A WYSIWYG Approach to Support Layout Configuration in Model Evolution

Yu Sun[1], Jeff Gray[2], Philip Langer[3], Gerti Kappel[4], Manuel Wimmer[4], Jules White[5]

[1]Department of Computer and Information Sciences, University of Alabama at Birmingham
[2]Department of Computer Science, University of Alabama
[3]Department of Telecooperation, Johannes Kepler University
[4]Business Informatics Group, Vienna University of Technology
[5]Department of Electrical and Computer Engineering, Virginia Tech

ABSTRACT

Model evolution has become an essential activity in software development with the ongoing adoption of domain-specific modeling, which is commonly supported and automated by using model transformation techniques. Although a number of model transformation languages and tools have been developed to support model evolution activities, the layout of visual models in the evolution process is not often considered. In many cases, after a transformation is performed the layout of the resulting model must be manually rearranged, which can be time consuming and error-prone. The automatic layout arrangement features provided by some modeling tools usually do not take a user's preferences or the semantics of the model into consideration, and therefore could potentially alter the desired layout in an undesired manner. This paper describes a new approach to enable users to specify the model layout as a demonstrated model transformation. We applied the Model Transformation By Demonstration (MTBD) approach and extended it to let users specify the layout information using the concept of "What You See Is What You Get" (WYSIWYG), so that the complex layout specification can be simplified.

KEYWORDS

INTRODUCTION

With the ongoing adoption of Domain-Specific Modeling (DSM) (Gray et al., 2007), models are emerging as first-class entities in many domains and play an increasingly significant role in every phase of software development (i.e., from system requirements analysis and design, to software implementation and maintenance). In the DSM context, whenever a software system needs to evolve, the models used to represent the system should evolve accordingly. For instance, system design models often need to be changed to adapt to new system requirements (Greenfield & Short, 2004). As an additional example, it is sometimes necessary to apply model refactoring (France et al., 2003) to optimize the internal structure of the implementation models (i.e., models used to generate implementation code through code generators). Furthermore, models used to control the deployment of a software system are occasionally scaled up for the purpose of improving performance (Sun et al., 2009a).

Although manual model evolution is often tedious and error-prone, automating complex model evolution tasks using model transformation technologies has become a popular practice (Gray et al., 2006). A number of executable model transformation languages (e.g., QVT (http://www.omg.org/cgi-bin/doc?ptc/2005-11-01, 2010), ATL (Jouault et al., 2008)) have been developed to enable users to specify model transformation rules, which take an input model and evolve it to produce an output model automatically.

Open problems. Although the implementation of model evolution concerning the abstract syntax has been well-supported, the layout of models is rarely considered in the traditional model evolution process. Most evolution efforts focus only on the semantic aspects of the evolution (e.g., adding or removing necessary model elements and connections, modifying attributes of model elements), and often ignore

model layout configuration concerns during the evolution (e.g., positions of model elements, font, color and size used in labels). For instance, executing a set of model transformation rules to add model elements and connections will sometimes lead to placing all the newly created elements in a random location in the model editor.

Ignoring the desired layout after model evolution has a strong potential to undermine the readability and understandability of the evolved model, and may even unexpectedly affect the implicit semantics under certain circumstances. For example, users may accidentally misunderstand the system because of a disordered layout (e.g., a sequence of actions to be executed is represented by a set of nodes with arrows indicating the sequence, but a disordered arrangement of the nodes may lead to a challenge in identifying the correct execution order). Furthermore, the positions of model elements and connections may correspond to special coordinates in the real world, such that an unoptimized layout could lead to unexpected problems for the actual system (e.g., the configuration of the actual hardware devices and cables might be based on the positions of model elements and connections representing them, or the color of the elements might represent the running status of the actual devices). It may be possible to incorporate the layout information related with the implicit semantics into the metamodel as part of the abstract syntax, but a change to the metamodel may trigger further model migration problems (Sprinkle, 2003). Although it is very direct to manually adjust the layout, it becomes a tedious, timing-consuming task when a larger number of model elements are involved in the model evolution process. Therefore, while the semantic concerns of model evolution have been implemented and automated, it is indispensible to realize the automatic configuration of the layout as part of the model evolution process.

The most commonly used approach to automatically arrange the layout of models is to apply layout algorithms (Battista & Tamassia, 1993; Misue et al., 1995) after the evolution process. A number of modeling tools (e.g., GMF (http://www.eclipse.org/modeling/gmf, 2011), GEMS (http://www.eclipse.org/gmt/gems, 2011), GME (Ledeczi et al., 2001), MetaCase+ (http://www.metacase.com, 2011)) provide automatic layout functionality in their model editors using specific algorithms. They can rearrange the layout of the models and make them more readable by avoiding the overlaps of model elements and connections, adding blank spaces among model elements, or grouping the same type of elements together. However, most of these algorithms do not consider the implicit semantics of the model elements and connections; the result being that a readable model does not necessarily result in an optimized system if part of the system implementation depends on the layout configuration. Furthermore, fixed layout algorithms usually cannot consider the underlying mental map of individual users (i.e., a user's understanding of the relationship between the entities in a diagram) (Misue et al., 1995) into consideration. Although a user might prefer to see different types of model elements grouped closely, the automatic layout algorithm might destroy the user's mental map by separating them. Although a few algorithms integrate implicit semantic issues or common mental maps (e.g., placing the parent model elements being extended above their children model elements), they are often fixed in the model editors and cannot be customized easily according to a user's preference, which is inadequate for handling various kinds of implicit semantic issues and mental maps.

An alternative to configuring the layout is to change the layout properties as part of the model evolution using a model transformation process. When specifying model transformation rules to evolve the semantic aspect of the model, extra rules may be given to handle the layout configuration. Although this offers a flexible way to enable users to customize the preferred implicit semantics and mental maps, some drawbacks exist. First of all, it forces the layout to be a crosscutting concern that becomes coupled with the semantics of the model transformation. In addition, testing and debugging the layout configuration are done by running the transformation and checking the final model, which is not direct and convenient. Imagine configuring the positions of a large number of newly added elements, which may require a great deal of effort to finally confirm the precise and desirable coordinate values for all of the modeling elements. Finally, implementing model transformations usually requires the user to know a model transformation language and the metamodel definition of the domain. For general users who are not

familiar with model transformation languages or abstract metamodel definitions, they may be prevented from configuring the desired layout for the models they are using.

Therefore, a desirable approach to configure the model layout concerns in model evolution tasks should include the following features:

1) It should enable users to customize the layout configuration flexibly in order to realize their desired implicit semantics and mental maps.
2) It should be separated clearly from the semantic aspect of the model evolution.
3) It should enable end-users to configure and test the result using the notation related to their domain.
4) It should be at a level of abstraction that is appropriate for end-user adoption, and not tied to low-level accident complexities of the transformation process.

Solution approach → Configuring model layout in a WYSIWYG style. This chapter presents an innovative approach to enable users to specify and customize the preferred layout information by directly showing and demonstrating the layout configuration.

This approach is based on the idea of Model Transformation By Demonstration (MTBD) (Sun et al., 2009; Langer et al., 2010), which aims to simplify the implementation of model transformations by demonstrating specific model transformation tasks on concrete examples, and then inferring the generic transformation patterns automatically. With MTBD, users do not need to know any model transformation languages and are fully isolated from abstract metamodel definitions. MTBD has already been applied successfully in a number of model evolution applications (Sun et al., 2009a; Sun et al., 2009b; Langer et al., 2010), showing improvement in the efficiency and simplicity of implementing model transformations.

By following and extending the concepts of MTBD, we have developed an approach that enables users to demonstrate the desired layout configuration in a graphical model editor. After demonstrating the semantic concerns of the model evolution using MTBD, users can continue to select target model elements and place them at the correct positions. At the same time, the underlying MTBD engine records all of the user's operations and then generates a transformation pattern that incorporates both the semantic evolution and the layout configuration. Various options can be applied when specifying the positions, such as when model elements are placed using absolute coordinates in the model editor. Users can also choose one or multiple elements as a reference to configure the relative coordinates, or the reference could be the whole model instance whereby users configure the relative coordinates to the boundary of the model. Moreover, other layout information such as the font, color and size used in the model editor can also be configured and integrated into the model transformation.

By demonstrating the layout configuration directly, rather than specifying it explicitly in model transformation rules, users are able to customize their desired layout and preserve their mental maps (or other implicit semantic issues) without knowing any model transformation language or metamodel definition. The approach also offers a more convenient environment to give precise positions as well as to test and debug the resulting layout transformation. The demonstration of layout configuration occurs after the demonstration of the semantic evolution, so that the two concepts are separated without being tangled as crosscutting concerns.

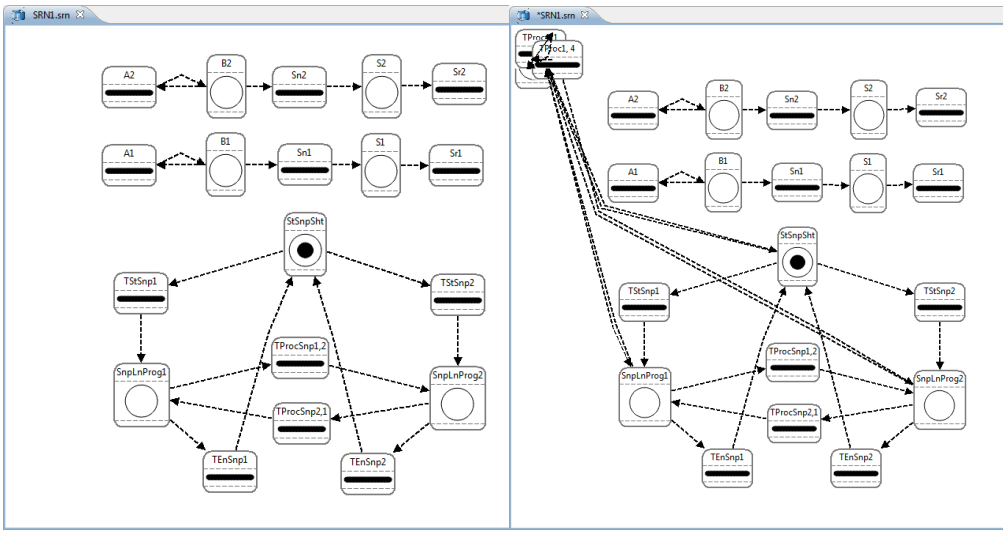
The rest of the chapter is organized as follows. We first illustrate the problems and challenges of layout configuration during model transformation using two motivating examples, followed by the analysis on the related model configuration techniques. Then, the MTBD framework is introduced, which is the basis for the layout configuration approach. To solve the two motivating examples, we will present the layout configuration extensions to MTBD and the implementation details. Finally, we offer concluding remarks and summarize future work.

MOTIVATING EXAMPLES

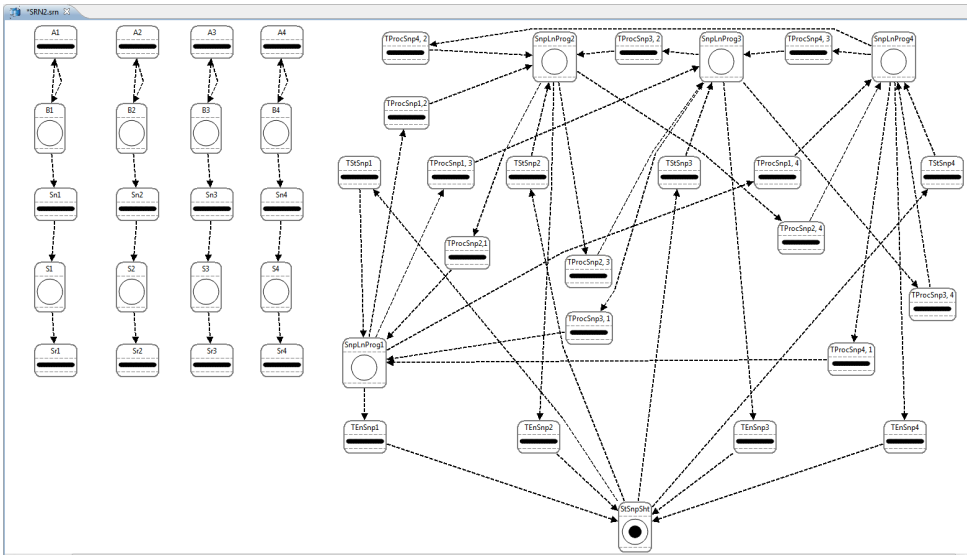
In this section, we illustrate the problems of layout configuration during model evolution using two motivating examples. For each example, we first explain the model evolution scenario as the underlying context, followed by showing what the model will look like after traditional model transformation is applied without layout configuration. We also explain why the fixed layout algorithms cannot handle most implicit semantic issues and user mental maps. Additionally, an excerpt of the model transformation rules written in a specific model transformation language will be given to explain why specifying the layout configuration explicitly with manually created model transformation rules is not a preferred approach.

Evolving Stochastic Reward Nets Models

Stochastic Reward Nets (SRNs) (Muppala et al., 1994) can be used for evaluating the reliability of complex distributed systems. SRNs have been used extensively for designing and modeling reliability and performance of different types of systems. The Stochastic Reward Net Modeling Language (SRNML) (Kogekar et al., 2006) was developed to describe SRN models of large distributed systems, which share similar goals with performance-based modeling extensions for the UML, such as the schedulability, performance, and time profiles. For example, the SRN model defined by SRNML in Figure 1a depicts the Reactor pattern (Schmidt et al., 2000) in middleware for network services, providing mechanisms to handle synchronous event demultiplexing and dispatching.

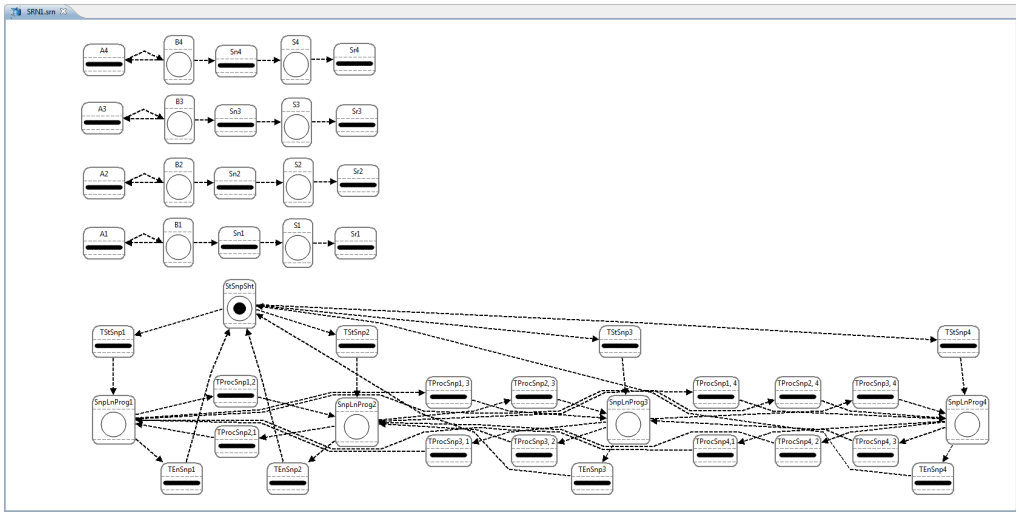


a) SRN model before evolution **b)** SRN model after evolution with configuration layout



c) SRN model after evolution with layout configured using auto-layout function

Figure 1. SRN models



d) SRN model after evolution with desired layout configuration

Figure 1. SRN models (continued)

In the Reactor pattern, an application registers an event handler with the event demultiplexer and delegates the incoming events to it. On the occurrence of an event, the demultiplexer dispatches the event to its corresponding event handler by making a callback. An SRN model consists of two parts: the event types handled by a reactor (the top part of Figure 1a) and the associated execution snapshot (the bottom part of Figure 1a). The execution snapshot depicts the underlying mechanism for handling the event types, so any changes made to the event types will require corresponding changes to the snapshot.

Model evolution scenario in SRNML: A typical SRN model evolution scenario arises from the addition of new event types and connections between their corresponding event handlers. As shown in Figure 1d, when two new event types (*3* and *4*) need to be modeled, two new sets of event types and connections (i.e., from *A3* to *Sr3*, from *A4* to *Sr4*) should be added. Also, the snapshot model should be scaled accordingly by adding new *Snapshot Places* (i.e., *SnpLnProg3*, *SnpLnProg4*), *Snapshot Transitions* from starting place to end place (i.e., *TStSnp3*, *TEnSnp3*, *TStSnp4*, *TEnSnp4*), *Snapshot Transitions* between each new place and each existing place (i.e., *TProcSnp3,1*, *TProcSnp1,3*, *TProcSnp3,2*, *TProcSnp2,3*,

TProcSnp4,1, *TProcSnp1,4*, *TProcSnp4,2*, *TProcSnp2,4*, *TProcSnp3,4*, *TProcSnp4,3*), as well as all the needed connections between places and transitions.

A number of new elements and connections are created during this model evolution scenario. The creation process can be automated by executing model transformation rules or calling APIs provided by the modeling environment. Figure 1b shows the SRN model after executing the transformation rules defined in the Embedded Constraint Language (ECL) (Lin et al., 2008), which scale the model from 2 event types to 4. Although the correct number of elements (i.e., 26 model elements) are created and the correct connections are made (i.e., 38 connections), all the newly created elements and connections are placed randomly in the upper-left corner of the editor and overlapped with each other, which is unreadable without arranging the layout. However, manual layout arrangement is tedious and time-consuming, especially when the model is scaled to adapt a larger number of event types (e.g., over 100 new elements will be created when scaling a SRN model from 5 event types to 10, and over 150 connections are needed to connect them).

One option that avoids manual layout arrangement is to use the auto-layout functionality provided by the modeling tool. For instance, Figure 1c shows the scaled SRN model after applying the auto-layout function embedded in the GMF editor. Compared with Figure 1b, it can be seen that the overlaps of all the newly created elements are removed; the location of each element is changed so that the distances between each two elements are more similar; and all the elements connected are grouped together. A clear and readable model is obtained by a single mouse-click. However, a readable model does not necessarily preserve the implicit semantics and a user's mental map. As shown in Figure 1c, it is challenging to determine the corresponding part in the execution snapshot for each of the existing event types, and in Figure 1b the execution snapshot is clearly separated by different event types. On the other hand, the layout of event definitions in Figure 1c is changed from the original horizontal arrangement to vertical. Although it does not significantly affect the understandability or implicit semantics of the definitions, users might have their own preferences of placing the event definition horizontally, and the auto-layout functionality obviously destroyed this particular mental map.

An alternative layout configuration approach to address the problems associated with auto-layout algorithms is to specify layout information in the model transformation rules (i.e., the <x, y> coordinates of model elements is often an inherent property that can be modified with model transformation languages). For instance, Figure 2 shows an excerpt of the ECL code that configures layout information in the model transformation process. Each model element has a *Location* attribute that can be used to configure the coordinates of the element, and setting this attribute after adding the new element can result in placing it at the given position. Although users can control fully the configuration of the layout, it forces the semantic aspect of the model transformation rules to be entangled with layout concerns, the consequence being that any changes to the transformation rules about the model elements might lead to modifications on the layout configurations. Such an approach makes a transformation rule less cohesive. Furthermore, to test and debug the layout configuration, executing the rules and adjusting the configurations will have to be iterated, which is a tedious and time-consuming process. This task becomes more challenging when many relative coordinate configurations are involved. Additionally, adding the layout configuration in the transformation rules requires users to understand the model transformation language, the layout configuration APIs, as well as the metamodel definition. In many cases, when end-users (e.g., domain experts who are not familiar with model transformation languages) encounter the layout problems, it would be challenging for them to learn the languages and metamodel definitions.

```
strategy addEvents(min_new, max_new : integer; TEnSnpGuardStr : string)
{
     if (min_new <= max_new) then
         addNewEvent(min_new, TEnSnpGuardStr);
         addEvents(min_new+1, max_new, TEnSnpGuardStr);
     endif;
 }
```

```
strategy addNewEvent(event_num : integer; TEnSnpGuardStr : string)
{
     declare start, stTran, inProg, endTran : atom;
     declare TStSnp_guard : string;

     start := findAtom("StSnpSht");
     stTran := addAtom("ImmTransition", "TStSnp" + intToString(event_num));
     stTran.setLocation(500, 600);
     TStSnp_guard := "(#S" + intToString(event_num) + " == 1)?1 : 0";
     stTran.setAttribute("Guard", TStSnp_guard);
     inProg := addAtom("Place", "SnpInProg" + intToString(event_num));
     inProg.setLocation(stTran.getLocation().getX() + 100, 600);
     endTran := addAtom("ImmTransition", "TEnSnp" + intToString(event_num));
     endTran.setLocation(100, getUpperMostAtom().getLocation().getY() + 100);
     endTran.setAttribute("Guard", TEnSnpGuardStr);

     addConnection("InpImmedArc", start, stTran);
     addConnection("OutImmedArc", stTran, inProg);
     addConnection("InpImmedArc", inProg, endTran);
     addConnection("OutImmedArc", endTran, start);
}
...
```

Figure 2. An excerpt of ECL to complete the model evolution with integrated layout configuration (adapted from (Lin et al. 2006))

The Administration of Cloud Computing Management Models

Another example is based on the Cloud Computing Management Modeling Language (C2M2L) (Sun et al., 2009a) - a domain-specific modeling language (DSML) constructed specifically to describe the deployment of application nodes in a cloud computing server that monitors the running status of each node. For instance, the top of Figure 3 shows a diagram of an EJB cloud application deployed in Amazon EC2 (http://aws.amazon.com/ec2, 2011), containing several Nodes, such as *Web Tier Instance*, *Middle Tier Instance*, *Data Tier Instance* all connected to the *Load Balancer*. *NodeServices* (not shown in the figure) are included in each Node (e.g., *Apache*, *Tomcat*, *MySQL*, *JBoss*, *OpenSSH*) to define the services needed for each tier instance. A list of properties can be configured for each *Node*, such as the name of the host (i.e., *HostName*), the running status of the Node (i.e., *IsWorking*), the load of the CPU (i.e., *CPULoad*), and the changing rate of the CPU load (i.e., *CPULoadRateOfChange*). The connections between Nodes indicate the data flows from the source *Node* to the target *Node*. This model configures the deployment and execution parameters of an application in a cloud computing server.

To facilitate the management of applications in the cloud, a causal relationship is built between the running applications and the model. Changes to the state of the cloud application must be communicated back to the modeling tool and translated into changes in the elements of the model, while changes from the model must also be pushed back into the cloud. Therefore, the models defined by C2M2L serve as an interface to deploy, monitor, and manage the applications in the cloud at runtime.

Model evolution scenario in C2M2L: One essential task in the management of applications in the cloud is to ensure that each node is handling a proper amount of work load without being overloaded. For instance, if the *CPULoad* of a certain Node is out of the normal range (e.g., *CPULoad > 100*), the *Node* will stop working, so the connection between the failed *Node* and the *NodeBalancer* needs to be removed. Furthermore, *Nodes* containing the same *NodeServices* and configuration, and the corresponding, connections need to be replicated in order to balance the work load, as shown in the bottom of Figure 3.

When managing the C2M2L models, layout configuration is also indispensible. For instance, when a failed *Node* is replaced with two new *Nodes*, the new *Nodes* should be placed under the original *Node* as illustrated in the bottom of Figure 3. Also, since the failed *Node* is no longer used, it is better to place it at the top of the editor rather than taking space among the working *Nodes*. To improve the management

process and make it more illustrative, it is also desired to highlight the failed *Node* with a red background color, and to use the green for the newly added *Nodes*.

This layout configuration process will be challenging to accomplish manually, particularly when a large number of application *Nodes* are running in the cloud. Because the existing auto-layout functionality cannot preserve such kind of layout requirements, it is necessary to have an approach to automate the layout configuration process without using model transformation languages or knowing metamodel definitions.

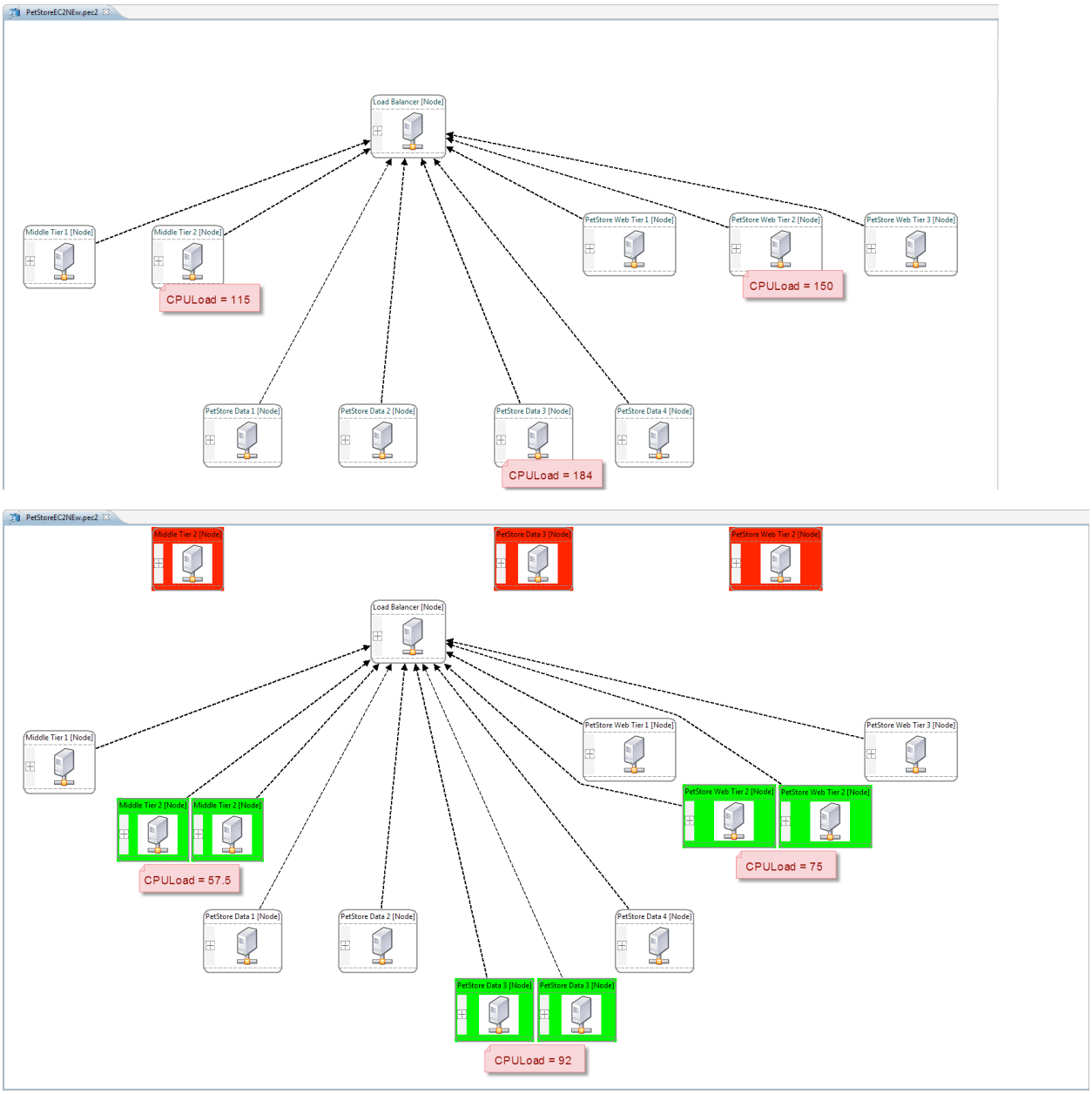


Figure 3. A C2M2L model with three *Nodes* overloaded (top) and the model after removing and replicating the overloaded *Nodes* (bottom)

RELATED WORKS

A compact and readable diagram layout is vital for leveraging the full potential of graphical DSMLs. Because models are frequently subjected to evolution and transformation during their life cycles, techniques enabling the automatic adaption of a model's diagram layout become crucial for retaining model compactness and readability.

Concerning the goal of the demonstration-based approach to layout configuration as presented in this chapter, we distinguish between two categories of related work: (i) approaches aiming to improve single diagram layouts, and (ii) approaches for optimizing sequences of diagram layouts. In this chapter, we describe a novel technique for the latter category, i.e., deriving a new version of an existing diagram layout after its underlying model has been transformed. Nevertheless, one way of deriving a new version of a diagram layout is to simply apply single diagram layout algorithms after each change of the underlying model, which is often referred to as dynamic graph drawing in the literature (Di Battista et al., 1998).

Layout Configuration for Single Diagrams

Due to the graph-based nature of diagrams, algorithms for configuring the layout of diagrams are strongly related to algorithms solving the graph drawing problem (Di Battista et al., 1998). In this research area, several different approaches have been proposed. A popular representative of such approaches is the spring embedder layout (Fruchterman & Reingold, 1991), which is a force-driven layout algorithm. In particular, each node in a graph contains attractive and repulsive forces that either move an element toward or away from other nodes. Iteratively, each node is moved until the sum of all forces settles at a minimum. Although such algorithms are capable of largely avoiding overlapping nodes and edges in a diagram, due to their generic nature, they do not take type information of graph nodes and domain-specific layout preferences into account. However, domain-specific layout patterns, such as horizontal tree layouts for control flow languages, are commonly applied in practice to support users in understanding a diagram. Therefore, domain-specific layout algorithms have been developed which place nodes and edges according to certain layout constraints or layout rules. Hower and Graf (Hower & Graf, 1996) provided an extensive survey on constraint-based layout approaches in several application domains. More recently, Dwyer et al. proposed an authoring tool specifically designed for network diagrams that also places elements according to domain-specific layout constraints (Dwyer et al., 2009). A constraint solver computes a solution for these declarative constraints to produce an improved diagram layout. Layout-based approaches employ domain-specific layout rules instead of constraints. As proposed by Maier and Minas (Maier & Minas, 2009), a rule consists of a condition and an action that is executed if the condition is fulfilled. For instance, a rule might be activated as soon as an element is too small for its compartments (condition) and induce a resizing of the element (action).

Generic, as well as domain-specific layout algorithms, also found their way into several graphical modeling tools. For instance, UML tools such as ArgoUML, Visual Paradigm , and Enterprise Architect, as well as meta-modeling tools such as GMF, GEMS, GME, and MetaCase+ provide automatic layout functionality. Most modeling tools use domain-specific layout algorithms because they are tailored for specific modeling languages. Metamodeling tools usually provide generic layout algorithms for modeling editors that are created from metamodel specifications. Additionally, many model editors also offer extension points for attaching domain-specific layout facilities.

Layout Configuration for Diagram Sequences

The aforementioned approaches focus on establishing a diagram layout from scratch or improving an existing diagram layout, but they are not tailored to optimize the layout across sequences of diagrams. However, when a model is transformed, the diagram has to be adjusted to the evolved model. One major requirement in this task is to preserve the mental map (Misue et al., 1995) across one or more

transformations of the diagram's underlying model. This aspect is the particular focus of the previous works (Jucknath-John et al., 2006; Pilgrim, 2007; Johannes & Gaul, 2009).

Jucknath-John et al. aim at layout graphs that are transformed by a sequence of endogenous graph transformations. The design rationale of their algorithm is to (i) achieve an optimal quality for each single graph layout, (ii) retain the mental map of a graph layout, (iii) consider its future extension, and (iv) identify the changes between two succeeding graph layouts by visually emphasizing the differences. To achieve these goals, the authors propose an iterative layout algorithm based on the aforementioned spring embedder layout (Fruchterman & Reingold, 1991). In particular, the spring embedder layout is extended by the concept of node aging and protection of the layout of senior nodes. By this, senior nodes (i.e., nodes that have been introduced earlier than others) are less likely to be repositioned by the algorithm than younger nodes in order to retain the mental map of the graph layout.

The focus of Pilgrim is to retain the mental map in exogenous model transformations. The transformation of the semantic model is applied using ATL (Jouault & Kurtev, 2005). The proposed algorithm takes the transformed input model, the input diagram layout, the output model, and the transformation trace as input to create a new diagram layout for the generated output model. Nodes representing elements in the output model are placed according to the position of nodes representing input model elements linked by the transformation trace in order to retain the mental map. The output diagram layout is optimized by scaling and adjusting the nodes to avoid overlaps. Furthermore, a 3D editor is used to display the source diagram and target diagram (and its correspondences in terms of traces) in a single window.

Johannes and Gaul considered diagram layout when composing domain-specific models. In their approach, the layout composition information is delivered through a graphical model composition script, which specifies how the semantic models should be composed. After the composed model is created, the diagrams of the composed model are merged into a new composed diagram according to the positions in the graphical model composition script. Johannes and Gaul also apply algorithms to adjust the final layout to remove overlaps.

All mentioned approaches for configuring the layout of diagram sequences particularly focus on retaining the mental map in endogenous or exogenous transformations. Pilgrim and Johannes and Gaul tackle this issue for exogenous transformations. Only Jucknath-John et al., as our approach, focus on endogenous transformations. In the approach of Jucknath-John et al., the position of existing nodes is protected by applying the concept of node aging, but in contrast to our approach, they do not consider transformation rule-specific layout preferences. With specific transformation rule layout preferences, the transformation engineer may regard certain aspects of a transformation in the resulting layout which enables users to understand a diagram better, especially in the context of a specific evolution task after the transformation has been performed. Only Johannes and Gaul partially consider this aspect because the resulting diagram layout is set up according to the positioning in the graphical composition script. However, they do not support the configuration of relative positioning of nodes to certain existing context nodes of a transformation. Additionally, all aforementioned approaches do not support the automatic assignment of other layout properties (e.g., background colors) after a transformation has been performed. Another major difference of our approach to these existing approaches is the adoption of the WYSIWYG technique to easily specify the desired layout after a transformation is created by demonstration.

INTRODUCTION TO MODEL TRANSFORMATION BY DEMONSTRATION

Our solution to the auto-layout customization problem is to use a demonstration-based technique to support specification of the layout configuration by automating the whole process as model transformations. The idea is based on our previous work on Model Transformation By Demonstration (MTBD), which is a new approach to implement endogenous model transformations, with the goal being to enable general users (e.g., domain experts or non-programmers) to realize transformation tasks without knowing model transformation languages or metamodel definitions. In this section, we introduce MTBD, and then explain how to use MTBD to support general model evolution tasks by using the motivating

examples presented previously. The extensions to support layout configuration will be presented in the next section.

Overview of MTBD

The basic idea of MTBD is that rather than manually writing model transformation rules, users are asked to use concrete model instances and demonstrate how to transform a source model to a target model by directly editing and changing it. During the demonstration process, a recording and inference engine captures all of the user operations and automatically infers a transformation pattern that summarizes the desired evolution task captured as a transformation. This generated pattern can be executed by the engine in any model instance to carry out the same evolution task on other parts of a model.

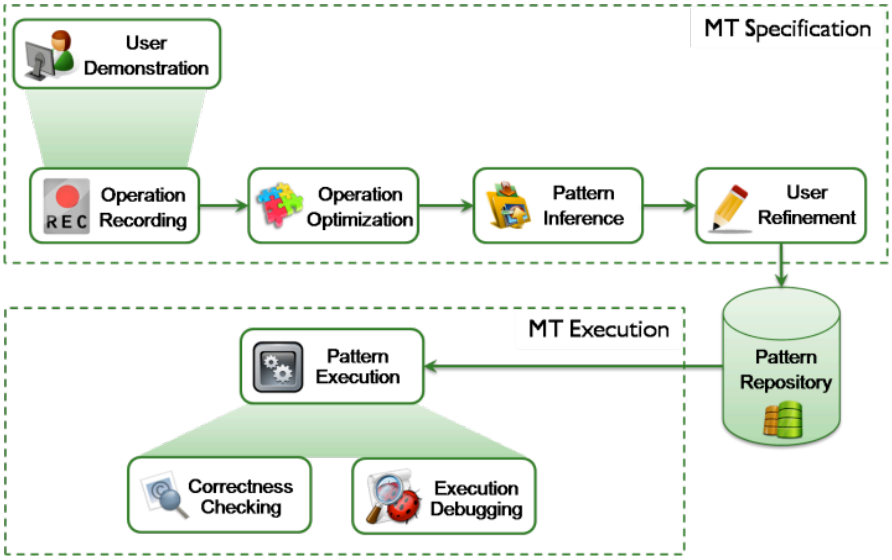


Figure 4. Overview of MTBD

Figure 4 is an overview of the MTBD idea, which consists of the following main steps and components:

Step 1 – User Demonstration and Recording. Users must first give a demonstration by directly editing a model instance (e.g., add a new model element or connection, modify the attribute of a model element, connect two model elements) to simulate an evolution task. During the demonstration, users are expected to perform operations not only on model elements and connections, but also on their attributes, so that the attribute evolution can be supported. An attribute refactoring editor has been developed to enable users to access all the attributes in the current model editor and specify the desired transformation (e.g., string and arithmetic computation). At the same time, an event listener monitors all the operations occurring in the model editor and collects the information for each operation in sequence.

Step 2 – Operation Optimization. The list of recorded operations indicates how a model evolution should be performed. However, not all operations in the list are meaningful. Users may perform useless or inefficient operations during the demonstration. For instance, without a careful design, it is possible that a user first adds a new element and modifies its attributes, and then deletes it in another operation later, with the result being that all the operations regarding this element actually did not take effect in the transformation process and therefore are meaningless. Thus, after the demonstration, the engine optimizes the recorded operations to eliminate meaningless actions.

Step 3 – Pattern Inference. With an optimized list of recorded operations, the transformation can be inferred. Because the MTBD approach does not rely on any model transformation languages, it is not necessary to generate specific transformation rules, although that is possible. Instead, we generate a transformation pattern, which summarizes the precondition of a transformation (i.e., where a

transformation should be done) and the actions needed in a transformation (i.e., how a transformation should be done).

Step 4 – User Refinement. The initial pattern inferred is specific to the demonstration and is usually not generic and accurate enough for general reuse, due to the limitation on the expressiveness of the user demonstration. Users are permitted to refine the inferred transformation by providing more feedback for the desired transformation scenario. For instance, users may give more restrictive preconditions on the desired evolution, such as "replace element *A* only if *A* has no incoming or outgoing connections," or "add new element *B* in *C* only when the attribute value of *C* is greater than *200*." Users can also identify which operations should be generic (i.e., operations should be repeated as long as appropriate model elements are available, rather than being executed only once). All the user refinements are still performed at the model instance level without explicitly modifying the metamodel, after which a transformation pattern will be finalized and stored in the pattern repository for future use.

Step 5 – Pattern Execution. The final generated patterns can be executed on any model instances. Because a pattern consists of the precondition and the transformation actions, the execution starts with matching the precondition in the new model instance and then carries out the transformation actions on the matched locations of the model. Notifications are made to users when the selected model fails to match the transformation pattern. Multiple transformation patterns can be executed in sequence on the same model, in order to apply some continuous evolution tasks. The same matching process is taken before executing each pattern.

Step 6 – Correctness Checking and Debugging. Although the location matching the precondition guarantees that all transformation actions can be executed with necessary operands, it does not ensure that executing them will not violate the syntax, semantics definitions or external constraints. Therefore, the execution of each transformation action will be logged and model instance correctness checking is performed after every execution. If a certain action violates the metamodel definition, all executed actions are undone and the whole transformation is cancelled. An execution control component has been developed as part of MTBD to control the number of execution times, and enable the execution of multiple patterns together in sequence. A debugger is under development to enable end-users to track the execution of the transformation pattern without being exposed to low-level execution information.

Using MTBD, users are only involved in editing model instances to demonstrate the model transformation process including specific attribute configurations and giving feedback after the demonstration. All of the other procedures (i.e., optimization, inference, generation, execution, and correctness checking) are fully automated. No model transformation languages are used and the generated transformation patterns are invisible to users. One demonstration results in a transformation pattern. Therefore, users are completely isolated from knowing MTLs and the metamodel definition.

Admittedly, a user demonstration is not as expressive and powerful as specified rules, the result being that MTBD can only support a subset of tasks that can be realized using MTLs (e.g., QVT and ATL). For example, selecting the element with a certain maximum value can be specified directly with a function call, but it is challenging to demonstrate by mouse and keyboard. However, MTBD has been applied successfully in a number of model evolution tasks, such as model refactoring, aspect-oriented modeling, and model scalability. It has been shown in many cases to be a practical alternative to enable end-users to realize model evolution tasks without the knowledge of MTLs and metamodels.

Applying MTBD to the SRNML

To better illustrate the idea, we show how to use MTBD to demonstrate scaling of SRN models (the motivating example from an earler section) without configuring the layout. The demonstration illustrates how the transformation pattern can be generated and reused to automate the scaling process in other model instances.

By examining the scalability requirements of the example, the task of adding one more event type to an existing SRN model consists of the following three steps, as shown in Figure 5:

Step 1. Create a new set of *Places*, *Transitions* and connections for the new event type. Specify proper names for them based on the name of the event.

Step 2. Create the *TStSnp* and *TEnSnp Snapshot Transitions* and the *SnpInProg Snapshot Place*, as well as the associated connections.

Step 3. For each pair of <existing *Snapshot Place*, new *Snapshot Place*>, create two *TProcSnp Transitions* and connect their *SnpInProg Places* to these *TProcSnp Transitions*.

To give this demonstration, we choose the 2-event SRN model as shown in Figure 1a. Then, we manually edit the model and demonstrate the task following the three steps. For Step 1, the operations shown in List 1 are performed.

List 1. Operations for Step 1 of example 2.1 in the demonstration

Sequence	Operation Performed
1	Add a *Place* in *SRNRoot*
2	Create an artificial name with the value: *EventName = "3"*
3	Set *SRNRoot.Place.name = "A" + EventName = "A3"*
4	Add a *Transition* in *SRNRoot*
5	Set *SRNRoot.Transition.name = "B" + EventName = "B3"*
6	Add a *Place* in *SRNRoot*
7	Set *SRNRoot.Place.name = "Sn" + EventName = "Sn3"*
8	Add a *Transition* in *SRNRoot*
9	Set *SRNRoot.Transition.name = "S" + EventName = "S3"*
10	Add a *Place* in *SRNRoot*
11	Set *SRNRoot.Place.name = "Sr" + EventName = "Sr3"*
12	Connect *SRNRoot.A3* and *SRNRoot.B3*
13	Connect *SRNRoot.B3* and *SRNRoot.A3*
14	Connect *SRNRoot.B3* and *SRNRoot.Sn3*
15	Connect *SRNRoot.Sn3* and *SRNRoot.S3*
16	Connect *SRNRoot.S3* and *SRNRoot.Sr3*
17	Connect *SRNRoot.A3* and *SRNRoot.B3*

All of these operations are used to create the new elements and necessary connections for the event definition (i.e., *A3*, *B3*, *Sn3*, *S3*, *Sr3*). Each event has a unique event name, and the names of all the newly created elements are based on this event name (e.g., the new event is called "3," so the places and transitions are named as "A3," "B3," "Sn3,"). Therefore, Operation 2 is used to manually create a name for a certain value, which can be reused later in the rest of the demonstration to setup the desired name for each element. For instance, when setting up the attribute in operations 3, 5, 7, 9, 11, users just need to give the specific composition of the attributes by using the artificial names and constants, or simply select an existing attribute value in the attribute refactoring editor. After applying these operations, the model will have a new event type definition, as shown in Figure 5 (Step 1).

To finish Step 2, the necessary *Snapshot Places* and *Snapshot Transitions* are added for the new event type by performing the operations indicated in List 2. Figure 5 (Step 2) shows the model after these operations.

List 2. Operations for Step 2 of example 2.1 in the demonstration

Sequence	Operation Performed
18	Add a *SnpPlace* in *SRNRoot*
19	Set*SRNRoot.SnpPlace.name = "SnpLnProg"+EventName = "SnpLnProg3"*
20	Add a *SnpTransition* in *SRNRoot*
21	Set *SRNRoot.SnpTransition.name = "TStSnp" + EventName = "TStSnp3"*
22	Add a *SnpTransition* in *SRNRoot*
23	Set *SRNRoot.SnpTransition.name = "TEnSnp" + EventName = "TEnSnp3"*
24	Connect *SRNRoot.StSnpSht* and *SRNRoot.TStSnp3*
25	Connect *SRNRoot.TStSnp3* and *SRNRoot.SnpLnProg3*
26	Connect *SRNRoot.SnpLnProg3* and *SRNRoot.TEnSnp3*
27	Connect *SRNRoot.TEnSnp3* and *SRNRoot.StSnpSht*

To demonstrate Step 3, two *Snapshot Transitions* for each <existing *Snapshot Place*, new *Snapshot Place*> need to be created. For example, *TProcSnp1,3* and *TProcSnp3,1* should be added between *SnpLnProg1* and *SnpLnProg3*, while *TProcSnp2,3* and *TProcSnp3,2* are needed between *SnpLnProg2* and *SnpLnProg3*. Because the number of the existing *Snapshot Place* varies in different model instances, instead of demonstrating the addition of *Snapshot Transitions* in every pair, we only need to demonstrate the process for one pair, followed by identifying the operations as generic in the user refinement step. This is needed so that the engine will generate the correct transformation pattern to repeat these operations when needed according to the different number of the existing *Snapshot Place*. The operations performed are shown in List 3. We select *SnpLnProg2* as the existing *Snapshot Place*, and demonstrate the creation of *Snapshot Transitions - TProcSnp2,3* and *TProcSnp3,2*.

List 3. Operations for Step 3 in the demonstration of example 2.1
(* represents generic operations to be identified)

Sequence	Operation Performed
28*	Add a *SnpTransition* in *SRNRoot*
29*	Set *SRNRoot.SnpTransition.name = "TProcSnp" + SRNRoot.SnpLnProg2.name.subString(9) + "," + EventName = "TProcSnp" + "2" + "," + "3" = "TProcSnp2,3"*
30*	Add a *SnpTransition* in *SRNRoot*
31*	Set *SRNRoot.SnpTransition.name = "TProcSnp" + EventName + "," + SRNRoot.SnpLnProg3.name.subString(9) = "TProcSnp" + "3" + "," + "2" = "TProcSnp3,2"*
32*	Connect *SRNRoot.SnpLnProg2* and *SRNRoot.TProcSnp2,3*
33*	Connect *SRNRoot.TProcSnp2,3* and *SRNRoot.SnpLnProg3*
34*	Connect *SRNRoot.SnpLnProg3* and *SRNRoot.TProcSnp3,2*
35*	Connect *SRNRoot.TProcSnp3,2* and *SRNRoot.SnpLnProg2*

When specifying the name attributes, complex String composition can be given using the Java APIs, as done in operations 29 and 31. After the demonstration is completed and generic operations are identified in the user refinement step, the inference engine automatically infers and generates the transformation pattern, which will be saved in the transformation repository.

After the pattern is saved, a user may select any model instance and a desired transformation pattern, and the selected model will be scaled by adding a new event type. The execution controller can be used to enable execution of a pattern multiple times. Figure 1b is the result of adding two event types using the inferred pattern automatically. Although the correct number of new elements and connections has been created with consistent names, all of the elements are overlapped and randomly placed on the upper-left corner of the editor. To address the layout problem in model transformation, we extended MTBD to enable demonstration of layout configuration.

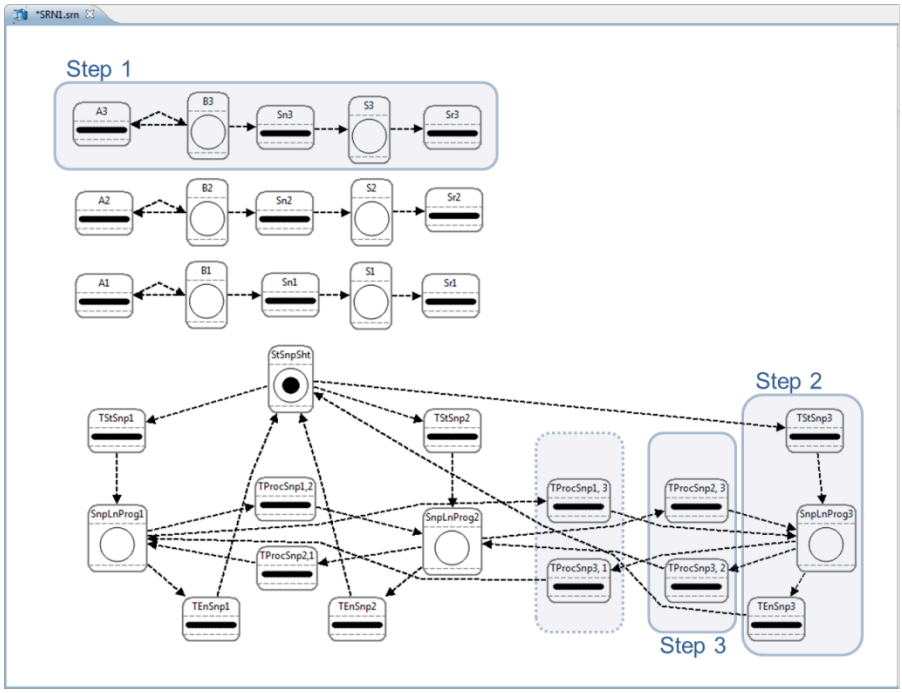


Figure 5. The process of scaling a SRN model from two events to three events

LAYOUT SUPPORT IN MTBD

The idea of supporting layout configuration using MTBD is based on an additional demonstration step. After demonstrating the basic model transformation task, users are able to demonstrate how to configure the layout (e.g., where to place each element, what layout properties to specify), so that the layout information can be summarized and integrated in the generated transformation pattern. Because the demonstration is performed on the concrete model instances in the model editor and the engine automatically records the low-level layout information (e.g., the specific coordinate values), users can configure the layout in a WYSIWYG manner without being aware of the implementation details.

Currently, we focus on configuring the layout from two perspectives: the location of the model elements and the appearance of model elements. Model connections are not considered in the current work, because in most modeling tools and editors, the layout of connections depends on the source and target model elements and cannot be customized by users, but the idea proposed in this chapter can be applied to connections as well.

Configuring Locations of Model Elements

The main layout of a model depends on the location of each model element. In most modeling tools, a *Location* attribute is attached to each model element internally, which specifies the coordinates of the element. In the editors, the *Location* attribute of an element changes automatically when it is moved. Therefore, by capturing moving operations (i.e., the drag-and-drop operation in most editors) in the

demonstration, coordinate values can be recorded automatically by reading the updated location of the element. As model elements often need to be moved in the editor multiple times before reaching the desired location, rather than recording every moving operation, a confirmation location operation is provided for users to confirm the final desired location of a model element, which is recorded and integrated into the generated transformation pattern. The confirmation location operation can be based on either absolute coordinates or relative coordinates.

Absolute coordinates. The most direct and simplest layout configuration is to use absolute coordinates. Users can demonstrate where to place each element exactly in the editor. As shown in List 4, two kinds of operations are added to the editor to support locating and choosing the absolute coordinates of a certain element. When the transformation is executed, the chosen model elements will be placed in the exact same location as in the demonstration.

List 4. Layout configuration operations using absolute coordinates

Operation Type	**Description**
Set *X* as Current	Set *X* in the current coordinates as the desired *X*
Set *Y* as Current	Set *Y* in the current coordinates as the desired *Y*

For example, in the top of Figure 6, the *Node* in the lower-right corner is selected and confirmed with an absolute coordinate for both *X* and *Y* in the demonstration. When the generated transformation pattern is executed, the *Node* is configured with the same coordinate values automatically as shown in the bottom of Figure 6.

While confirming the absolute coordinates, the actual coordinate values are not visible to users, so that users are separated from the low-level layout information. The recording engine reads the values, and saves them in the final generated transformation pattern. In the execution process, the execution engine loads the values and passes them as parameters to the location configuration process.

The absolute coordinates approach is easy to implement, but not flexible and practical in most model transformation scenarios. Unless the user is configuring the layout information for a single and unique model element (e.g., the root or folder of a domain model), using absolute coordinates cannot adapt the transformation to diverse model evolution scenarios. For example, if any model elements or connections exist or cross at certain absolute coordinates configured in the demonstration, placing a new element there will lead to overlaps. In addition, when applying a transformation pattern multiple times, all the newly created elements will be placed in the same location. Therefore, in many cases, configuring the layout using relative coordinates is more preferable.

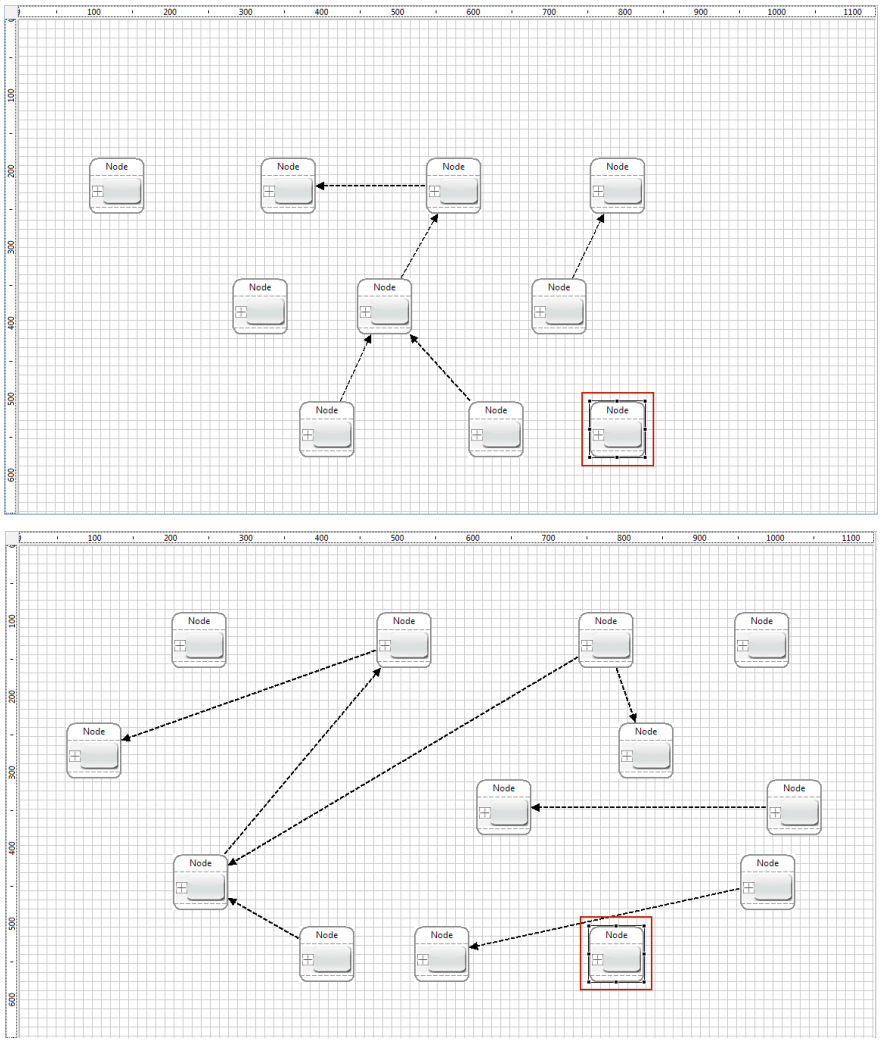


Figure 6. Using absolute coordinates in the demonstration (top) make the element be in the same location in every model evolution scenario (bottom)

Relative coordinates to model boundary. Using relative coordinates needs a reference point. One type of reference is to consider all the model elements and connections as a whole rectangle (i.e., the minimum rectangle that includes all the current model elements and connections), and use the boundary of the rectangle as the reference. The coordinates can be relative to each side of the rectangle from either inside or outside. Thus, a total of eight operations can be extended, as shown in List 5.

List 5. Layout configuration operations using relative coordinates to model boundary

Operation Type	Description
Set *Y* Relative to Uppermost (Inside/Outside)	Set the desired *Y* to be the current *Y* relative to the uppermost boundary of the current model from inside or outside
Set *Y* Relative to Lowermost (Inside/Outside)	Set the desired *Y* to be the current *Y* relative to the lowermost boundary of the current model from inside or outside
Set *X* Relative to Leftmost (Inside/Outside)	Set the desired *X* to be the current *X* relative to the leftmost boundary of the current model from inside or outside
Set *X* Relative to Rightmost (Inside/Outside)	Set the desired *X* to be the current *X* relative to the rightmost boundary of the current model from inside or outside

Similar to using absolute coordinates, users may demonstrate the relative values by a drag-and-drop process in the editor without being aware of the low-level details. It is the recording engine that automatically captures the rectangle boundary in the current editor and calculates the specific relative values. During the execution process of the generated transformation, the execution engine will capture the boundary of the model again and set up the location attribute using the stored relative values. For instance, in the top of Figure 7, *Node1* and *Node2* are two newly created model elements. When configuring the layout in the demonstration, *Node1* is specified using *Set X Relative to Rightmost Outside*, and *Set Y as Current*, while *Node2* applies *Set X Relative to Leftmost Inside* and *Set Y Relative to Lowermost Inside*. The result is that when applying the transformation in other models, *Node1* will always be placed to the right of the existing model, but at the same vertical level as in the demonstration; and *Node2* will always appear on the left-lower corner of the existing model, as shown in the bottom of Figure 7.

The relative coordinate to the model boundary proves to be useful in practice when a large number of new elements are created in the model evolution process or the same process is executed multiple times (e.g., the first motivating example). As the model is enlarged, it is always necessary to add new elements based on a layout pattern incrementally. However, when the model transformation focuses on modifying a small number of elements without adding many new elements (e.g., the second motivating example), relative coordinates to the boundary are not sufficient, and a different type of reference with improved granularity is needed.

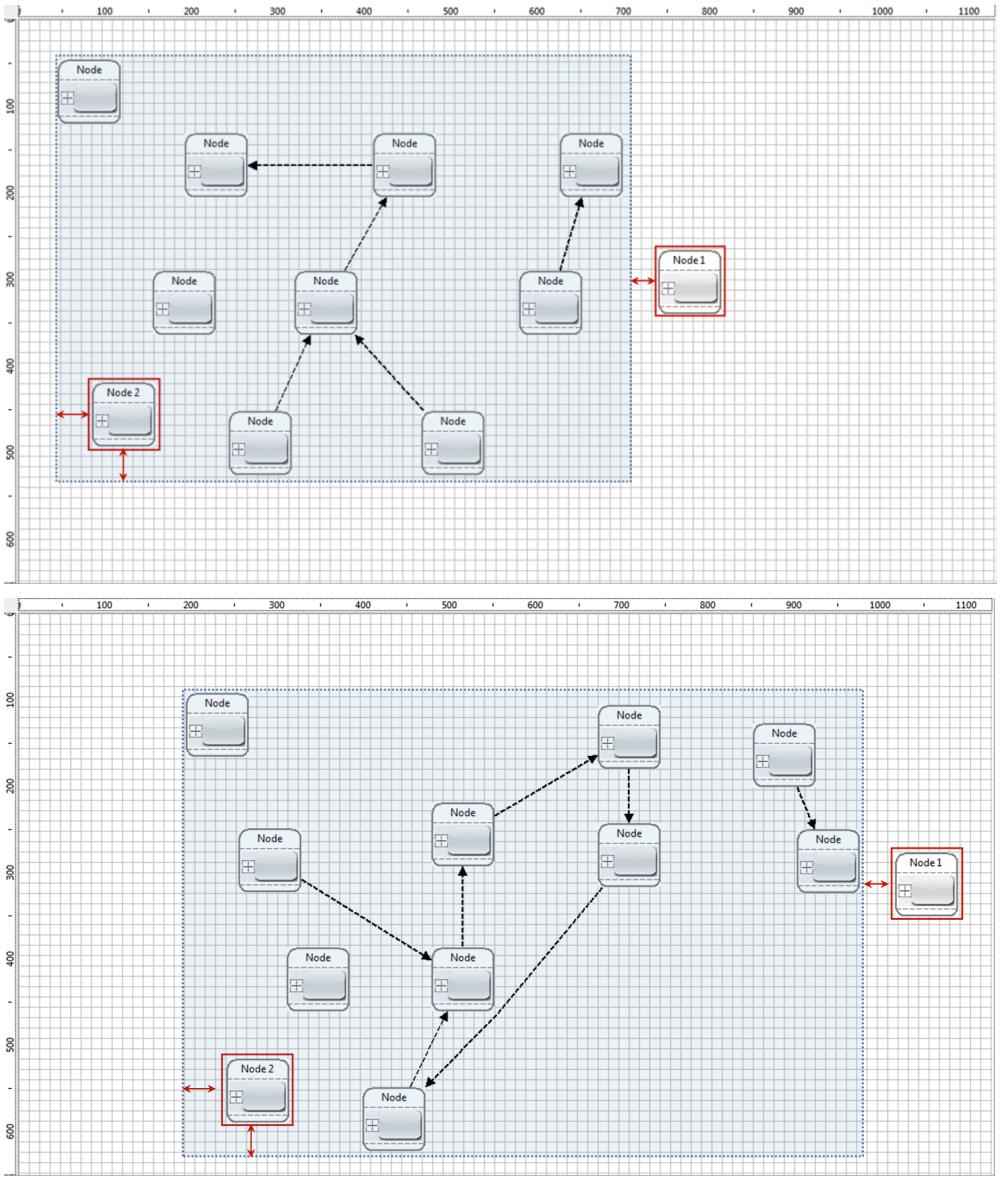


Figure 7. Using coordinates relative to the boundary of the existing model in the demonstration (top) make the element be in the location relative to the existing model in every model evolution scenario (bottom)

Relative coordinates to model element(s). A more improved granularity and flexible reference is to set up the coordinates of a model element relative to other model element(s). As enumerated in List 6, users can configure *X/Y* based on the location of another model element.

List 6. Layout configuration operations using relative coordinates to model element(s)

Operation Type	**Description**
Set *X* Relative to Model Element *E*	Set the desired *X* to be the current *X* relative to the *X* of the model element *E*
Set *Y* Relative to Model Element *E*	Set the desired *Y* to be the current *Y* relative to the *X* of the model element *E*

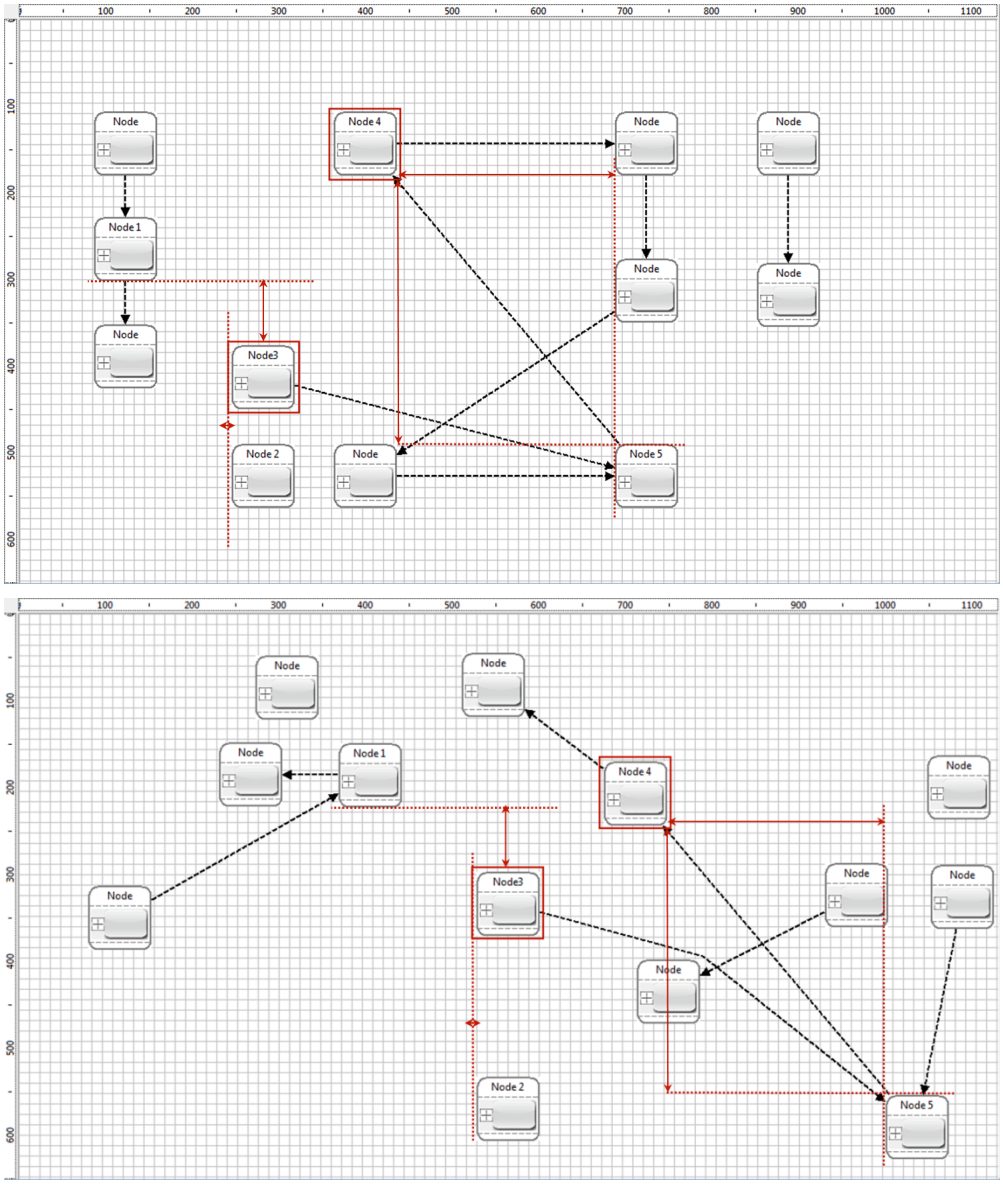


Figure 8. Using coordinate relative to the other model elements in the demonstration (top) make the element be in the location relative to the same model elements in every model evolution scenario (bottom)

In the current implementation, a model element selector has been developed that enables users to choose any element from the existing model instance, and set up the *X* or *Y* coordinate. Again, the recording engine calculates the relative value and stores it, while the execution engine loads the value and sets up the location. The calculated relative value can be either positive or negative according to the relative

locations (i.e., the value will be negative if the element is to the left or above the reference element, and will be positive if the element is to the right or below the reference element).

For example, at the top of Figure 8, several model elements (i.e., *Node1, Node2, Node3, Node4, Node5*) are involved in a model transformation scenario. Users configure the location of *Node3* using *Set X Relative to Model Element Node2*, and *Set Y Relative to Model Element Node1*, so that *Node3* will always be in the same horizontal level as *Node2* and have the same vertical distance to *Node1* no matter where *Node2* and *Node1* are located in different model instances. On the other hand, both *X* and *Y* of *Node4* are configured relative to *Node5*, the result being that *Node4* is always on the upper-left part of *Node5* with the same distance as illustrated in the bottom of Figure 8.

Relative coordinates to specific model elements provide more freedom for users to configure model element locations. When the reference element(s) were used in the demonstration of the semantic aspect of the evolution process, the recording and execution engine can save and load the values directly; but if the reference element(s) are not included in the previous demonstration about semantic aspect evolution (e.g., using the *Node* in the upper-left corner as a reference in the top of Figure 7), the element(s) will be automatically added to the generated transformation as a structural precondition, which means that the execution engine must match and find out the element in the model transformation to ensure the correct layout configuration using it.

Configuring the appearance of model elements

Apart from the location of model elements, the appearance (e.g., the color, shape, font, size used in the model element) is also essential to the layout of the model or even the semantics of the model. Compared with the location configuration, the appearance is easier to handle. The recording engine captures all the operation events regarding changing the appearance, and integrates them in the generated transformation pattern, so that the execution of the pattern will replay these operation events and configure the same appearance specified in the demonstration.

EXAMPLE LAYOUT DEMONSTRATION

In this section, we demonstrate the use of MTBD to automate the layout configuration for the two motivating examples presented earlier in the chapter.

Configure Layout for SRN Model Evolution

After demonstrating the model transformation as shown in the previous section, the model evolution at the semantics level has been accomplished. At this point, users can continue to drag-and-drop each element in the editor and confirm the desired location using the provided layout configuration operations.

Figure 9 shows the desired layout configuration for each element in the model transformation process. According to the three steps in this model evolution scenario, the newly created model elements and connections belong to three parts. The first part is the event definition (i.e., *A3, B3, Sn3, S3, Sr3*). Assume that most users prefer to place these elements always above the previous definitions. Therefore, they use the uppermost boundary of the existing model as the reference for *Y*, and the *X* coordinate of each element in event type *1* for *X*. Users would generally perform the operations in List 7 in the layout demonstration.

For the new execution snapshot part definition (i.e., *TStSnp3, SnpLnProg3, TEnSnp3*), we set all the *X* to be relative to the rightmost boundary, and *Y* relative to the root of the execution snapshot *StSnpSht* (*TStSnp3.Y* is set to be directly relative to *StSnpSht.Y*, *SnpLnProg3.Y* is set to be relative to *TStSnp3.Y*, and *TEnSnp3.Y* to be relative to *SnpLnProg3.Y*), please see List 8 for the specific details. Finally, for the *Execution Snapshot Transitions*, the *X* is relative to the rightmost boundary, and *Y* is relative to the *Snapshot Place* it is connected to, please see List 9 for those details.

List 7. Operations to configure layout demonstration for part one of the motivating example

(The layout demonstration is immediately after the model transformation demonstration)

Sequence	Operation Performed
36	Set *SRNRoot.A3.Y* Relative to Uppermost Outside
37	Set *SRNRoot.A3.X* Relative to *A1.X*
38	Set *SRNRoot.B3.Y* Relative to Uppermost Outside
39	Set *SRNRoot.B3.X* Relative to *B1.X*
40	Set *SRNRoot.Sn3.Y* Relative to Uppermost Outside
41	Set *SRNRoot.Sn3.X* Relative to *Sn1.X*
42	Set *SRNRoot.S3.Y* Relative to Uppermost Outside
43	Set *SRNRoot.S3.X* Relative to *S1.X*
44	Set *SRNRoot.Sr3.Y* Relative to Uppermost Outside
45	Set *SRNRoot.Sr3.X* Relative to *Sr1.X*

List 8. Operations to configure layout demonstration for part two of the motivating example

Sequence	Operation Performed
46	Set *SRNRoot.TStSnp3.X* Relative to Rightmost Outside
47	Set *SRNRoot.TStSnp3.Y* Relative to *SrnRoot.StSnpSht.Y*
48	Set *SRNRoot.SnpLnProg3.X* Relative to Rightmost Outside
49	Set *SRNRoot.SnpLnProg3.Y* Relative to *TStSnp3.Y*
50	Set *SRNRoot.TEnSnp3.X* Relative to Rightmost Outside
51	Set *SRNRoot.TEnSnp3.Y* Relative to *SnpLnProg3.Y*

List 9. Operations to configure layout demonstration for part three of the motivating example

Sequence	Operation Performed
52	Set *SRNRoot.TProcSnp2,3.X* Relative to Rightmost Outside
53	Set *SRNRoot.TProcSnp2,3.Y* Relative to *SrnRoot.StSnpSht.Y*
54	Set *SRNRoot.TProcSnp3,2.X* Relative to Rightmost Outside
55	Set *SRNRoot.TProcSnp3,2.Y* Relative to *TStSnp3.Y*

After the demonstration is completed, the recording engine calculates all the values and integrates them in the final generated transformation pattern. Executing the final pattern will result in the model shown in Figure 1d.

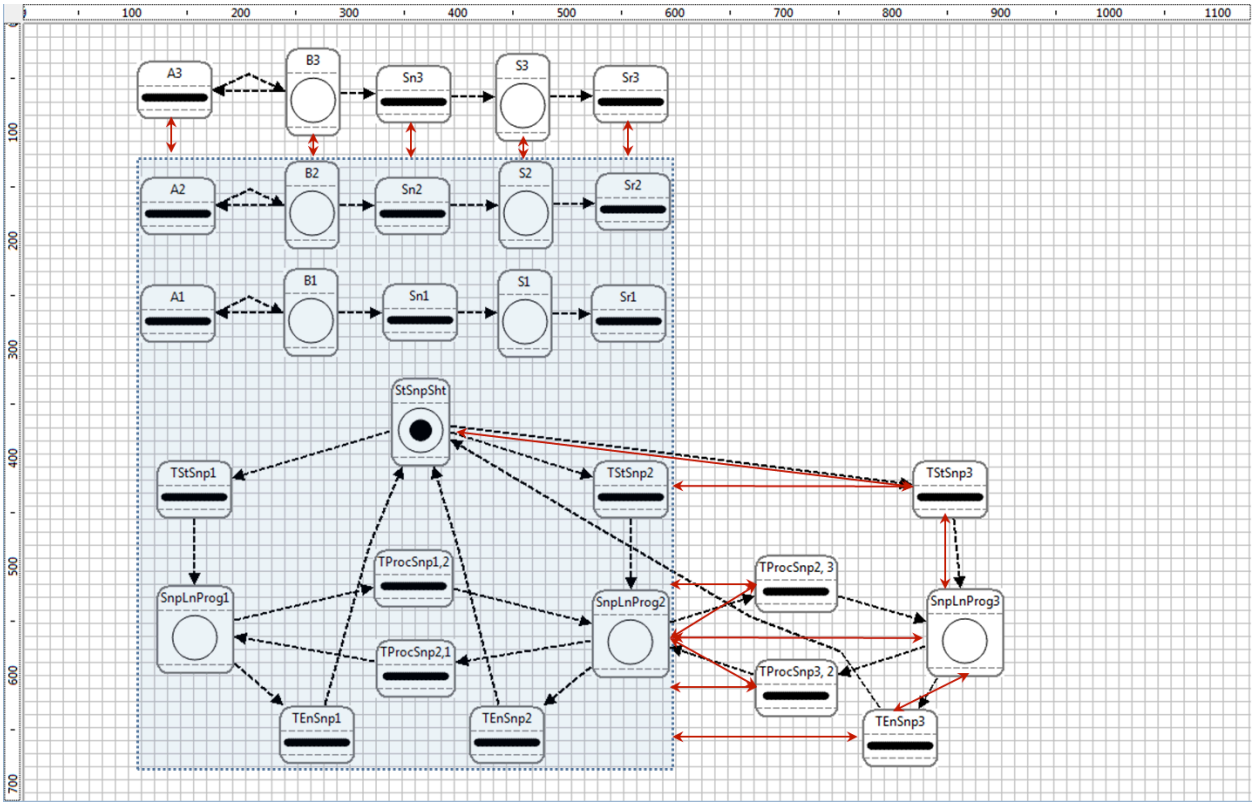


Figure 9. The layout demonstration in action for the first motivating example

Configuring the Layout for C2M2L Model Evolution

Before addressing the layout problem in the second motivating example, we first demonstrate how to evolve the model at the semantics level by replicating the overloaded *Node*. The demonstration in this scenario is straightforward, which is based on an overloaded *Node* (e.g., the overloaded *Node* is defined as the *Node* having *CPULoad* > *100*). Replicating a model element requires the creation of a new copy of the same type of element, as well as setting up all the attributes of the element to be the same as the one being replicated. Therefore, operations 1-8 and operations 9-14 in List 10 replicate two *Nodes* and copy all the attributes from the overloaded *Node* - *MiddleTier2*, except the *CPULoad* is balanced by dividing the original value. Operations 15-17 deal with the connections. The precondition is given after the demonstration in the precondition specification dialog, where users can choose any model element or connections involved in the demonstration and specify the precondition constraints.

Layout demonstration comes after the traditional use of MTBD, which is based on demonstrating the operational parts of an evolution task. The desired layout as shown in Figure 10 uses the original location of the overloaded *Node* as a reference to set up the location for the two new *Nodes*. The overloaded *Node* is moved to the uppermost part of the editor (using its own *X* and relative *Y* to the *NodeBalancer*). Also, two different colors are configured for the new and old *Nodes*. Please see List 11 for details.

List 10. Operations in the demonstration for the second motivating example

Sequence	Operation Performed
1	Add a *Node* in *C2M2LRoot (Replicate the 1st Node)*
2	Set *Node.Name = MiddleTier2.Name = "MiddleTier2"*
3	Set *Node.AMI = MiddleTier2.AMI = "ami-45e7002c"*
4	Set *Node.Annotation = MiddleTier2.Annotation = "Middle Tier for PetStore"*
5	Set *Node.HeartbeatURI = MiddleTier2.HeartbeatURI = "http://ps01.aws.amazon.com/hb"*
6	Set *Node.HostName = MiddleTier2.HostName = "http://ps01.aws.amazon.com/hb"*
7	Set *Node.CPULoad = MiddleTier2.CPULoad / 2 = 115 / 2 = 57.5* ^{p1}*MiddleTier2.CPULoad > 100*
8	Add a *Node* in *C2M2LRoot (Replicate the 2nd Node)*
9	Set *Node.Name = MiddleTier2.Name = "MiddleTier2"*
10	Set *Node.AMI = MiddleTier2.AMI = "ami-45e7002c"*
11	Set *Node.Annotation = MiddleTier2.Annotation = "MiddleTier for PetStore"*
12	Set *Node.HeartbeatURI = MiddleTier2.HeartbeatURI = "http://ps01.aws.amazon.com/hb"*
13	Set *Node.HostName = MiddleTier2.HostName = "http://ps01.aws.amazon.com/hb"*
14	Set *Node.CPULoad = MiddleTier2.CPULoad / 2 = 115 / 2 = 57.5*
15	Remove the connection between *MiddleTier2* and *LoadBalancer*
16	Connect *MiddleTier2* (1st) and *LoadBalancer*
17	Connect *MiddleTier2* (2nd) and *LoadBalancer*

List 11. Operations to configure layout in the demonstration of the second motivating example

Sequence	Operation Performed
18	Set *MiddleTier2(1st).X* Relative to *MiddleTier2(overloaded).X*
19	Set *MiddleTier2(1st).Y* Relative to *MiddleTier2(overloaded).Y*
20	Set *MiddleTier2(2nd).X* Relative to *MiddleTier2(overloaded).X*
21	Set *MiddleTier2(2nd).Y* Relative to *MiddleTier2(overloaded).Y*
22	Set *MiddleTier2(overloaded).Y* Relative to *NodeBalancer.Y*
23	Set *MiddlerTier2(1st)* background color to *Green*
24	Set *MiddlerTier2(2nd)* background color to *Green*
25	Set *MiddlerTier2(overloaded)* background color to *Red*

When the final generated pattern is applied to other model instances, all the overloaded *Nodes* can be detected automatically based on the precondition, and the required new *Nodes* can be created to replicate the old ones, with the location configured and colors highlighted.

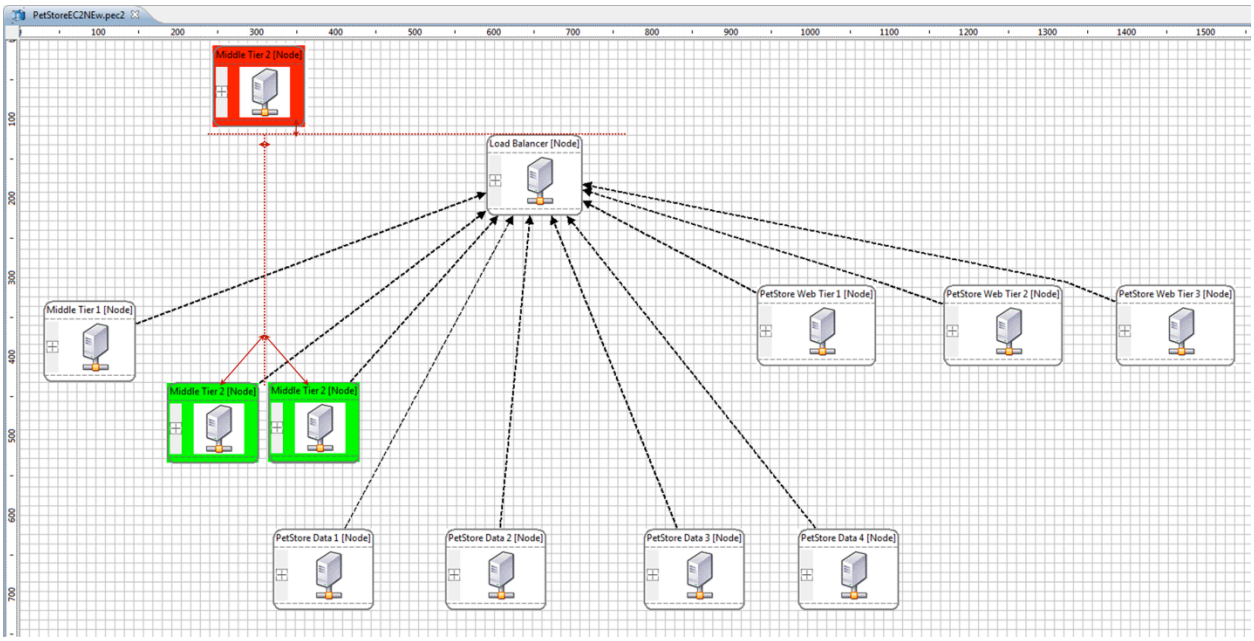


Figure 10. Demonstrating the layout configuration for the C2M2L model

CONCLUSION AND FUTURE WORK

This chapter presents a new approach for configuring model layout during model evolution, which is based on the demonstration-based technique – MTBD. The demonstration is performed on concrete model instances, whereby users move model elements and confirm locations or appearance to customize the desired layout. The ability to demonstrate the desired layout can reflect the implicit semantics and the user's own mental map, without the need to be aware of the low-level details associated with model transformation and metamodeling. Because the demonstration is performed in a WYSIWYG manner, the layout configuration is more precise. We have found that this enables easier testing and debugging of the layout concerns associated with a model evolution task. Moreover, the layout configuration is performed after demonstrating the model transformation of the core evolution task, which clearly separates the core evolution task from the model layout concerns, rather than being entangled together. Furthermore, no model transformation languages are used in the process, and users do not need to understand metamodel definitions, which enables general end-users and non-programmers to configure their desired layout in the model evolution process.

As future work, dealing with the overlaps of model elements in model evolution is our next goal. Although the relative coordinate configuration helps to avoid the overlaps, it cannot adapt to every scenario perfectly, particularly when model elements are used as references. Integrating overlap removal algorithms might solve the problem, but there is also a possibility that the implicit semantics and a user's mental map will be affected after applying the algorithm. Additionally, we plan to also implement the configuration of model layout for exogenous model transformations using MTBD (i.e., the model transformation between two different domains) so that the layout of a target model is set up based on the source model's layout.

ACKNOWLEDGEMENT

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

REFERENCES

Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/ (last accessed June, 2011).

Battista, G., Eades, P., & Tamassia, R. (1993). Algorithms for Automatic Graph Drawing: An Annotated Bibliopgraphy. *Technical report, Department of Computer Science, Brown University*.

Bottoni, P., Guerra, E., & de Lara, J. (2006). Metamodel-based Definition of Interaction with Visual Environments. *In Proceedings of the 2nd International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI '06)*, CEUR Workshop Proceedings 214.

Di Battista, G., Eades, P., Tamassia, R., & Tollis, I. G. (1998). *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall.

Dwyer, T., Marriott, K., & Wybrow, M. (2009). Dunnart: A Constraint-Based Network Diagram Authoring Tool. *In 16th International Symposium on Graph Drawing (GD '08)*, Springer-Verlag LNCS 5417, Hersonissos, Heraklion Crete, Greece, pp. 384-389.

France, R., Ghosh, S., Song, E., & Kim, D. (2003). A Metamodeling Approach to Pattern-Based Model Refactoring. *IEEE Software*, vol. 20, no. 5, pp. 52-58.

Fruchterman, T., & Reingold, E. (1991) Graph Drawing by Force-Directed Placement. *Software Practice and Experience*, vol. 21, pp. 1129-1164.

Generic Eclipse Modeling System (GEMS). http://www.eclipse.org/gmt/gems/. (last accessed June, 2011).

Graphical Modeling Framework (GMF). http://www.eclipse.org/modeling/gmf/. (last accessed June, 2011).

Gray, J., Lin, Y., & Zhang, J. (2006). Automating Change Evolution in Model-Driven Engineering. *IEEE Computer*, vol. 39, no. 2, pp. 51-58 (2006).

Gray, J., Tolvanen, J.P, Kelly, S., Gokhale, A., Neema, S., & Sprinkle, J. (2007). Domain-Specific Modeling. *Handbook of Dynamic System Modeling*, CRC Press.

Greenfield, J., & Short, K. (2004) *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, John Wiley and Sons.

Hower, W., & Graf, W.H. (1996). A Bibliographical Survey of Constraint-based Approaches to CAD, Graphics, Layout, Visualization, and Related Topics. *Knowledge-Based Systems*, 9(7), Elsevier, 449-464.

Johannes, J., & Gaul, K. (2009). Towards a Generic Layout Composition Framework for Domain-specific Models. *In Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, Orlando, FL, 6 pages.

Jouault, F., & Kurtev, I. (2005). Transforming Models with ATL. *In Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, Springer-Verlag LNCS 3844, pp. 128-138.

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: A Model Transformation Tool. *Science of Computer Programming*, vol. 72, no.1-2, pp. 31-39.

Jucknath-John, S., Graf, D., & Taentzer, G. (2006). Evolutionary Layout of Graph Transformation Sequences. *In Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs)*, Electronic Communications of the EASST , vol. 1.

Kogekar, A., Kaul, D., Gokhale, A., Vandal, P., Praphamontripong, U., Gokhale, S., Zhang, J., Lin, Y., & Gray, J. (2006). Model-driven Generative Techniques for Scalable Performabality Analysis of Distributed Systems. *IPDPS Workshop on Next Generation Systems*, Rhodes Island, Greece, pp. 292-292.

Langer, P., Wimmer, M., & Kappel, G. (2010). Model-to-Model Transformations By Demonstration. *In Proceedings of International Conference on Model Transformation*, Malaga, Spain, pp. 153-167.

Lin, Y., Gray, J., Zhang, J., Nordstrom, S., Gokhale, A., Neema, S., & Gokhale, S. (2008). Model Replication: Transformations to Address Model Scalability. *Software: Practice and Experience*, vol. 38, no. 14, pp. 1475-1497.

Lédeczi, A., Bakay, A., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). Composing Domain-specific Design Environments. *IEEE Computer*, vol. 34, no. 11, pp. 44-51.

Maier, S., & Minas, M.. (2009). Rule-based Diagram Layout using Meta Models. *In Proceedings of the Workshop on Visual Languages and Computing 2009 (VLC 2009)*, San Francisco, USA.

MetaCase+ (MetaCase+). http://www.metacase.com/. (last accessed June, 2011).

Misue, K., Eades, P., Lai, W., & Sugiyama, K. (1995). Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, vol. 6, no. 2, pp. 183-210.

MOF Query/Views/Transformations Specification (QVT). http://www.omg.org/cgi-bin/doc?ptc/2005-11-01 (last accessed June, 2011).

Muppala, J., Ciardo, G., & Trivedi, K. (1994). Stochastic Reward Nets for Reliability Prediction. *Communications in Reliability, Maintainability and Serviceability*, vol. 1, no. 2, pp. 9-20.

Pilgrim, J. (2007). Mental Map and Model Driven Development. *In Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED)*, Electronic Communications of the EASST, vol. 7.

Schmidt, D., Stal, M., Rohnert, H., & Buschman, F. (2000). *Pattern-Oriented Software Architecture – Volume 2: Patterns for Concurrent and Networked Objects*, John Wiley and Sons.

Sprinkle, J. (2003). *Metamodel Driven Model Migration*. PhD thesis, Vanderbilt University, Nashville, TN.

Sun, Y, Gray, J., Langer, P., Wimmer, M., & White, J. (2010). A WYSIWYG Approach for Configuring Model Layout using Model Transformations. *10th Workshop on Domain-Specific Modeling*, held at SPLASH 2010, Reno, NV, pp. 20-25.

Sun, Y., Gray, J., & White, J. (2010). MT-Scribe: A Flexible Tool to Support Model Evolution. *Workshop on Flexible Modeling Tools (FlexiTools)*, held at SPLASH 2010, Reno, NV.

Sun, Y., White, J., & Gray, J. (2009b). Model Transformation by Demonstration. *In Proceedings of International Conference on Model Driven Engineering Languages and Systems*, Denver, CO, pp. 712-726.

Sun, Y., White, J., Gray, J., & Gokhale, A. (2009a). Model-Driven Automated Error Recovery in Cloud Computing. *Model-driven Analysis and Software Development: Architectures and Functions*, IGI Global, Hershey, PA, USA.