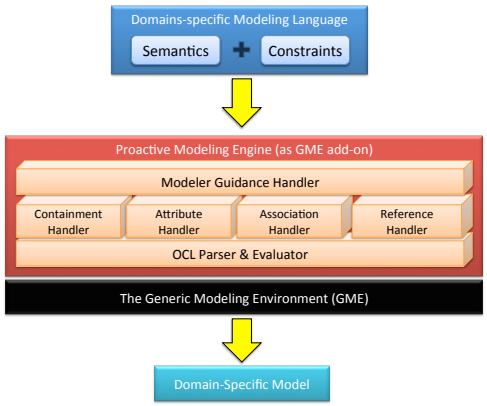

Figure 4: Architecture of Proactive Modeling Engine (PME).

As shown in Figure 4, the PME is composed of the following key components:

- **OCL parser and evaluator.** The OCL parser is responsible for parsing OCL constraints and dynamically creating an abstract syntax tree from the parsed OCL constraints. Because a GME add-on is stateful, the parsed OCL expressions are cached for retrieval later on. The OCL parser in the PME is designed and implemented using the *Boost Spirit Parser Framework* (`boost-spirit.com`). This parser works only with constraints defined in the DSML and is independent of the DSML's metamodel. This allows the OCL parser to be used as a standalone parser.
  The OCL evaluator for the PME works as follows: it is invoked by handlers on the root node of the abstract syntax tree (AST). The individual objects that form the AST are responsible for evaluating a certain aspect of the constraint (*e.g.*, a method or expression). The evaluation control traverses the AST in a top-down fashion and each object returns back the evaluated result back to its parent, stopping at the root. PME then transforms the model based on the evaluated value and information collected during semantic analysis.
- **Containment handler.** The containment handler is responsible for automating the model element creation process by resolving the containment relationships between model elements. For example, when a Library model is added to the example model shown in Figure 2, the containment handler first analyzes the LMS's metamodel to identify what model elements a Library model can contain through semantic analysis. In this case, the containment handler will identify the Book, Patron, Borrows, Shelf, HRStaff, Librarian, Employees, and Patronref model element types.
  After the containment handler completes its semantic analysis, it uses constraint analysis to parse and analyze each constraint associated with the newly created model element, by forwarding the constraints to the OCL parser and evaluator. If a constraint is a containment constraint and is violated, then containment handler auto-generates the model elements associated with that constraint until it is valid. For example, when a Library model element is added to the model, then PME will auto-generate 3 Book, 3 Patron, 2 Shelf, 2 HRStaff, and 2 Librarian model elements as shown in Figure 5.

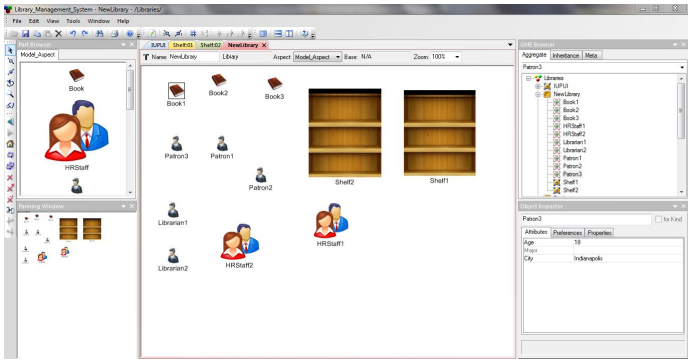

Figure 5: Model created by PME when a Library model element is added to the model.

- **Attributes handler.** The attributes handler is responsible for handling a model element's attribute values during the creation process, *i.e.*, ensuring the created object does not violate any attribute constraints. This, however, does not mean that a modeler cannot change an attribute's value after the model has been created.
  For example, when a Patron model is added to the model shown in Figure 2, the attributes handler first analyzes the LMS's metamodel to identify its attributes. The attribute handler then collects the constraints associated with the Patron model element and forwards it to the OCL parser and evaluator. The attributes handler, however, evaluates only the attribute constraints associated with Patron model element (shown in Listing 1). In this example, the value of City attribute is automatically set to "Indianapolis" (see the lower right window in Figure 5).
- **Association handler.** The association handler is responsible for identifying valid destination model elements for a given source model element when making a connection between two model elements. For example, to create a connection between a Patron model and Book model, the modeler first selects a Patron model. The selection triggers the association handler to analyze the metamodel and present the modeler with a list of valid connection types. Once the modeler selects a connection type, the association handler identifies all valid endpoint models for the selected connection type.
  The handler then collects, one at a time, the constraints associated with Patron model element (*i.e.*, the source model element) and forwards them to the OCL parser. The association handler then evaluates the association constraints, which allows it to filter any model elements that will violate its constraints. Figure 6 shows PME displaying a list of valid destination model elements for Tanumoy patron (a Computer Science student). This is similar to current state-of-the-art techniques in model guidance.

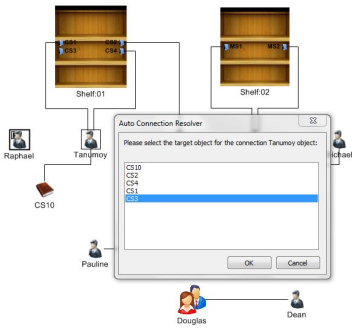

Figure 6: Dialog of valid Book elements presented to modeler when a Patron model is selected.

- **Reference handler.** The reference handler is responsible for identifying valid model elements that can be referred to by a reference model element. For example, when a modeler adds a Patronref model to the model, the reference handler gathers a list of valid model types that can be referenced by a Patronref model (*e.g.*, Patron model types). The reference handler then uses the type information to gather a list of all elements that are instances of the identified model types. For example in Figure 7, the reference handler will gather the following Patron models: Tanumoy, Monica, Lisa, Michael, Joe, Raphael, and Sam.

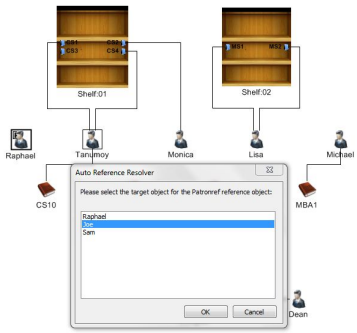

Figure 7: Dialog of valid Patron models presented to modeler when Patronref model element is added to model.

The handler then collects, one at a time, the constraints associated with selected reference model element and forwards them to the OCL parser and evaluator. The reference handler evaluates each OCL constraint with the goal of filtering the initial list of plausible model elements such that no element in the final list violates any constraints. Figure 7 shows an example of PME displaying the list of Patron model elements that validate the constraint shown in Listing 3 (*i.e.*, Joe, Raphael, and Sam) when the modeler adds a Patronref model element to the model.

- **Modeler guidance handler.** The modeler guidance handler is responsible for providing a modeler with a list of valid operations to execute when proactive modeling finishes auto-generating model elements. The operations presented to the modeler are in compliance with both the DSML's semantics and the constraints. For example, when a modeler starts a new project, the modeler guidance handler presents the modeler with the list of all the model elements that can be added to the `RootFolder`. Likewise, the modeler guidance handler prompts the modeler to select a model to operate. Upon selection, the modeler is presented with a list of operations that are specific to the selected model as shown in Figure 8.

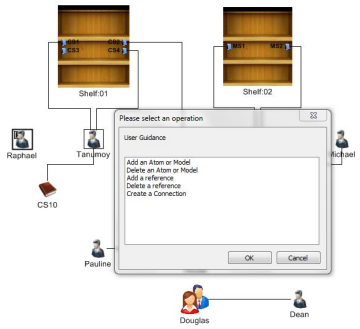

Figure 8: Dialog of valid operations for Library model.

The modeler guidance operations currently supported by PME are as follows:

- **Add a modeling element.** This operation is used to add a selected model type to a selected parent model if allowed.
- **Delete a modeling element.** This operation is used to delete a model element from the selected parent model element if allowed.
- **Create a connection.** This operation is used to create a connection between two model elements within the selected parent model element.

The modeler guidance handler therefore provides relevant and valid operations to the modelers, which reduces the modeler's decision set and can improve their modeling experience. Likewise, we can easily extend the modeler guidance handler to support other operations as we learn them.

4.3 Chain Reactions in PME

As stated above, PME is a GME add-on and a GME add-on is a reentrant component. This means that when PME modifies the model, PME will receive an event associated with the latest modification. Upon receiving the new event, PME handles the new event similar to how it handled the previous event. If there is no decision-making need on the modelers part, then PME automatically handles the event (per the discussion above). If PME requires user input, then it queries for it and proceeds.

In the best case scenario, the first model modification (*e.g.*, starting a new project, or opening an existing project) triggers PME and PME auto-generates all the elements required in the model. In this scenario, modeling effort is very low since the modeler does not have to do anything. In the worst case scenario, the modeler is prompted by PME after each modification to the model since PME is not able to automatically generate anything. In this scenario, PME is similar to manually creating a model except for the fact that PME ensures you only execute valid actions.

5. RELATED WORKS

Partial model creation. Sen et al. [6] presented a framework for generating model completion recommendations in model editors. In their approach, the metamodel is transformed into a *constraint logic program* (CLP) [6], and processed by a Prolog engine. The

processed CLP is then able to complete a partial model (*i.e.*, one that has been manually created by the modeler). Our approach extends their effort in that proactive modeling can assist in either automatically creating the partial model, or recommending what actions to take on the model. Once the modeler has a valid partial model created using PME, the modeler can use their approach to complete it.

Hessellund et al. [1] created an extension of the Eclipse Modeling Framework called *SmartEMF*. SmartEMF provides support for representing, checking, and maintaining four kinds of consistency constraints: well-formedness of individual artifacts, referential integrity across artifacts, references with additional constraints, and style constraints. Similar to Sen et al., SmartEMF provides editing guidance to the modeler by evaluating precondition constraints that exist on editing operations. Our work therefore extends Hessellund's in that it can not only provide modeling guidance to modelers but it can also automatically perform model transformations, such as automatically adding/deleting of valid model elements in accordance with the constraints.

White et al. [7] created a Domain-Specific Intelligence Framework (DSIF) that provides model guidance for large and complex models. White's approach also converts constraints to a Prolog knowledge base, and the knowledge base is used to auto-generate complete models from partial models that satisfy the original constraints. Our approach extends White's DSIF in that it can assist with creating the partial model, which is currently done manually, that is needed to auto-generate the complete model.

Decision making. Janota et al. [2] improved modeling experience by their work on Interactive model derivation, which is a process of constructing models and meta-models with the help of automatic adaptive guidance. This guidance system assists a modeler by providing a list of valid edit operations to chose from. The major work involved developing guidance algorithms for concrete modeling languages. These guidance algorithms identify transformations that *refine* the model. Our approach extends Janota's work in that proactive modeling can not only provide decision-making capabilities but also auto-generate model elements when a model element is first created, *i.e.*, automatically perform multiple editing operations. Moreover, proactive modeling also provides modelers with a sequence of valid operations to choose from after it has finished auto-generating model elements.

Constraint-driven model intelligence. White et al. [8] developed a model intelligence mechanism that guides modelers towards correct models. In White's approach, the modeler first selects a relationship type and an element for the new relationship. The model intelligence then evaluates constraints associated with the selected element. It then presents a list of valid elements that can be associated with the selected element. Our work extends White's work in that proactive modeling automates the modeling process based on the metamodel's semantics and constraints, not just its constraints. Once the proactive modeling reaches a point where it needs human intervention, it prompts the modeler for the next action. At that point, our work is similar to White's.

6. CONCLUDING REMARKS

As domain-specific models increase in both size and complexity, it will be *hard* for modeler's to cope. This was illustrated by current model intelligence solutions. It, however, is necessary to go beyond the existing model intelligence solution approaches because it will enable to continue improving the modeler's experience. As illustrated in this paper, we presented a model intelligence approach called proactive modeling. We believe that this is a new area of model intelligence has the potential to open new areas of research. Based on our experience implementing proactive modeling in GME, and applying it to several DSMLs, the following is a list of lessons learned and future research directions:

- **Assists novice modelers with learning a new DSML.** Proactive modeling guides a modeler throughout the modeling process by providing list of valid operations to choose from. Moreover, proactive modeling also enhances modeling experience through actions like auto-generation of elements, auto-reference resolving, auto-connection resolving, and automatic value entry for constrained attributes. These features of proactive modeling make it suitable for novice modelers because it prevents them from violating constraints, and helps them get through the tedious, labor-intensive, time-consuming process of manually creating a model.
- **Proactive modeling can fall victim to the "Clippy" syndrome.** Microsoft Office included an Office Assistant named "Clippy" that would try to assist the end-user based on their current actions. Unfortunately, "Clippy" was considered intrusive and annoying [9]. It is possible that proactive modeling can fall victim to this condition, which we call the "Clippy" syndrome. It is therefore critical that proactive modeling finds a way to be useful without being too intrusive. Otherwise, modelers will not want to use proactive modeling engines regardless of their benefits.

PME is available in open-source format, and integrated into the CoSMIC tool suite. CoSMIC can be download from the following location: `www.dre.vanderbilt.edu/cosmic`.

7. REFERENCES

[1] A. Hessellund, K. Czarnecki, and A. Wąsowski. Guided development with multiple domain-specific languages. *Model Driven Engineering Languages and Systems*, pages 46–60, 2007.

[2] M. Janota, V. Kuzina, and A. Wąsowski. Model construction with external constraints: An interactive journey from semantics to syntax. *Model Driven Engineering Languages and Systems*, pages 431–445, 2008.

[3] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment, 2001.

[4] Object Management Group. *Object Constraint Language*, 2006.

[5] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[6] S. Sen, B. Baudry, and H. Vangheluwe. Domain-specific model editors with model completion. *Models in Software Engineering*, pages 259–270, 2008.

[7] J. White, D. C. Schmidt, A. Nechypurenko, and E. Wuchner. Domain-Specific Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains. *GPCE4QoS (October 2006)*, 2006.

[8] J. White, D. C. Schmidt, A. Nechypurenko, and E. Wuchner. Model intelligence: an approach to modeling guidance. *UPGRADE*, 9(2):22–28, 2008.

[9] Wikipedia. Office Assistant. `http://en.wikipedia.org/wiki/Office_Assistant`.

APPENDIX

A. DETAILS OF LIBRARY MANAGEMENT SYSTEMS

A.1 Constraints

This section presents the list of constraints that have not been introduced in Section 2, shown as follows:

- **Minimum number of books required.** This constraint is used to enforce the minimum number of books that a library must contain. As shown in Listing 4, this constraints checks that a Library has at least 3 Book model elements. The constraint shown in this listing is automatically generated by GME from the cardinality expressed on the containment connection between the Library and Book model element in Figure 1.

```
1  let partCount = self.parts( "Book" )->size in
2  (partCount >= 3)
```

Listing 4: OCL constraint showing minimum number of books required.

- **Minimum number of patrons required.** This constraint is used to enforce the minimum number of patrons that a library must contain. As shown in Listing 5, the constraint checks that a Library has at least 3 Patron model elements. The constraint shown in this listing is automatically generated by GME from the cardinality expressed on the containment connection between the Library and Patron model element in Figure 1.

```
1  let partCount = self.parts( "Patron" )->size in
2  (partCount >= 3)
```

Listing 5: OCL constraint showing minimum number of patrons required.

- **Minimum number of shelves required.** This constraint is used to enforce the minimum number of shelfs that a library must contain. As shown in Listing 6, the constraint checks that a Library has at least be at least 2 Shelf model elements. The constraint shown in this listing is automatically generated by GME from the cardinality expressed on the containment connection between the Library and Shelf model element in Figure 1.

```
1  let partCount = self.parts( "Shelf" )->size in
2  (partCount >= 2)
```

Listing 6: OCL constraint showing minimum number of shelves required.

- **Number of HR staff required.** This constraints checks that a Library model contains at least 2 and at most 5 HRStaff elements as shown in Listing 7. The constraint shown in this listing is automatically generated by GME from the cardinality expressed on the containment connection between the Library and HRStaff model element in Figure 1.

```
1  let partCount = self.parts( "HRStaff" ) -> size in
2  ((partCount >= 2) and (partCount <= 5))
```

Listing 7: OCL constraint showing the number of required HR staff.

- **Number of librarians required.** This constraint checks the the minimum and maximum number of librarians that work at the library. As shown in Listing 8, the constraint checks that a library has at least 2 and at most 10 Librarian model elements. The constraint shown in this listing is automatically generated by GME from the cardinality expressed on the containment connection between the Library and Librarian model element in Figure 1.

```
1  let partCount = self.parts( "Librarian" )->size in
2  ((partCount >= 2) and (partCount <= 10))
```

Listing 8: OCL constraint showing the number of required librarians.

- **Required age.** This constraint checks the age of each patron that is a member of the library. As shown in Listing 9, a patron must be at least 18 years of age. This is a domain-specific constraint that is added manually by the person who created the metamodel.

```
1  self.Age >= 18
```

Listing 9: OCL constraint showing the required age.

- **Salary range.** This constraint checks that a librarians salary is within the accepted salary range. As shown in Listing 10, the salary should be in the range $30,000 to $40,000. This is a domain-specific constraint that is added manually by the person who created the metamodel.

```
1  (self.Salary >= 30000) and (self.Salary <= 40000)
```

Listing 10: OCL constraint showing the salary range of a librarian.

- **Book borrowing limit.** This constraint validates that a patron can only borrow a certain number of books. As shown in Listing 11, a patron can borrow only 5 books from the library.

```
1  self.attachingConnections(Borrows)->size <= 5
```

Listing 11: OCL constrain showing the book borrowing limit.