Automating the Maintenance of Non-functional System Properties using Demonstration-based Model Transformation

Yu Sun, Jeff Gray, Romain Delamare, Benoit Baudry, Jules White

HAL Id: hal-00916287

https://hal.inria.fr/hal-00916287

Submitted on 10 Dec 2013

Automating the Maintenance of Non-functional System Properties using Demonstration-based Model Transformation

[1]Yu Sun, [2]Jeff Gray, [2]Romain Delamare, [3]Benoit Baudry, [4]Jules White

[1]Department of Computer and Information Sciences, University of Alabama at Birmingham
`yusun@cis.uab.edu`

[2]Department of Computer Science, University of Alabama
`{gray, rdelamare}@cs.ua.edu`

[3]IRISA / INRIA Rennes
`benoit.baudryg@irisa.fr`

[4]Department of Electrical and Computer Engineering, Virginia Tech
`julesw@vt.edu`

Abstract

Domain-Specific Modeling Languages (DSMLs) are playing an increasingly significant role in software development. By raising the level of abstraction using notations that are representative of a specific domain, DSMLs allow the core essence of a problem to be separated from irrelevant accidental complexities that are typically found at the implementation level in source code. In addition to modeling the functional aspects of a system, a number of non-functional properties (e.g., quality of service constraints, timing requirements) also need to be integrated into models in order to reach a complete specification of a system. This is particularly true for domains that have distributed real-time and embedded needs. Given a base model with functional components, maintaining the non-functional properties that crosscut the base model has become an essential modeling task when using DSMLs.

The task of maintaining non-functional properties in DSMLs is traditionally supported by manual model editing or using model transformation languages. However, these approaches are challenging to use for those unfamiliar with the specific details of a modeling transformation language and the underlying metamodel of the domain, which presents a steep learning curve for many users. This paper presents a demonstration-based approach to automate the maintenance of non-functional properties in DSMLs. Instead of writing model transformation rules explicitly, users demonstrate how to apply the non-functional properties by directly editing the concrete model instances and simulating a single case of the maintenance process. By recording a user's operations, an inference engine analyzes the user's intention and generates generic model transformation patterns automatically, which can be refined by users and then reused to automate the same evolution and maintenance task in other models. Using this approach, users are able to automate the maintenance tasks without learning a complex model transformation language. In addition, because the demonstration is performed on model instances, users are isolated from the underlying abstract metamodel definitions. Our demonstration-based approach has been applied to several scenarios, such as auto-scaling and model layout. The specific contribution in this paper is the application of the demonstration-based approach to capture crosscutting concerns representative of aspects at the modeling level. Several examples are presented across multiple modeling languages to demonstrate the benefits of our approach.

1. Introduction

Domain-Specific Modeling (DSM) [17] has become an important part of Model-Driven Engineering (MDE) [27] to address the complexity of software systems. A Domain-Specific Modeling Language (DSML) can be used to declaratively define a software system with specific domain concepts. Various software artifacts (e.g., source code, simulation scripts, XML deployment description files) can be generated automatically from the models. By raising the level of abstraction, DSM helps to protect key intellectual assets from technology obsolescence, resulting in less effort and fewer low-level details to specify a given system, which can lead toward improvements for supporting end-user development [9].

Although much effort has been focused on using DSMLs to specify the functional properties of a system, the correct and complete specification of non-functional properties is also critical in many domains such as distributed systems, embedded systems, or large-scale applications. The non-functional properties range from performance, timing, and Quality of Service (QoS) constraints, to some behavior requirements such as logging and concurrency control. Although several modeling tools (e.g., GME [22], GMF [14], GEMS [13]) have been developed to allow users to define various types of DSMLs and instantiate model instances, the tools to support the maintenance of non-functional properties in a DSML are not well-developed. In most situations, manual maintenance has to be done through the basic operations provided by the editing environment, which is tedious, error-prone, and time-consuming, particularly when the base model is large and complex. A manual process for maintaining model changes often involves repeated and complicated changes and computations [15].

An alternative for automating the maintenance of non-functional properties with DSMLs is to use executable Model Transformation Languages (MTLs) [28], which can define a set of transformation rules that specify how to add, remove, or modify the non-functional properties in a base model. Although MTLs are very powerful and expressive for handling a wide range of non-functional properties in diverse DSMLs, using a transformation language is not always the perfect solution. Firstly, even though most MTLs are high-level declarative languages, they have a steep learning curve due to the complexity of their syntax, semantics, and other special features (e.g., OCL [24] specification is used in many MTLs). Moreover, model transformation rules are often defined at the metamodel level, rather than in the context of a concrete model instance, which exposes users to abstract metamodel concepts. Because many potential model users (e.g., requirements engineers, domain experts) are not necessarily software engineers or programmers, learning transformation languages and understanding the formal syntax definitions may be beyond their capability. Such challenges may prevent some users from realizing non-functional property maintenance tasks for which they have extensive domain experience. Consequently, this leads to an irony that a technology (i.e., DSM) meant to enable end-users to participate in the specification of functional components of a software system does not enable end-users to manage the non-functional properties easily.

To overcome the problem and provide an end-user approach to support automating the maintenance of non-functional properties in DSMLs, we have been investigating and extending the idea of Model Transformation By Demonstration (MTBD) [29], which applies a demonstration-based approach to enable users to specify the desired maintenance process of non-functional properties and execute the process to any model instance through an execution engine. The goal of the approach is to assist general model users (e.g., domain experts and non-programmers) in realizing the automated maintenance of non-functional properties in DSMLs without knowing a specific model transformation language or the metamodel definitions.

This paper provides the following contributions to the study of non-functional system property maintenance in DSMLs:

- We describe how MTBD is extended and adapted to support the demonstration of non-functional system property maintenance tasks in a DSML (including a user refinement mechanism and an updated transformation engine to enable users to accurately specify *where* to apply the properties and *how* to apply them) without using MTLs or exposing metamodel definitions to end-users.
- We present how MTBD can generalize and infer a transformation pattern that can be used to automate the whole maintenance process through an execution engine, as well as how users can refine the pattern.
- We select two typical non-functional system property maintenance tasks (i.e., adding a logging mechanism and applying QoS strategies) in DSMLs that have already been solved using MTLs, and present the new solutions using MTBD.
- We present an evaluation by comparing efforts (based on operational effort to perform the same tasks) between the two approaches to solve the exact same tasks. This evaluation identifies the

key advantages of using MTBD over using MTLs in the maintenance of non-functional system property tasks.

The remainder of the paper is organized as follows: two non-functional property maintenance scenarios in two different DSMLs are presented in Section 2 as motivating examples; Section 3 explains the MTBD concept through two motivating examples. In Section 4, a comparison of MTDB and the use of traditional model transformation languages is given in the context of the two motivating examples. Related works are discussed in Section 5, and Section 6 offers concluding remarks and summarizes the directions of future work.

2.   Motivating Examples of Non-Functional Properties in Models

This section presents two examples that motivate the need for automating the maintenance of non-functional system properties in different DSMLs. For each of the examples in Sections 2.1 and 2.2, background information about the specific application domain and the context of the DSML will be given, followed by an illustration using a concrete model instance. Then, we present a typical non-functional system property maintenance scenario in the domain, and describe the desired model instance after the maintenance of these properties. The challenges of accomplishing each task will be summarized afterwards. We have used both of these examples in previous efforts focused on MTLs [15][16]. In this paper, we compare our new approach that is focused on end-user demonstration to the more traditional approach that uses MTLs. The solution of using MTBD to address the exact same problems will be given in Section 3, so that a better comparison can be made between the traditional MTL approach and the new MTBD approach.

2.1 Maintaining Non-Functional System Properties in ESML Models

The Embedded Systems Modeling Language (ESML) [20] is a DSML used to graphically model real-time mission computing embedded avionics systems. ESML is specifically designed for modeling component-based avionics application deployment and distribution middleware infrastructure, which allows users to model the system from several different aspects such as *Interfaces*, *Events*, *Components*, *Interactions*, and *Configurations*.

Figure 1 shows an excerpt of an ESML model specified for one usage scenario. The top of Figure 1 illustrates the interaction among components in a mission-computing avionics application. The set of components collaborate to process a video stream that provides a pilot with real-time navigational data. The bottom of Figure 1 shows the internal representation of two components (*Display_Device* and *AirFrame_SynchProxy*), revealing the data elements and other constructs used to describe the infrastructure of component deployment and the distribution middleware. An event-driven model is implemented for the infrastructure that enables different components to update and transfer data to each other through event notification and callback methods.

Maintaining Non-Functional Properties in ESML. The need to maintain non-functional properties in ESML models arises from the requirement to apply non-functional properties to each *Data* in the components. For instance, different logging mechanisms (e.g., *LogOnRead* and *LogOnWrite*) can be added to each data element to realize the recording policy of a black-box flight data record; specific constraints are occasionally required to attach the data element as preconditions to assert a set of valid values when a client invokes the component at runtime; in some cases, concurrency elements (e.g., *Internal Locking* and *External Locking*) specifying the synchronization strategy should be distributed across the components.

As the first example used in this paper, we choose the task of adding a logging mechanism for each data element, which is defined as follows:

Example 2.1: In each implementation Component (i.e., the name of the component ends with "impl"), if an Action exists in the Component, a LogOnRead logging element should be attached to every Data element in the Component.

The tasks of adding data constraints and synchronization policies share a similar process as Example 2.1, and therefore can be solved using the same approach. A correct process of maintaining these non-functional properties involves navigating the model and locating the correct places to insert the non-functional properties, and then adding the correct non-functional elements and specifying the corresponding attributes. Although the maintenance process can be done by manual editing, it is tedious, time-consuming, and error-prone, particularly when the base model is large and complex (e.g., the partial system model shown in Figure 1 is only a subset of a system with thousands of components). Thus, an automated process can benefit the maintenance process when evolving model properties.
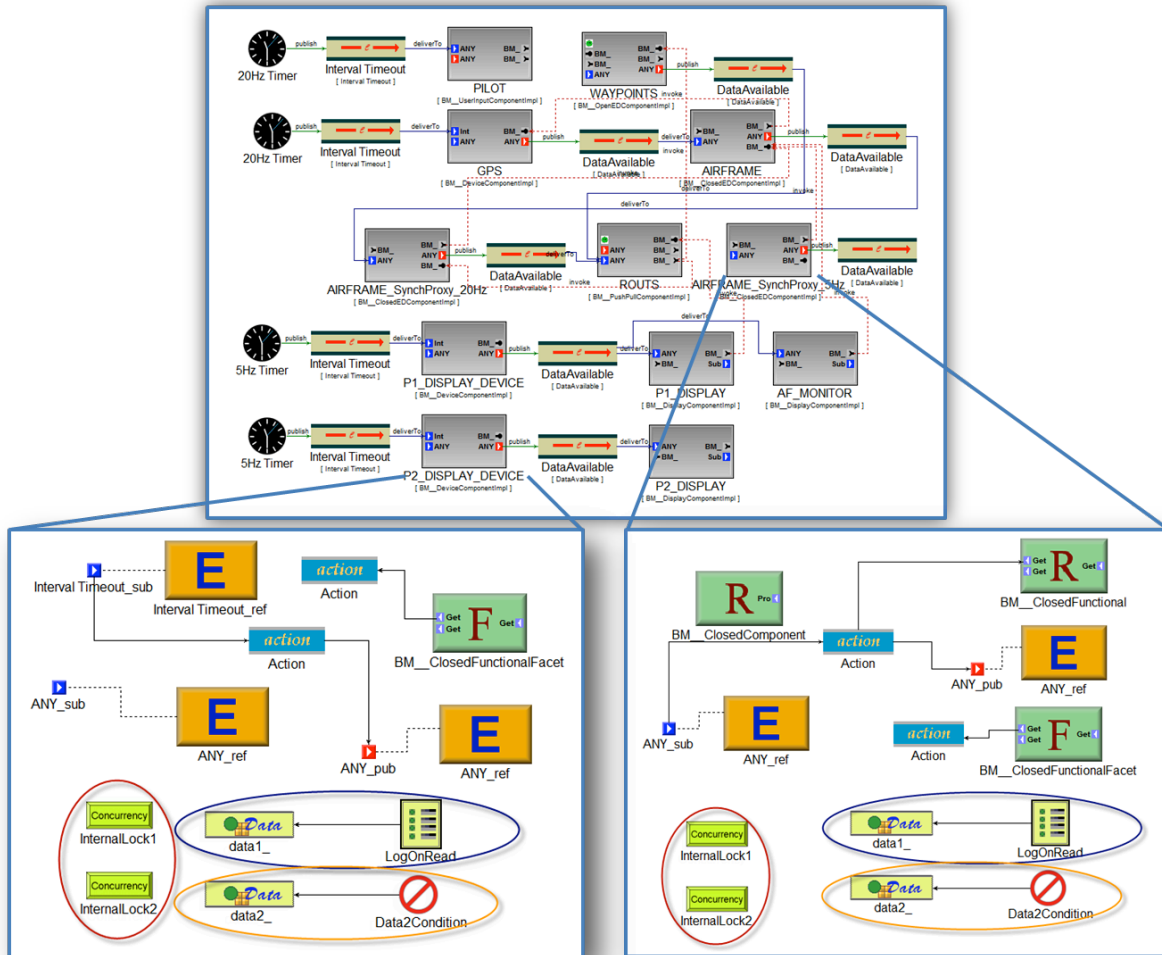

Figure 1. Adding non-functional properties in ESML models (adapted from [15])

2.2 Maintaining Non-Functional System Properties in QoSAML Models

The maintenance and evolution of distributed real-time and embedded systems is often challenging because of the consideration of different Quality-of-Service (QoS) constraints that might conflict with each other and must be treated as trade-offs among a series of alternative design decisions [16]. The QoS Adaption Modeling Language (QoSAML) [16] was designed to address the challenges of using an MDE approach, which uses a finite state machine representation extended with hierarchy and concurrency mechanisms to model the QoS adaptive behavior of the system.

One successful usage of QoSAML is to specify the QoS properties for an Unmanned Aerial Vehicle (UAV) [19]. A UAV is an aircraft that is capable of surveying dangerous terrain and territories. The UAV continuously sends video streams to a central distributor, so that operators can observe the video and give further commands to the UAV. In order to reach a precise and timely response from operations, a smooth video stream must be guaranteed, which means that the video must not be stale, or be affected by jittering. However, due to the changing conditions in the surveillance environment, the fidelity of the video stream must be maintained by adjusting the QoS parameters. For example, as shown in Figure 2, in good conditions where a reliable network transmission is available, a smooth video stream can be kept using a high video *Size* and a high video *FrameRate*, while in a poor environment, both video *Size* and *FrameRate* should be reduced in order to keep the same video transmission latency.
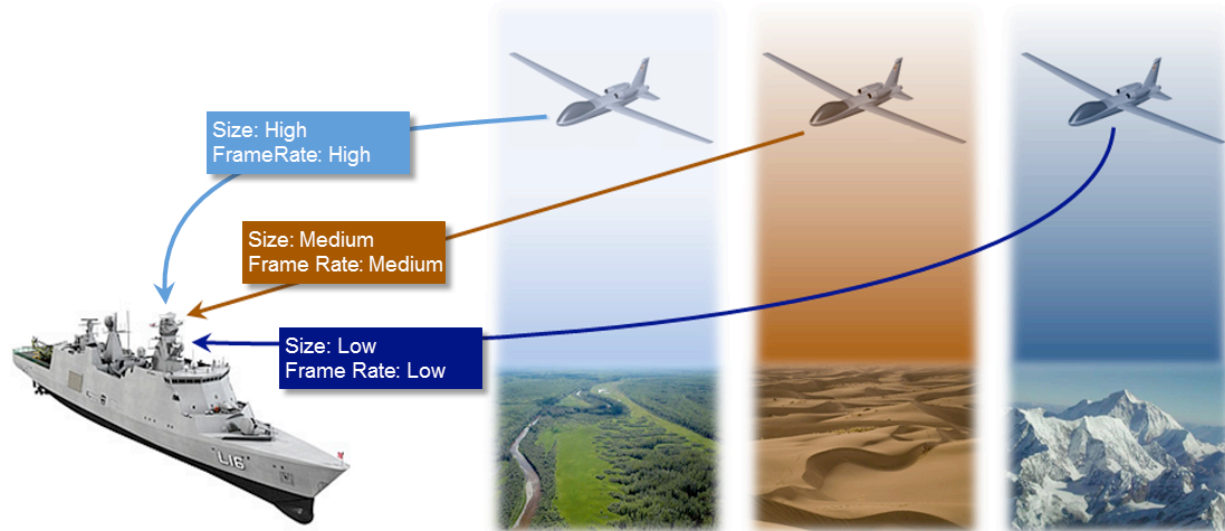

Figure 2. QoS Issue in the UAV Image Transmission Scenario

Figure 3 is part of a QoSAML model that specifies the QoS properties for a UAV application. In this model, the latency is a dependent variable input to a hierarchical state machine called *Outer State*. Inside the *OuterState*, there are state machines that describe the adaption of identified independent control parameters, such as *Size State*, *FrameRate State*. In each of the state machines, several *States* are included to represent the options for the corresponding control parameter. A state specifies three *Data* for the option: *Pri* defines the priority of this option; *Max* defines the maximum value that can be used for this parameter; and *Min* defines the minimum value for the parameter. Model translators have been developed in previous efforts that generate run-time conditions (expressed in the Contract Description Language [19]) from the QoSAML models automatically, which can be integrated into the runtime kernel of the system [16].

Maintaining Non-Functional Properties in QoSAML. The model in Figure 3 is not complete, because the transitions between different states have not been specified. A transition connects a source state to a target state, representing how a control parameter can be changed and adapted. To give the transitions, there are two different strategies that can be used, as illustrated in Figure 4. The left side of the figure specifies a protocol that exhausts the effect of one independent parameter (*Size*) before attempting to adjust another independent parameter (*FrameRate*). In other words, the *FrameRate* parameter has a higher priority so that it is not reduced or increased until there is no further reduction or increment possible to the *Size*. By contrast, the strategy in the right side of the figure is more equitable, with a zig-zag pattern suggesting that the reduction or the increment of one parameter is staggered with the reduction or the increment of another [16]. From this scenario, it can be seen that weaving the strategy protocols becomes a challenging task when more control parameters are involved, or a large number of intermediate states are included in the state.
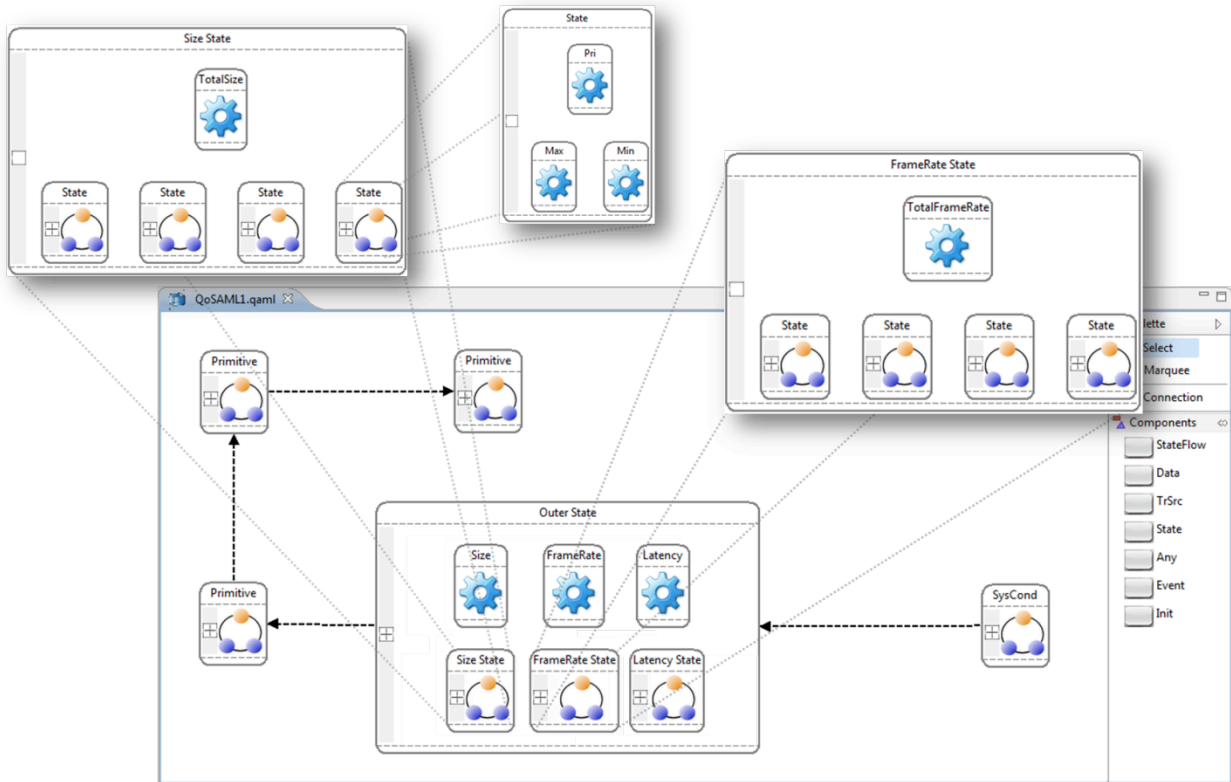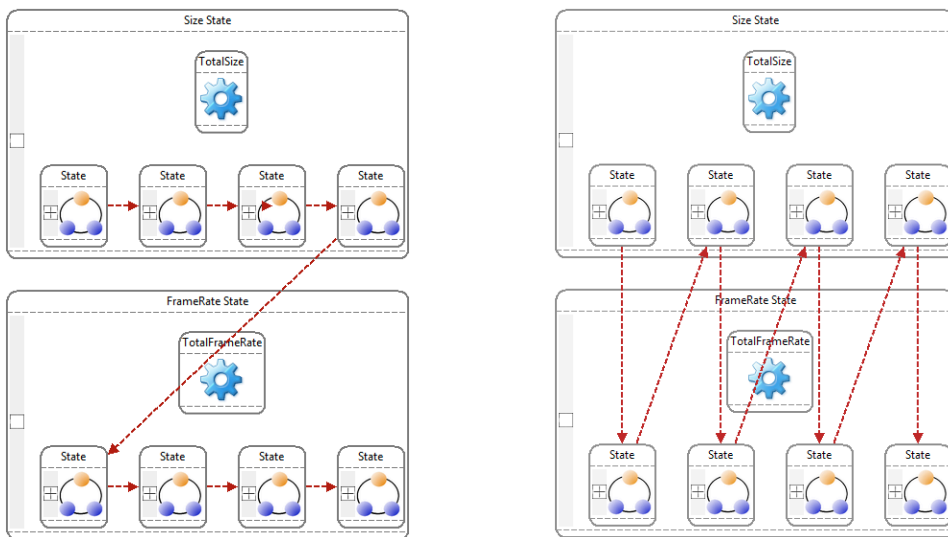
Figure 3.QoSAML Model

Figure 4. Two state transition protocols to adapt to environment - Priority Exhaustive (left) and Zig-zag (right)

As the second example for this paper, we choose the task of weaving the priority exhaustive protocol to the QoSAML model, which is defined as follows:

Example 2.2: In a given state machine, for each pair of two states included in the state machine, if their priority data are less than 5, and if the priority data of one state is one less than the other, add a transition between the two states from the state with the lower priority (SourceState) to the one with the higher priority (TargetState). In addition, set up the attributes for the transition: the Guard of the transition should be given from the users input, and the Action of the transition should be in the format of "ControlParameterName = (SourceState.Max + TargetState.Max) / 2)."

For example, Figure 5 shows the model after applying the priority exhaustive protocol to *Size State* and *FrameRate State*. The challenges of this task result from locating all pairs of the states that consist of the qualified priority data (i.e., *SourceState.Priority = TargetState.Priority – 1*, *SourceState.Priority < 5*, *TargetState.Priority < 5*), performing the repeated computation to obtain the average value from the two Max data, as well as enabling user input.
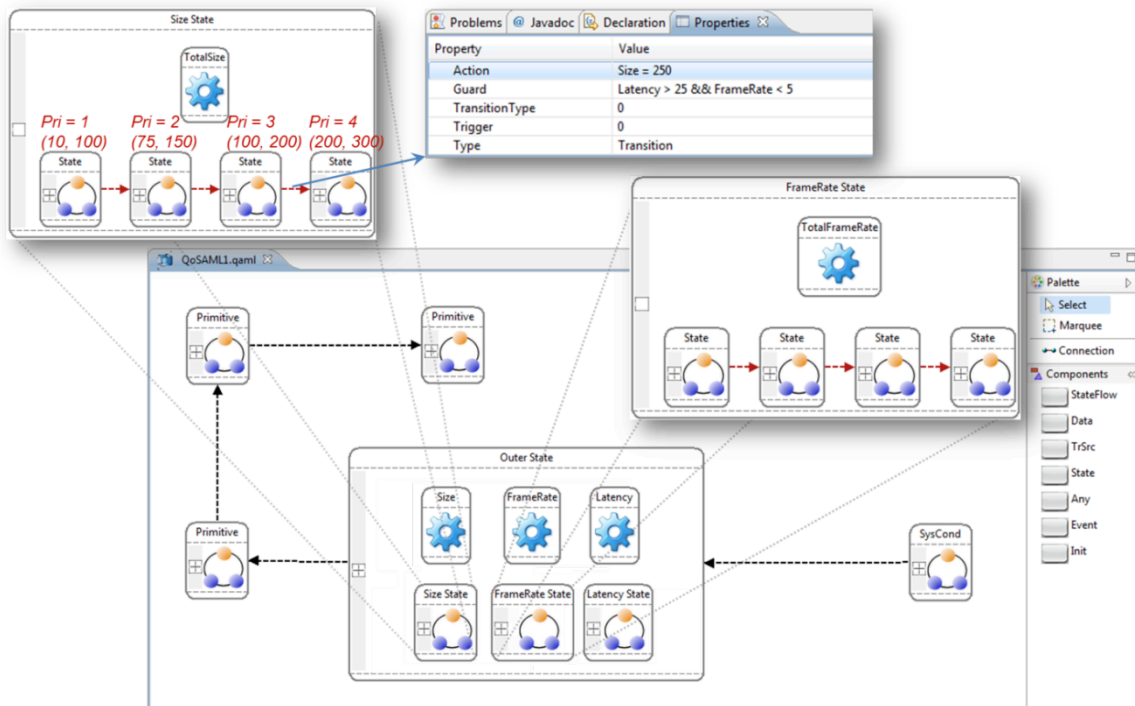

Figure 5. A QoSAML model after applying the Priority Exhaustive protocol

3. Demonstration-based Non-Functional Property Maintenance using MTBD

Model Transformation By Demonstration (MTBD) is our solution that enables general end-users to evolve and maintain non-functional properties of models, without knowing programming languages or metamodel definitions. MTBD is a new model transformation approach that was initially designed to handle model refactoring problems [29]. The idea of MTBD is to offer an alternative to writing model transformation rules manually, such that users are asked to demonstrate how the model transformation should be performed by directly editing (e.g., add, delete, connect, update) the model instance to simulate the model transformation process step-by-step. The user evolves and maintains a source model to the target model during the demonstration process. A recording and inference engine captures all user operations and infers a user's intention in a model transformation task. A transformation pattern is generated from the inference, which can be reused and executed in any model instance at any time to carry out the same model transformation process.

We have extended the MTBD concept to non-functional property maintenance tasks, so that it can be used to demonstrate the specific maintenance process and generate a reusable pattern to automatically realize the process to the rest of the model, as well as other model instances.
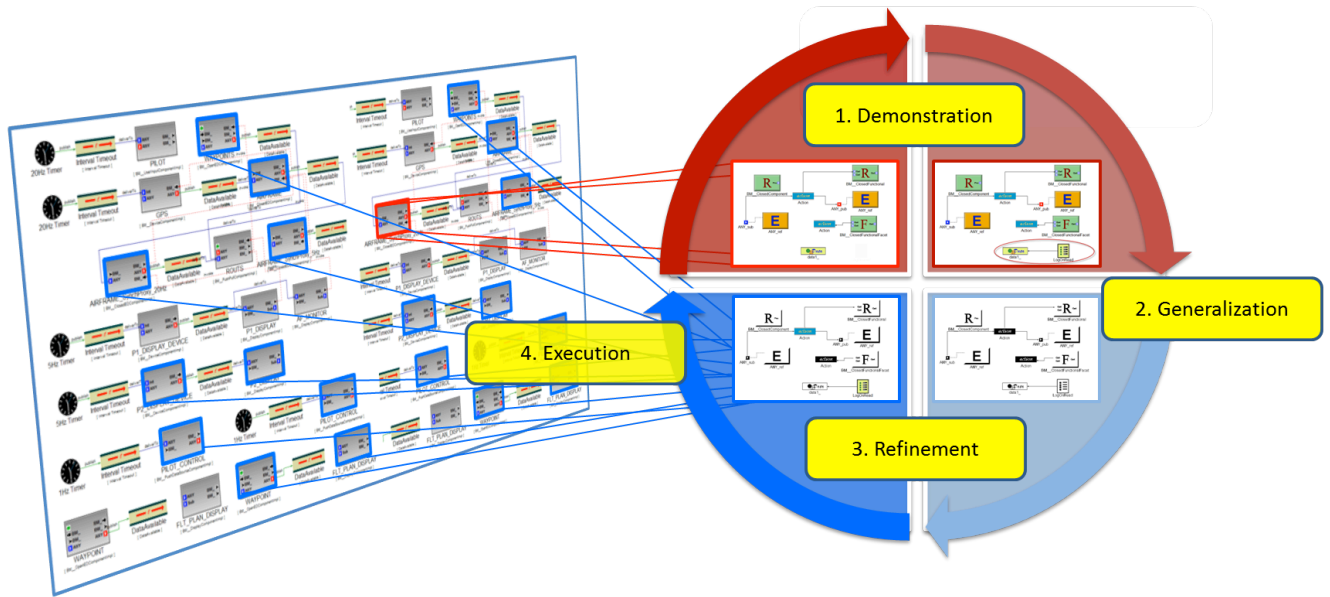
Figure 6. High-level overview of using MTBD to address non-functional property maintenance

Figure 6 is an overview of the approach. In any non-functional property maintenance task, two fundamental specifications are required to precisely define a task. First, we need to specify the "where" (i.e., the locations in the models where the non-functional property or behavior emerges). The desired locations can be specified through constraints on its structure (i.e., the presence of required model elements and connections) and attributes (i.e., the model elements and connections with qualified attribute values). Second, we need to specify the "what" (i.e., the specific composition of the non-functional properties or behavior). As shown in the two examples, a non-functional property can contain various types of actions ranging from adding new elements or connections (removing and modifying elements and connections also happens in some tasks), conducting computations, asking for user input and initializing attributes. Using the MTBD approach, users can simply select one of the qualified locations, and then perform the necessary non-functional property maintenance tasks on the specific location. At the same time, the recording engine captures all the information about the demonstration, and later generalizes an initial pattern from the user's demonstrated behavior. Because the initial pattern may not reflect the precise specification about the locations, refinements are enabled in the extended MTBD to allow a user to provide more detailed feedback in order to restrict the pattern in terms of the structure and attributes. The pattern is finalized after the refinement, and can be executed and applied to the whole model instance to realize the complete weaving process.

In Section 3.1, we explain the approach with more implementation details about each step, along with solving the motivating Example 2.1. Section 3.2 continues to illustrate the approach by presenting the solution to motivating Example 2.2.

3.1 MTBD in Action

The main steps of the extended MTBD approach for non-functional property maintenance are shown in Figure 7. To better describe the idea, we use Example 2.1 to explain the specific processes.
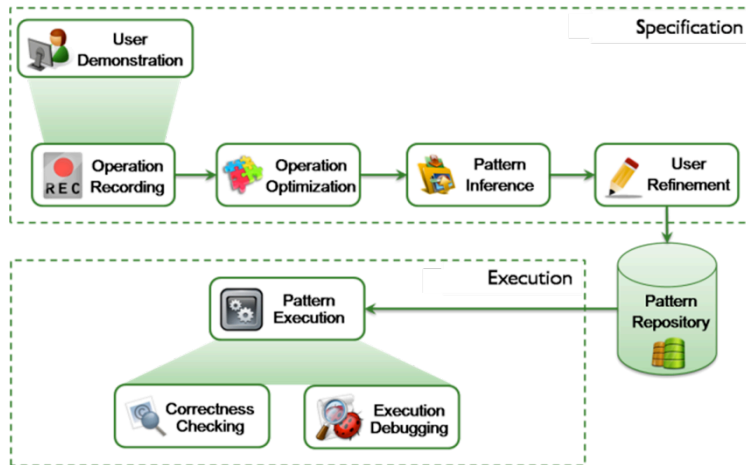
Figure 7. The implementation of MTBD (initially described in [29])

Step 1 – User Demonstration and Recording. Users first give a demonstration by locating one of the correct places in the model where non-functional properties are needed, and 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 the maintenance 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 composition 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 computation (e.g., string and arithmetic computation). It also provides a mechanism for users to create a temporary data pair (i.e., a variable name plus its value), and use this data to simulate user input data during the whole demonstration. At the same time, an event listener has been developed to monitor all the operations occurring in the model editor and collect the information for each operation in sequence.
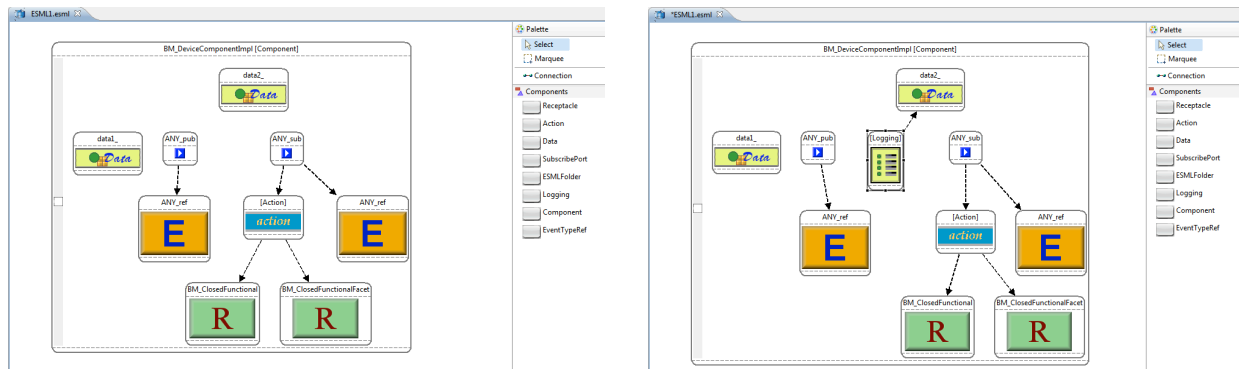

Figure 8. A selected Component before (left) and after (right) the demonstration of adding a logging mechanism

In Example 2.1, the task is to add a *Logging* mechanism to each *Data* in the *Component* whose name ends with "*impl*"; there also must be an *Action* in the *Component*. Therefore, the demonstration is performed on a selected *Component* (*BM_DeviceComponentImpl*) that qualifies these conditions as shown in the left of Figure 8.
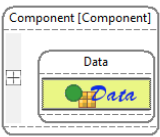
With a desired location, the demonstration is straightforward. Users can add a new *Logging* element in the *Component*, and set up its type attribute as "*LogOnRead*," after which a user makes a connection between the new *Logging* element to a *Data* element. List 1 shows all the operations performed during the demonstration, and the model after the demonstration is presented in the right of Figure 8. The demonstration only needs to be done for one of the *Data* elements, because the execution engine can automatically match all the other locations.

List 1. Operations performed for *Example 2.1* in the demonstration

Sequence	Operation Performed
1	Add a *Logging* in *ESMLRoot.BM_DeviceComponentImpl*
2	Set *Logging.LogType = "LogOnRead"*
3	Connect *Logging* to *data1*

Step 2 – Operation Optimization. The list of recorded operations indicates how a non-functional property should be composed in the base model. However, there is no guarantee that all operations in the demonstration 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 contradict each other and do not take effect in the transformation process. Thus, the operations are meaningless, even though they do not lead to an incorrect demonstration. Thus, after the demonstration, the engine optimizes the recorded operations to eliminate any meaningless actions. The optimization algorithm is specified in [29]. The operations in List 1 are all meaningful and are unchanged after this step.

Step 3 – Pattern Inference. With an optimized list of recorded operations, the transformation can be inferred by generalizing the behavior in the demonstration. Because the MTBD approach does not rely on any model transformation language, 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 non-functional property emerges) and the actions needed in a transformation (i.e., how a non-functional property should be maintained in this location).

Precondition		Actions
elem1.elem2		1. Add Logging in elem2
elem1.elem2.elem3		2. Set up Logging.LogType = "LogOnRead"
elem1: ESMLRoot		3. Connect Logging to elem3
elem2: Component		
elem3: Data		

Figure 9. The initial generalized pattern

As the initial generalized pattern, the precondition in this pattern is called the minimum precondition, which means that it only ensures that all the operations can be executed correctly with the minimum required model elements and connections. As shown in Figure 9, the precondition in this pattern defines that there must be a *Component* available, and in this *Component*, a *Data* element must exist. Obviously, this precondition provides sufficient operands for the operation actions to be executed correctly. However, in many cases, the generalized precondition is not restrictive and precise enough due to the limited expressiveness of the user demonstration. For example, although an *Action* is included in the *Component* in the demonstration, it is not incorporated in the generalized precondition, because this element was never used in any of the user operations. Similarly, we intentionally chose the *Component* called "*BM_DeviceComponentImpl*" as the desired location of our demonstration. The constraint on the name of this component is not inferred. Therefore, to provide further constraints on the precondition, more feedback is needed from users.

Step 4 – User Refinement. The inaccuracy of the initial pattern resulting from the limitation on the expressiveness of the user demonstration is a main issue undermining the applicability of MTBD to the maintenance tasks of non-functional properties, because it is often essential to accurately specify the exact locations of applying a non-functional property based on precise constraints. In order to address this problem, we extended the original version of MTBD to permit users to refine the inferred transformation by providing more feedback for the precondition of the desired transformation scenario from two perspectives – structure and attributes. For instance, users can restrict the precondition by selecting and
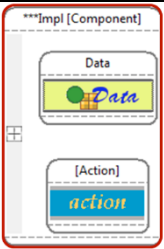
confirming extra model elements or connections in the model editors that must be included in the pattern. A new type of operation – *Confirm Containment*– is implemented in the editor for this purpose, which allows users to right-click on any model element in the editor and confirm to the engine that it should be contained in the final precondition. In addition, the precondition related with the implicit structure is also supported, such as replacing element *A* only if *A* has no incoming or outgoing connections. The refinement on the attributes can be realized by choosing the element in the demonstration and typing the specific conditions (e.g., add new element *B* in *C* only when the attribute value of *C* is greater than *200*). All the user refinements are still performed at the model instance level without being aware of the metamodel definitions, after which a transformation pattern will be finalized and stored in the pattern repository for future use.

Reconsidering Example 2.1, two more user refinement operations are needed in this step, as shown in List 2.

List 2. Refinement operations performed for Example 2.1 in the demonstration

Sequence	Operation Performed
4	Confirm the containment of ESMLRoot.*BM_DeviceComponentImpl.Action*
5	Specify precondition: *ESMLRoot.BM_DeviceComponentImpl.name.endsWith("Impl")*

Figure 10 shows the final transformation pattern after user refinement (the bold rectangle outline represents the attachment of preconditions on its attributes). The generated pattern will be stored in the repository for future reuse.

Precondition		Actions
elem1.elem2 (name.endsWith("Impl"))		1. Add Logging in elem2
elem1.elem2.elem3		2. Set up Logging.LogType = "LogOnRead"
elem1.elem2.elem4		3. Connect Logging to elem3
elem1: ESMLRoot		
elem2: Component		
elem3: Data		
elem4: Action		

Figure 10. The final generated pattern after user refinement

Step 5 – Pattern Execution. The final generated patterns can be executed on any model instances. Users can select a specific part of the model to apply the pattern, or by default apply the pattern on the whole model. 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 carrying out the transformation actions on the matched locations of the model. The MTBD engine has also been updated to validate the preconditions refined by users in the previous step. The execution can be carried out in any selected locations in the model, or to the whole model by default. The execution engine applies a backtracking algorithm that works similar to graph matching, which can be found in [29]. The selected existing model elements in a model instance serve as a candidate pool, and the precondition is matched using subsets of the pool to find the satisfied locations based on both the structural and attribute constraints.

Users can select the pattern in the execution controller dialog to execute an inferred transformation pattern from the repository. Users can select multiple patterns to execute in sequence, which is particularly useful when a model transformation task is divided by sub-tasks and specified by different demonstrations. In addition, the total times for executing the selected pattern(s) can be specified, because in some use cases (e.g., model scalability), a transformation pattern(s) needs to be executed multiple times to transform the model to a specific state and configuration.

Executing the generated pattern from Step 4 leads to the *Logging* element added to all the locations that qualify the conditions.

Step 6 – Correctness Checking and Debugging. Although the location matching the precondition guarantees that all transformation actions can be executed with the 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 on syntax and static semantics 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 has been created (out of the scope of this current paper's context) to enable end-users to track the execution of the transformation pattern without being exposed to low-level execution information.

3.2 Solving Non-functional Property Maintenance Tasks in QoSAML

The demonstration of adding a QoS transition strategy is performed on the selected *Size State*, as shown in Figure 11. Inside the *Size State*, we locate the two *States* with the proper *Pri* values, and perform the operations in List 3. The *Action* attribute configuration by operation 2 is conducted through the attribute refactoring editor, which allows users to access all the model elements and connections in the current model editor and load the attribute values to specify the computation process and evaluate the concrete value. The attribute refactoring editor also provides a mechanism to let users create a temporary data pair, with a given name and a value. The creation of the temporary data is actually used to simulate the user input process, and the data can be used in any attribute configuration and computation process through the entire demonstration. The creation of the temporary data will be generalized as a user input action and will display an input box when the final pattern is executed.

List 3. Operations performed for *Example 2.2* in the demonstration

Sequence	Operation Performed
1	Add a *Transition* between *QoSAMLRoot.OuterState.SizeState.State1* and *QoSAMLRoot.OuterState.SizeState.State2*
2	Set Transition.Action = *QoSAMLRoot.OuterState.SizeState.name* + " = " + *(QoSAMLRoot.OuterState.SizeState.State1.Max.value* + *QoSAMLRoot.OuterState.SizeState.State2.Max.value) / 2* = *"Size = 125"*
3	Create a temporary data pair (Name: *guard*, Value: *"Latency > 25 && FrameRate < 5"*)
4	Set *Transition.Guard = guard.value = "Latency > 25 && FrameRate < 5"*

The initial pattern generalized from the demonstration is shown in Figure 12. Similar to Example 2.1, the precondition is not accurate enough, because the relationship between the two *Pri* values are not reflected in the demonstration. Moreover, although the two *Max* elements are included in the pattern, they share the same type as *Min* and *Pri* (their meta type are all *Data*), the consequence being that it is possible that the execution engine incorrectly uses *Min* or *Pri* to calculate the average value, or use *Max* or *Min* to compare the *Pri* relationship. Thus, it is necessary to further restrict the *Data* involved in the pattern with their names. The following operations in List 4 are performed in the user refinement step. Operations 5 – 10 confirm the required *Pri* data elements and their relationship, while operations 11 – 12 ensure that *Max* data elements exist in the two *States*. Figure 13 shows the final generated transformation pattern.
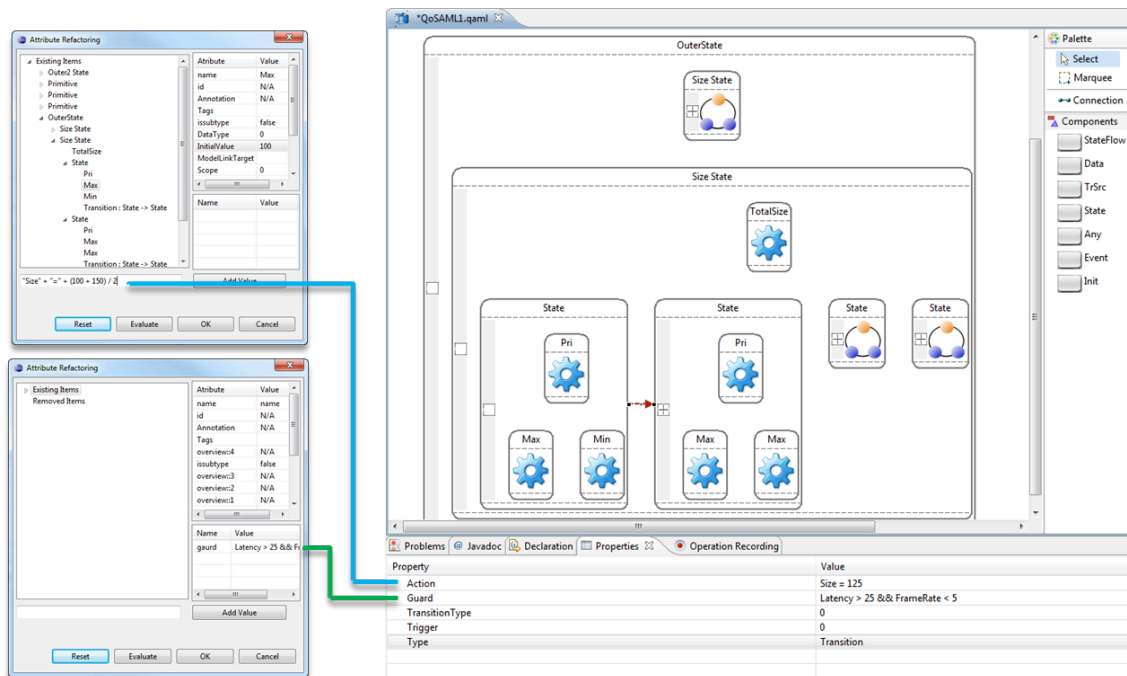
Figure 11. Demonstration of adding a transition and setting up the attributes for the new transition

List 4. Refinement operations performed for *Example 2.2* in the demonstration

Sequence	Operation Performed
5	Confirm the containment of *QoSAMLRoot.OuterState.SizeState.State1.Pri*
6	Confirm the containment of *QoSAMLRoot.OuterState.SizeState.State2.Pri*
7	Specify precondition *QoSAMLRoot.OuterState.SizeState.State1.Pri.name = "Pri"*
8	Specify precondition *QoSAMLRoot.OuterState.SizeState.State2.Pri.name = "Pri"*
9	Specify precondition *QoSAMLRoot.OuterState.SizeState.State1.Pri.value = QoSAMLRoot.OuterState.SizeState.State2.Pri.value - 1*
10	Specify precondition *QoSAMLRoot.OuterState.SizeState.State2.Pri.value < 5*
11	Specify precondition *QoSAMLRoot.OuterState.SizeState.State1.Max.name == "Max"*
12	Specify precondition *QoSAMLRoot.OuterState.SizeState.State2.Max.name == "Max"*

Executing the pattern on any selected *States* will make the execution engine automatically traverse the state and locate all the pairs of included *States* that satisfy the *Pri* relationship constraint and contains the needed *Max* elements. This allows the *Transition* to be added correctly and combined with a user input *Guard* value.
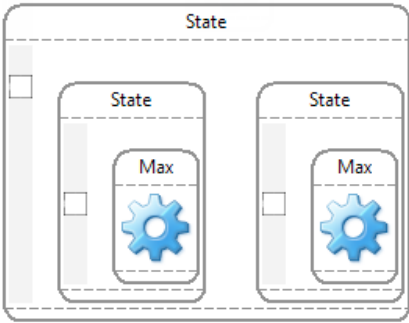
Precondition

elem1.elem2.elem3.elem4 elem1.elem2.elem3.elem5 elem1.elem2.elem3.elem4.elem6 elem1.elem2.elem3.elem5.elem7	
elem1: QoSAMLRoot elem2: State elem3: State elem4: State elem5: State elem6: Data elem7: Data	

Actions

1. Add a *Transition* between *elem4* and *elem5*
2. Set *Transition.Action* = *elem3.name* + "=" + (*elem6.value* + *elem7.value*) / 2
3. Create a data pair (*guard* = "*Latency > 25 && FrameRate < 5*")
4. Set *Transition.Guard* = *guard.value*

Figure 12. The initial generalized pattern

Precondition

elem1.elem2.elem3.elem4 elem1.elem2.elem3.elem5 elem1.elem2.elem3.elem4.elem6 (elem6.name == "Max") elem1.elem2.elem3.elem5.elem7 (elem7.name == "Max") elem1.elem2.elem3.elem4.elem8 (elem8.name == "Pri") elem1.elem2.elem3.elem5.elem9 (elem9.name == "Pri") (elem8.value == elem9.value – 1) (elem9.value < 5)	
elem1: QoSAMLRoot elem2: State elem3: State elem4: State elem5: State elem6: Data elem7: Data elem8: Data elem9: Data	

Actions

5. Add a *Transition* between *elem4* and *elem5*
6. Set *Transition.Action* = *elem3.name* + "=" + (*elem6.value* + *elem7.value*) / 2
7. Create a data pair (*guard* = "*Latency > 25 && FrameRate< 5*")
8. Set *Transition.Guard* = *guard.value*

Figure 13. The final generated pattern after user refinement
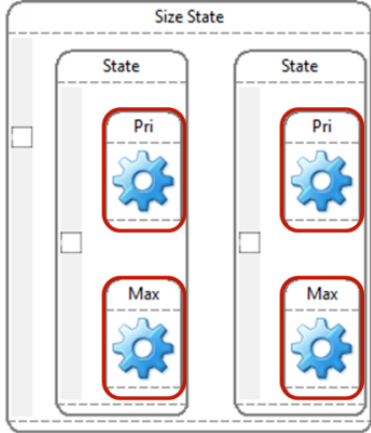
4. Comparison Between MTBD and Traditional Model Transformation Languages

The MTBD implementation is a plug-in called MT-Scribe that is hosted within the GEMS [13] modeling tool. MT-Scribe is triggered by the end user by pressing a "record" button in the model editor. Thus, any modeling language defined in GEMS that can be edited in the model editor can apply MTBD to address the maintenance of non-functional properties, which means MTBD is a general solution that can be applied across multiple DSMLs.

Using MTBD, users are only involved in editing model instances to demonstrate the non-functional property maintenance process and giving feedback after the demonstration. In other words, users only participate in Step 1 and Step 4 (Section 3.1). All of the other procedures (i.e., optimization, inference, generation, execution, and correctness checking) are fully automated. In both Step 1 and Step 4 where users are involved, all of the information exposed to users is at the model instance level, rather than the metamodel level. For instance, the demonstration is done using the basic editing operations in the GEMS model editor; the attribute configuration is accomplished using the attribute refactoring editor, which contains all the concrete attribute values from all the available elements and connections; the containment confirmation is simply realized by a one-click operation on the desired model element or connection; and the extra precondition is given using the dialog where users can access all the elements touched in the demonstration and type the constraints directly. The generated patterns are invisible to users (Figure 9, 10, 12, 13 are presented for the sake of explanation, which are not visible to users when using MTBD). Therefore, apart from the basic knowledge about the domain, users are isolated from the formal and abstract metamodel definitions and implementation details. Furthermore, no model transformation languages and tools are used in the implementation of MTBD. Thus, users are completely isolated from knowing model transformation languages or programming language concepts.

```
defines Start, FindData1, AddLog;

strategy FindData1()
{
    Atoms()->select(a | a.kindOf() == "Data")->AddLog();
}

strategy AddLog()
{
    declare parentModel : model;
    declare dataAtom, logAtom : atom;

    dataAtom := self;
    parentModel := parent();

    if (parentModel.atoms("")->select(m | m.kindOf() == "Action")->size() >= 1)
    then
        logAtom := parentModel.addAtom("Log", "LogOnWrite");
        logAtom.setAttribute("Kind", "On Write");
        parentModel.addConnection("AddLog", logAtom, dataAtom);
    endif;
}

strategy Start()
{
    rootFolder().findFolder("ComponentTypes").models().select(m | m.name()
        .endWith("impl"))->FindData1();
}
```

Figure 14. The ECL code to implement Example 2.1 [23]

```
defines AddTransition, FindConnectingState, ApplyTransitions;
```

```
strategy AddTransition(stateName, prevID, guard : string; prevPri : integer)
{
    declare pri, minVal, maxVal, avgVal : integer;
    declare endID : string;
    declare aConnection : node;
    findAtom("Priority").findAttributeNode("InitialValue").getInt(pri);
    if(pri == prevPri + 1)
    then
        getID(endID);
        findAtom("Min").findAttributeNode("InitialValue").getInt(minVal);
        findAtom("Max").findAttributeNode("InitialValue").getInt(maxVal);
        avgVal := (minVal + maxVal) / 2;
        <<CComBSTR action(stateName);     // exit statement to C++
        action.Append("="+XMLParser::itos(avgVal)); >>
        aConnection :=
        parent().addConnection("Transition", "Transition", "Transition",
                               endID, prevID);
        aConnection.addAttribute("Guard", guard);
        aConnection.addAttribute("Action", action);
    endif;
}

strategy FindConnectingState(stateName, guard : string)
{
    declare pri : integer;
    declare startID : string;
    findAtom("Priority").findAttributeNode("InitialValue").getInt(pri);
    getID(startID);
    if(pri< 4)
    then
        parent().models("State")->
        forAll(AddTransition(stateName, startID, guard, pri));
    endif;
}

strategy ApplyTransitions(stateName, guard : string)
{
    declare theModel : node;
    theModel := findModel(stateName);
    theModel.models("State")->forAll(FindConnectingState(stateName, guard));
}
```

Figure 15. The ECL code to implement Example 2.2 (from [16])

To compare MTBD to traditional usage of transformation languages, we considered MTBD with the actual transformation rules used to specify the same transformation tasks for the two motivating examples [15][16]. These transformation rules were used by C-SAW (Constraint-Specification Aspect Weaver) [15][23], as shown in Figures 14 and 15. We compare the two approaches by analyzing the specific implementation differences as well as some general development summaries in Section 4.1 and 4.2. As a special type of MTLs, we discuss the differences and advantages of MTBD over graphical MTLs in Section 4.3.

4.1 Comparison on specific implementation differences between MTBD and C-SAW

Identifying the desired locations of the non-functional properties that must be maintained is very easy using MTBD (i.e., a user just selects an example of the specific location using an instance model). In the transformation rules sent to C-SAW, the location of the non-functional properties is defined by extended OCL constraints (e.g., `forAll(), select()`) together with APIs provided in the transformation language (e.g., `models("State"), atoms()`). The process becomes more complex when the different APIs are called and used together in a single statement. By contrast, the main location specification in MTBD is automatically handled in the demonstration process. It is the recording engine that detects the location of

the operation happening and generalizes the location context information, so that users focus on selecting a desired location without being aware of a generalized locating process.

The specific constraints on the preconditions using MTBD are more intuitive and direct than writing transformation rules. By selecting and clicking on the desired model elements or connections, constraints on the structure can be specified. The location and selection of the attribute in MTBD is realized by clicking on the element in the precondition specification dialog and providing much simpler expressions based on the instance model. However, in C-SAW, OCL expressions and condition statements need to be applied (e.g., `if …select(m | m.kindOf() == "Action")->size() >= 1`). When it comes to defining the precondition on attribute values, we believe that MTBD is simpler than using conditional statements that access the modeling tool APIs. For example, the following is an expression that would be needed in a typical model transformation rule using the traditional approach:

```
findAtom("Priority").findAttributeNode("InitialValue").getInt(pri);
```

Non-functional property maintenance and composition are implemented using model manipulation APIs in C-SAW (e.g., `parent().addConnection("Transition", "Transition", "Transition",endID, prevID), aConnection.addAttribute("Guard",guard)`), while using MTBD, the composition process is demonstrated using the basic operations in the model editor (i.e., add, delete, update attributes).

4.2 Comparison on development efforts between MTBD and C-SAW

We have not performed a formal user study on the comparison between the two approaches. However, some general summaries related to development using the two approaches are given. Table 1 lists the concrete effort to create the deliverable transformation artifact, indicating that with the demonstration approach, only a small number of operations are needed using MTBD to accomplish the exact same tasks that were done by writing dozens of lines of transformation code. To better measure the effort, we asked two users to accomplish the two tasks using these two approaches so that we could measure the time spent using each approach. The user who used MTBD was a trained MTBD user, while the other user who wrote the transformation rules is one of the C-SAW co-authors. The result in Table 2 shows that using MTBD to solve these two examples is 10 times faster than writing C-SAW rules. This fast development cycle enables users to quickly test patterns and make modifications. For instance, in order to determine if a generated pattern is too general or not, users need to execute the pattern and test the result just as testing programs are tested. In some cases, modifications have to be made to fix errors followed by re-executing the new pattern. MTBD shortens these development iterations.

Table 3 describes the prerequisite knowledge for each approach. For MTBD, the development environment is integrated in the modeling tool itself, so users only need to know a few additional buttons to use MTBD. The demonstration is performed using the same model editing operations, so the only new things to learn are the 3 types of UIs for inputting pattern refinements. For C-SAW, the development environment is a plug-in to GME, and it requires a regular textual editor to specify the transformation rules. Learning C-SAW requires learning the new syntax and semantics of the language, including a number of new keywords, program structures and functions. Finally, we identified the possible errors that could happen in the development process using the two approaches. For MTBD, users might perform incorrect editing operations or give invalid expressions in the precondition specification and refinement. The base pattern and minimum preconditions are automatically generated at the background, so there is no way that users can make mistakes on this part. For C-SAW or nearly any other textual or visual transformation programming language, the transformation rules could suffer from general programming issues such as the syntax errors and semantics bugs in the logical perspective.

Example	MTBD	C-SAW Rules

Example 2.1	3 editing operations 2 precondition refinement (1 mouse-click on 1 element, 1 string expression constraint on an attribute)	23 SLOC
Example 2.2	4 editing operations 8 precondition refinement (2 mouse-click on 2 elements, 4 string expression constraints on 4 attributes, 2 numerical constraints on 2 attributes)	40 SLOC

Table 1. The comparison of development effort to solve the two motivating examples

Example	MTBD	C-SAW Rules
Example 2.1	30 seconds for demonstration 1.5 minute for refinement	20 minutes
Example 2.2	40 seconds for demonstration 4 minutes for refinement	35 minutes

Table 2. The comparison of development time to solve the two motivating examples

MTBD	C-SAW
1. MTBD development environment 2. How to demonstrate by model editing 3. How to perform refinement (i.e., the usage of 3 types of refinement UIs)	1. C-SAW syntax and semantics including 12 keywords, 5 main program structure, and 4 functions. 2. C-SAW development environment

Table 3. The comparison of prerequisite knowledge to solve the two motivating examples

MTBD	C-SAW
1. Demonstrate with the wrong editing operation 2. The incorrect specification on the attribute precondition expressions	1. General programming language syntax error. 2. Semantics error or bugs on the transformation logic

Table 4. The comparison of possible errors to solve the two motivating examples

4.3 Comparison with graphical MTLs

Besides the textual C-SAW transformation comparison, MTBD also shows advantages over general graphical MTLs (e.g., GReAT [2]). Graphical MTLs specify the transformation rules by configuring the source model in the left side and the target model in the right side. The transformation engine matches the source model and replaces it with the target model. Compared with the textual MTLs, graphical MTLs provides a more visual and straightforward approach to specify the rules. However, in order to specify a transformation rule using graphical MTLs, users must map the concrete model elements with the concrete syntax of a certain domain to the syntax of the graphical MTL they choose. Depending on the specific domain and MTLs, the syntax and semantics gap of converting the desired model transformation scenario into a rule varies. Additionally, the rules are still defined using metamodel definitions even though they are represented graphically. Similar to graphical MTLs, the execution engine of MTBD applies a similar semantics to match a precondition and carry out the execution process, but MTBD hides the left side and

right side rules (the right side rules in MTBD are in the format of an operational change list as described in Section 3) and avoids the gap by the demonstration process. Users specify the transformation on concrete model instances by directly editing, without having to convert the idea into another syntax format.

Apart from the general graphical MTLs, some innovative new graphical MTLs (e.g., MATA [33]) provide and improved user experience and ease of use by applying concrete syntax to specify the rules without the knowledge of metamodel definitions, as well as combining the left hand side rule and right hand side rule together into a single view. This type of graphical MTLs shows similar user experience as MTBD. However, there are still some fundamental differences between the two approaches as follows:

First, MTBD is not designed as a language or programming language, but as a new user experience and interface integrated with the model editing environment to perform model transformations. Thus, programming concepts are avoided in the MTBD process. Users demonstrate transformations rather than writing programmatic rules to specify them. The underlying tool takes care of deriving the programmatic expressions to generalize the demonstrated transformation. Some of the expressions used in the precondition specification are programmatic in nature, but have no conditional logic, loops, or complex expressions, and their usage is minimized through intuitive UIs (e.g., dialog showing the related model element and connections) to assist users' specification. By comparison, graphical MTLs such as MATA are still programming languages based on a grammar or specific metamodel (e.g., UML), with its own syntax and semantics. Although it is really easy to use, users still need to have the basic knowledge about a programming language or graph transformations to correctly utilize the language. For many domains, users do not have sufficient experience with programming language or graph transformation concepts to write transformation rules.

Furthermore, MTBD is designed to be at a higher level of abstraction than the graphical MTLs. This results in the fact that a lot of low-level implementation details are hidden (e.g., patterns are generated and executed based on graph theories in MTBD, but are invisible to end-users). On the other hand, the higher level of abstraction actually means that not all possible transformations can be expressed. We did not design MTBD to support all types of model transformation and all possible non-functional aspect modeling activities. A lot of tasks that can be easily specified using regular MTLs cannot be directly demonstrated using MTBD (e.g., using max/min in precondition specification, non-functional properties weaving based on multiple metamodels). However, we target the most practical and commonly used transformation scenarios in different domains and try to ensure the improved user experiences.

Finally, we would also like to highlight the fact that the target user group of MTBD is a little different from the general MTLs. We expect the main users of MTBD to be domain experts using DSMLs to build models, who do not necessarily have any programming or graph transformation background. For computer scientists or programmers, we believe that they may find it more efficient and comfortable to write specific rules using MTLs to accomplish certain tasks rather than using demonstration based approach, but for users without programming background, MTBD offers a feasible alternative for them to finish certain basic model transformation tasks in a potentially simpler way without a steep learning curve. With so many DSMLs being emerging, this group of users is becoming increasingly important in model-driven engineering community.

5. Related Work

Non-functional system properties in DSMLs have been well-investigated [5][6], ranging from the specification of non-functional properties to the measurement and analysis of the non-functional properties. Troya et al. [30] regard non-functional properties as dynamic semantics and specify the properties using domain-specific observers. Boulanger et al. [7] propose a multi-view modeling approach

to model both functional and non-functional viewpoints. Berardinelli et al. [4] apply a unifying conceptual framework to model context-aware aspects of mobile software systems. Yrjönen and Merilinna [35] propose a non-functional properties framework to check the satisfaction of specified requirements. Kupfer and Hadar [21] present a methodology to achieve predefined non-functional properties by understanding and representing deployment requirements.

Regarding the maintenance of non-functional properties, the current main practice is to use MTLs. For example, Paige et al. [26] propose to use Epsilon [10] to capture failure behavior. Of course, the traditional model transformation languages and tools are also applicable to support automating the maintenance of non-functional properties in DSMLs. The most direct way is to use general-purpose porgraming language (GPL) such as Java or C++ to access and manipulate the internal structure of a model instance using an API provided by a host modeling tool [28], but this approach is not feasible for end-users who do not have programming experience. The power of a transformation is often restricted by the supported API within the modeling tool.

Many modeling tools support importing and exporting model instances in the form of XMI. Therefore, it is also possible to use existing XML tools (e.g., XSLT [32]) to manage the non-functional properties outside of a modeling tool using XMI as an intermediate representation. However, the problem of XSLT is that it is tightly coupled to XML, requiring experience to define the transformations using concepts at a lower level of abstraction than DSMLs.

Model transformation languages are the most popular approach to support non-functional property maintenance, because the process of maintaining the properties in the base model can be considered as a model transformation. Textual MTLs specify the transformation rules using high-level expressions and statements. C-SAW [15], ATL [18], and QVT [25] are representative examples of powerful MTLs that can be used to manage the weaving of models. Moreover, graphical MTLs (e.g., GreAT, VIATRA [3]) also exist to convert non-functional property maintenance tasks into a graph transformation problem by utilizing graph matching and rewriting techniques. Compared with textual MTLs, it is easier to define specific model patterns using graphs, leading to a simplification of the transformation rules in many cases. However, whether a MTL has a high level of abstraction, graphical or textual, its usage always suffers from the challenges mentioned in Section 1 (i.e., the steep learning curve and need to understand the details of the metamodel).

An alternative approach to simplify the maintenance of non-functional properties is to use end-user model transformation approaches, as proposed in this paper. To address the challenges inherent from using MTLs, Model Transformation By Example (MTBE) [31][34] was developed so that instead of writing transformation rules manually, users are asked to build a prototypical set of interrelated mappings between the source and target model instances, and then the metamodel-level transformation rules are semi-automatically generated. This approach simplifies model transformation implementation to some extent, but is not appropriate for non-functional property maintenance tasks because: 1) it focuses on direct concept mapping between two different domains rather than maintaining property models within the same domain; 2) they do not support attribute transformation, preventing the automatic setup of attributes needed in many maintenance tasks.

Another work has been described by Brosch et al. [8], which uses an example-based approach to address model refactoring tasks. Because it supports model transformation within the same domain, it also has potential to be applied in non-functional property maintenance scenarios. However, the user feedback step may not be at the proper level of abstraction in their approach, and complex attribute transformation is not provided.

EMF Refactor [11] is a new open source component that provides tool support for generating and applying refactoring for models based on EMF. The capabilities of EMF Refactor simplify the process of adding new refactoring functions in EMF editors and supporting model evolution activities, which can be used also to address the maintenance of non-functional properties. However, the initial definition of the

maintenance rules is based on EMF Tiger [12], a graphical MTL, which is based on writing model transformation rules.

6. Conclusion and Future Work

In this paper, we presented an end-user approach to enable general users (e.g., domain experts or non-programmers) to manage the non-functional properties in different DSMLs without knowing model transformation languages or metamodel definitions. The MTBD approach is based on demonstrations made in a modeling tool by the end-user, which are then generalized. We have shown through two examples how non-functional properties can be introduced using demonstrations provided by a domain expert.

As future work, we will investigate using MTBD to support the maintenance of non-functional properties across different domains and metamodels (e.g., separation of common non-functional properties in a way that can be reused across multiple modeling languages). For instance, the base model may need to be transformed to a new domain first and then apply the non-functional properties, but the current implementation of MTBD cannot handle this type of maintenance because of the nature of supporting only endogenous model transformation in MTBD. Extending the MTBD idea to exogenous transformation will enlarge its applications in practice.

Due to the higher level of abstraction, which allows domain experts to manage non-functional properties, MTDB is less powerful and expressive than a well-defined transformation language. Some functions (e.g., *getMax*(), *getMin*()) can be simply realized using function calls or a customized part of code, which can be very challenging to demonstrate. Therefore, in order to enhance the functionality of MTBD, those commonly used functions in the maintenance of non-functional properties will be identified, and integrated in the demonstration and user refinement steps. Furthermore, there are also some features that can be done to improve the demonstration user experience. For instance, when confirming a containment of an element in the user refinement step, it would be a good feature to automatically include the basic constraints on not only the element type but also the attributes (e.g., name).

7. Acknowledgement

References

[1] Agrawal, A., Karsai, G., Lédeczi, Á.: An End-to-End Domain-Driven Software Development Framework. International Conference on Object-Oriented Programming, Systems, Languages, and Applications - Domain-driven Track, Anaheim, CA, October 2003, pp. 8-15 (2003)

[2] Balasubramanian, D., Narayanan, A., Buskirk, C., Karsai, G.: The Graph Rewriting and Transformation Language: GreAT. Electronic Communication of the European Association of Software Science and Technology, vol. 1, 8 pages (2006)

[3] Balogh, Z., Varró, D.: Advanced Model Transformation Language Constructs in the VIATRA2 Framework. Symposium on Applied Computing (SAC), Dijon, France, April 2006, pp. 1280-1287 (2006)

[4] Berardinelli, L., Cortellessa, V., Marco, A.: An Unified Approach to Model Non-Functional Properties of Mobile Context-Aware Software. The 2nd International Workshop on Non-Functional System Properties in Domain-Specific Modeling Languages, Denver, CO, October 2009.

[5] Bošković, M., Gašević, D., Pahl, C., Schätz, B.: The 1st International Workshop on Non-Functional System Properties in Domain-Specific Modeling Languages. Models in Software Engineering, Springer-Verlag LNCS 5421, pp. 227-228 (2010)

[6] Bošković, M., Gašević, D., Pahl, C., Schätz, B.: The 2nd International Workshop on Non-Functional System Properties in Domain-Specific Modeling Languages. Models in Software Engineering, Springer-Verlag LNCS 6002, pp. 291-295 (2010)

[7]   Boulanger, F., Jacquet, C., Rouis, E., Hardebolle, C.: Modeling Heterogeneous Points of View with ModHel'X. The 2nd International Workshop on Non-Functional System Properties in Domain-Specific Modeling Languages, Denver, CO, October 2009 (2009)

[8]   Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: An Example is Worth a Thousand Words: Composite Operation Modeling By-Example. International Conference on Model Driven Engineering Languages and Systems (MoDELS), Springer-Verlag LNCS 5795, Denver, CO, October 2009, pp. 271-285 (2009)

[9]   Burnett, M., Cook, C., Rothermel, G.: End-user Software Engineering. Communications of the ACM, vol. 47, no. 9, pp. 53-58 (2004)

[10]   Eclipse Epsilon, http://www.eclipse.org/gmt/epsilon/ (2010)

[11]   EMF Refactor, http://www.mathematik.uni-marburg.de/~swt/modref/ (2010)

[12]   EMF Tiger, http://tfs.cs.tu-berlin.de/emftrans/ (2010)

[13]   Generic Eclipse Modeling System (GEMS). http://www.eclipse.org/gmt/gems/ (2010)

[14]   Graphical Modeling Framework (GMF), http://www.eclipse.org/modeling/gmf/ (2010)

[15]   Gray, J., Lin, Y., Zhang, J.: Automating Change Evolution in Model-Driven Engineering. IEEE Computer, Special Issue on Model-Driven Engineering, vol. 39, no. 2, pp. 51-58 (2006)

[16]   Gray, J., Neema, S., Zhang, J., Lin, Y., Bapty, T., Gokhale, A., Schmidt, D.: Concern Separation for Adaptive QoS Modeling in Distributed Real-Time Embedded Systems. Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation, Idea Group, Chapter 4, pp. 85-113 (2009)

[17]   Gray, J., Tolvanen, J., Kelly, J., Gokhale, A., Neema, S., Sprinkle, J.: Domain-Specific Modeling. Handbook of Dynamic System Modeling, CRC Press, Chapter 7, 7-1 through 7-20 (2007)

[18]   Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming, vol. 72, nos. 1/2, pp. 31-39 (2008)

[19]   Karr, D., Rodrigues, C., Loyall, J., Schantz, R., Krishnamurthy, Y., Pyarali, I., Schmidt, D.: Application of the QuO Quality-of-Service Framework to a Distributed Video Application. International Symposium on Distributed Objects and Applications, Rome, Italy, pp. 299-309 (2001)

[20]   Karsai, G., Neema, S., Sharp, D.: Model-Driven Architecture for Embedded Software: A Synopsis and an Example. Science of Computer Programming, vol. 73, no. 1, pp. 26-38 (2008)

[21]   Kupfer, M., Hadar, I.: Understanding and Representing Deployment Requirements for Achieving Non-Functional System Properties. The 1st International Workshop on Non-functional System Properties in Domain-Specific Modeling Languages, Toulouse, France, October 2008 (2008)

[22]   Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-specific Design Environments. IEEE Computer, vol. 34, no. 11, pp. 44-51 (2001)

[23]   Lin, Y.: A Model Transformation Approach to Automated Model Evolution, Ph.D. Thesis, University of Alabama at Birmingham (2007)

[24]   Object Management Group, Object Constraint Language Specification. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL. (2010)

[25]   Object Management Group, Revised Submission for MOF 2.0 Query/View/Transformations RFP (ad/2002-04-10), OMG Document ad/2005-07-01 (2005)

[26]   Paige, R., Rose, L., Kolovos, D., Brooke, P., Ge, X.: Automated Safety Analysis for Domain-Specific Modeling Languages. The 1st International Workshop on Non-Functional System Properties in Domain-Specific Modeling Languages, Toulouse, France, October 2008 (2008)

[27]   Schmidt, D.: Special Issue on Model-Driven Engineering. IEEE Computer, vol. 39, no. 2, pp. 25-32 (2006)

[28]   Sendall, S., Kozaczynski, W.: Model transformation - The Heart and Soul of Model-Driven Software Development. IEEE Software, Special Issue on Model Driven Software Development, vol. 20, no. 5 , pp. 42–45 (2003)

[29] Sun, Y., White, J., Gray, J.: Model Transformation by Demonstration. Model Driven Engineering Languages and Systems (MoDELS), Springer-Verlag LNCS 5795, Denver, CO, October 2009, pp. 712-726 (2009)

[30] Troya, J., Rivera, J., Vallecillo, A.: On the Specification of Non-Functional Properties of Systems by Observation. The 2nd International Workshop on Non-Functional System Properties in Domain-Specific Modeling Languages, Denver, CO, October 2009.

[31] Varró, D., Balogh, Z.: Automating Model Transformation by Example using Inductive Logic Programming. Symposium on Applied Computing (SAC), Seoul, Korea, March 2007, pp. 978-984 (2007)

[32] W3C, XSLT Transformation version 1.0.http://www.w3.org/TR/xslt (1999)

[33] Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: A Unified Approach for Composing UML Aspect Models based on Graph Transformation. Transactions on Aspect-Oriented Software Development VI, vol. 5560/2009, pp. 191–237 (2009)

[34] Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards Model Transformation Generation By-Example. Hawaii International Conference on System Sciences (HICSS), Big Island, HI, January 2007, pp. 285 (2007)

[35] Yrjönen, A., Merilinna, J.: Extending the NFR Framework with Measurable Non-Functional Requirements. The 2nd International Workshop on Non-Functional System Properties in Domain-Specific Modeling Languages, Denver, CO, October 2009.