A Demonstration-based Model Transformation Approach to Automate Model Scalability

[1]Yu Sun, [2]Jules White, [3]Jeff Gray

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

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

[3]Department of Computer Science, University of Alabama
gray@cs.ua.edu

Abstract

Model-Driven Engineering (MDE) is playing an increasingly more significant role in software development. An important aspect of software development is scaling software models in order to handle design forces, such as enlarging and upgrading system features, or allocating more resources to handle additional users. Model scalability is the ability to refactor a base model, by adding or replicating the base model elements, connections or substructures, in order to build a larger and more complex model to satisfy new design requirements. Although a number of modeling tools have been developed to create and edit models for different purposes, mechanisms to scale models have not been well-supported. In most situations, models are manually scaled using the basic point-and-click editing operations provided by the modeling environment.

Manual model scaling is often tedious and error-prone, especially when the model to be scaled has hundreds or thousands of elements and the scaling process involves entirely manual operations. Although model scaling tasks can be automated by using model transformation languages, writing model transformation rules requires learning a model transformation language, as well as possessing a great deal of knowledge about the metamodel. Model transformation languages and metamodel concepts are often difficult for domain experts to understand. This requirement to learn a complex model transformation language exerts a negative influence on the usage of models by domain experts in software development. For instance, domain experts may be prevented from contributing to model scalability tasks from which they have significant domain experience.

In order to automate model scalability tasks, this paper presents a demonstration-based approach to automate model scaling. Instead of writing model transformation rules explicitly, users demonstrate how to scale models by directly editing the concrete model instances and simulating the model replication processes. By recording a user's operations, an inference engine analyzes the user's intention and generates generic model transformation patterns automatically, which can be reused to scale up other model instances. Using this approach, users are able to automate scaling 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.

Keywords

Model Evolution, Model Scalability, Model Transformation by Demonstration

1 Introduction

Software systems often need to evolve in order to accommodate new features, to process larger workloads, or to handle other scaling issues [1]. Scaling a software system is crucial for its long-term success. With the increasing adoption of Model-Driven Engineering (MDE) [2], models are emerging as a high level abstraction of software systems. The focus on models as first-class entities in many domains (e.g., automotive and avionics domains) has promoted models to an important role in software development.

When scaling a software system in the context of MDE, it is common to scale the related software models, rather than the low-level software artifacts (e.g., source code). For instance, feature models [34] are used as design models in software product-lines to configure the components of a software system, such that adding new product functionality often consists of adding new feature elements to a model. Domain-specific models [33] can be built to specify software systems and generate implementation code, which means that expanding the implementation of a software system is based on scaling the corresponding domain-specific models. Moreover, when a software system is about to be deployed, deployment models can be used to specify how to allocate software to the underlying hardware infrastructure [32] and to monitor and control the infrastructure at runtime [31]. In order to allocate additional infrastructure to handle larger workloads, the underlying deployment models must be scaled. Thus, model scalability [6] (i.e., the ability to build a complex model from a base model by adding, replicating, or modifying its model elements, connections or substructures) is an important aspect of MDE-based software evolution.

To support model scalability, the host modeling tool must allow users to rapidly change the model representation [5]. Although manually editing and scaling models is the most direct approach, it is obviously laborious, time consuming and error prone, particularly when a large number of model elements and connections exist. Editing a large model may require a staggering amount of clicking and typing operations within the modeling tool [6]. Therefore, the process of scaling models can benefit immensely from automation.

Model transformation has proven to be an effective approach to automate model scalability tasks [6]. Scaling a base model to a more complex model is a type of model transformation. More specifically, this type of scaling is an endogenous model transformation (i.e., model transformations within the same metamodel or the same domain) [7]. A number of executable Model Transformation Languages (MTLs) have been developed to assist users in specifying

the transformation rules that describe how to scale a model from a base state to a desired more complex state.

Although model transformation languages are powerful and expressive approaches to automating some model scalability tasks, adopting an MTL is not always the ideal solution. Firstly, even though most MTLs are high-level and declarative languages, they have a steep learning curve due to the complexity of their syntax, semantics, and other special features (e.g., OCL [30] specification is used in many MTLs). This learning curve is particularly apparent for domain experts, such as automotive engineers, who are not computer scientists and have not received training on the use of MTLs. Furthermore, model transformation rules are often defined at the metamodel level, rather than in the context of a concrete model instance, which exposes users to metamodel concepts not specific to the modeling language.

Developing a deep and clear understanding of a metamodel is challenging, especially for large and complex domains. In some cases, domain concepts may be hidden in the metamodel and difficult to unveil [29], which makes comprehension more difficult. In the context of MDE, more general users (e.g., domain experts or non-programmers) can participate in building and using software models. The difficulties associated with using MTLs may prevent these users from contributing to certain model scaling tasks from which they have a large amount of domain experience.

Our contribution in this paper is an innovative approach to automate model scalability tasks, so that domain experts are able to implement model evolution tasks without using a model transformation language and without having to understand the metamodel definition. The approach described in this paper extends our previous work, Model Transformation By Demonstration (MTBD) [10]. MTBD simplifies the implementation of model transformations by inferring transformation patterns from a user's demonstrated operations to transform a concrete model instance. Several new extensions and features have been made to enhance our original approach (i.e., MT-Scribe, which is our implementation of the MTBD concept) so that it can be adapted to handle special needs related to model scalability. We have applied our approach to a number of model scalability scenarios that were previously performed by manually writing transformation rules to demonstrate the reduction in manual effort that our approach provides. Finally, we have created a new formal model of MTBD to precisely define its semantics and inference techniques.

The rest of the paper is organized as follows. Three model scalability scenarios in different domains are presented in Section 2 to motivate the need to support software evolution by automating model scalability. In Section 3, the original MTBD project combined with its limitations in dealing with model scalability are introduced, followed by a presentation of

new extensions and features that have been added to address those problems. The solutions to solve the three motivating examples using the extended MTBD are then given in Section 4. Section 5 evaluates the new approach, pointing out its advantages and limitations. Related works and techniques are compared in Section 6, with Section 7 offering concluding remarks.

2 Motivating Examples Illustrating Model Scalability Issues

This section presents three examples that motivate the need for automating model scalability to support software evolution in different phases of software development – design, implementation, and maintenance. For each of the three examples in Sections 2.1, 2.2 and 2.3, background information about the specific application domain and context will be given, followed by an illustration using a concrete model instance. Then, we present a typical scaling evolution scenario in the domain, as well as a desired model instance after the scaling process. The challenges of accomplishing these model scalability tasks will be summarized in Section 2.4. The specific approach for using MTBD to address the needs arising from these examples will be given in Section 4.

2.1 Adding New Event Types: Evolving Software Design Models

Stochastic Reward Nets (SRNs) [13] can be used for evaluating the reliability of complex distributed systems. SRNs have been used extensively for designing and modeling reliability and performability of different types of systems. The Stochastic Reward Net Modeling Language (SRNML) was developed to describe SRN models of large distributed systems [6], which shares 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 1 depicts the Reactor pattern [21] in middleware for network services, providing mechanisms to handle synchronous event demultiplexing and dispatching.

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 and the associated execution snapshot. The execution snapshot depicts the underlying mechanism for handling the event types included in the top part, so any change made to the event types will require corresponding changes to the snapshot. In Figure 1, the original model has two event types, *1* and *2*, each from its arrival (e.g., *A1*), to queuing (e.g., *Sn1*) and finally service (e.g., *Sr1*) through the immediate transitions (e.g., *B1, S1*). It also models the process of taking

successive snapshots and non-deterministic service of event handles in each snapshot through some snapshot transitions and places (e.g., *StSnpSht*, *TStSnp1*, *TProcSnp1,2*).

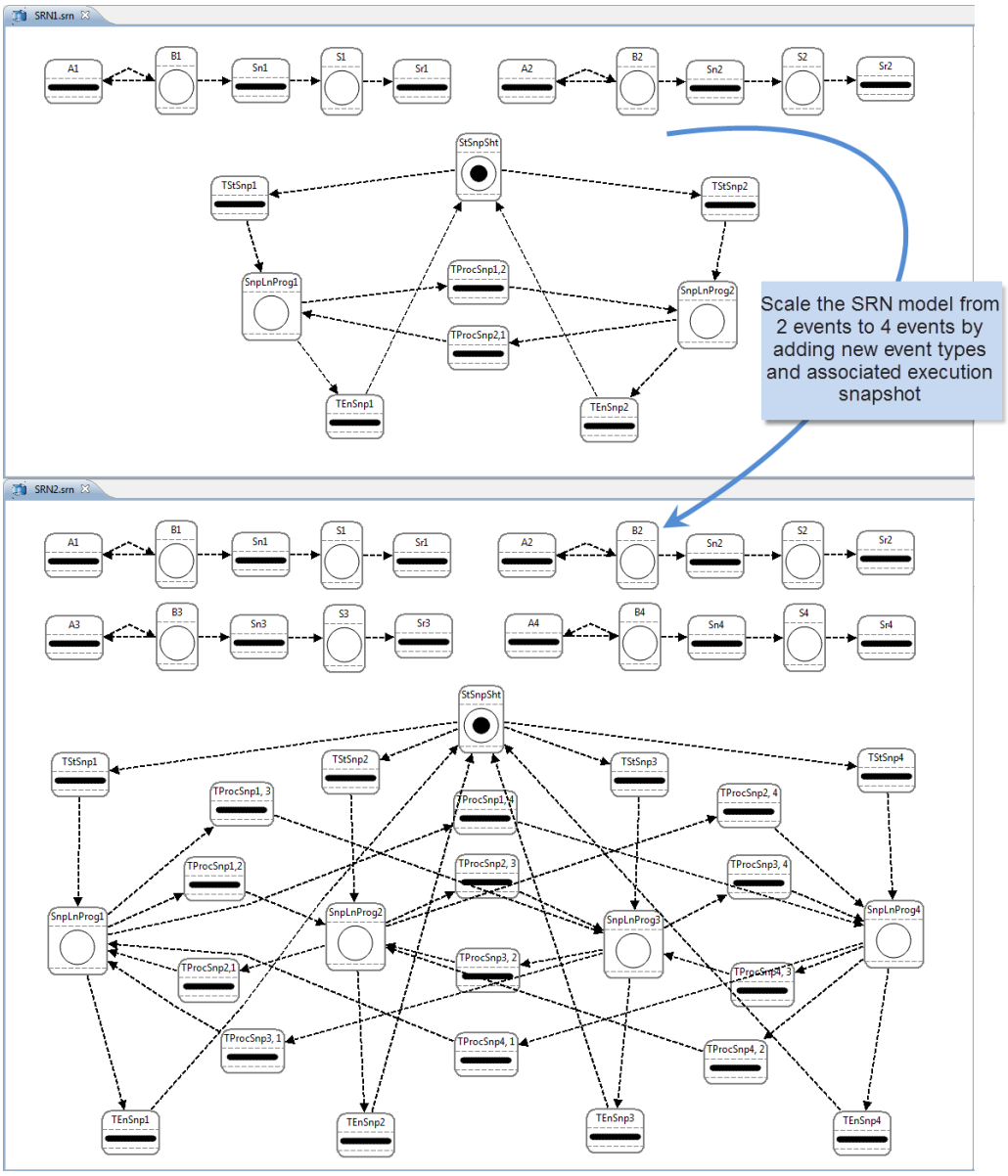


Figure 1. An SRN model before (top) and after (bottom) scaling

Scalability Scenario in SRNML (Example 2.1): The scalability challenges of SRN models arise from the addition of new event types and connections between their corresponding event handlers. As shown in the bottom of Figure 1, 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*), transitions from starting place to end place (i.e., *TStSnp3*, *TEnSnp3*, *TStSnp4*, *TEnSnp4*), 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.

2.2 Enlarging Event Services: Evolving Software Implementation Models

The Event QoS Aspect Language (EQAL) [11] is a Domain-Specific Modeling Language (DSML) for graphically specifying publisher-subscriber service configurations for large-scale distributed systems. Publisher-subscriber mechanisms, such as event-based communication models, help to reduce software dependencies, enhance system composability and evolution, and allow application components to communicate anonymously and asynchronously, particularly in large-scale distributed and real-time embedded (DRE) systems (e.g., avionics mission computing, distributed interactive simulations). Several EQAL model translators have been implemented to take EQAL models as input and generate publisher-subscriber service configuration files, component property descriptions, and part of the underlying code to support system implementation.

The top of Figure 2 illustrates a model defined by EQAL to specify a federation of event channels in different sites, which allows sharing of filtering information and event communications in the channels through CORBA gateways. A *Site* can contain an *EventChannel*, multiple *Gateways*, an *EventConsumer*, an *EventSupporter*, and multiple *EventTypeRefs*. Connections can be built between the *EventChannel* and *Gateway*, as well as *EventConsumer*, *EventSupporter* and *EventTypeRef*.

Scalability Scenario in EQAL (Example 2.2): One complex scalability issue in EQAL arises when a small federation of event services must be scaled to a very large system, which usually accommodates a large number of publishers and subscribers. The bottom of Figure 2 shows a federated event service with five *Sites*, which is scaled up from the federated event services with three *Sites*. The scaling process involves adding new *Sites* that contain an *EventChannel*, a number of *Gateways* (the number of *Gateways* depends on the number of existing *Sites*), an *EventSupplier*, an *EventConsumer*, two *EventTypeRefs* and the connections among them. In addition, new *Gateways* need to be added to each original *Site* and new connections need to be built to connect the new *Site* with original *Sites*.

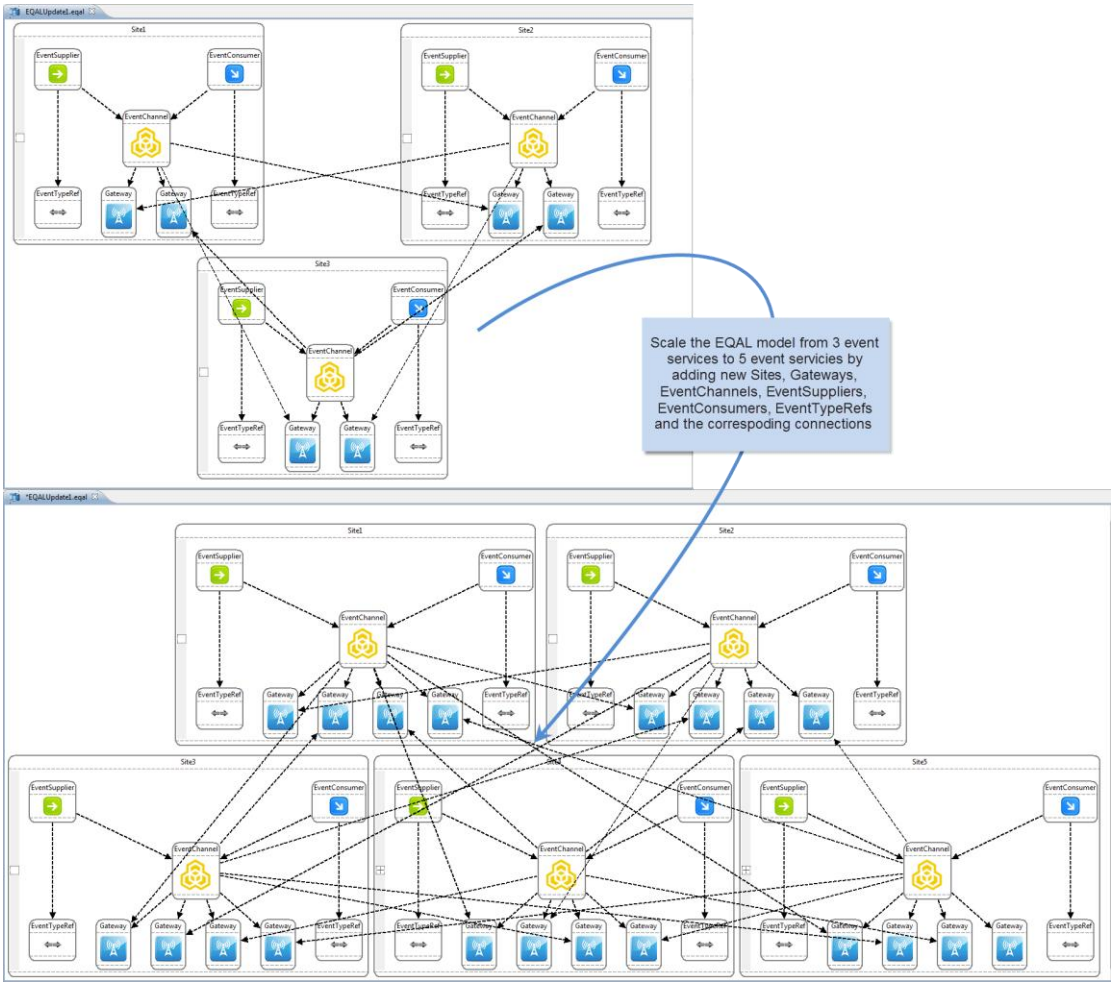


Figure 2. An EQAL model before (top) and after (bottom) scaling

2.3 Replicating Overloaded Application Nodes: Evolving Software Maintenance Models

Cloud computing shifts the computation from local, individual devices to distributed, virtual, and scalable resources, thereby enabling end-users to utilize the computation, storage, and other application resources on-demand [35]. A user can create, deploy, execute, and terminate the application instances in the cloud as needed, and pay for the cost of time and storage that the active instances use based on a utility cost model.

Cloud Computing Management Modeling Language (C2M2L) [31] is a DSML constructed specifically to describe the deployment of application nodes in the cloud and monitor 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 [36], containing four *Nodes – Web Tier Instance*, *Middle Tier Instance*, *Data Tier Instance* and *Load Balancer*. *NodeServices* 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*). 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.

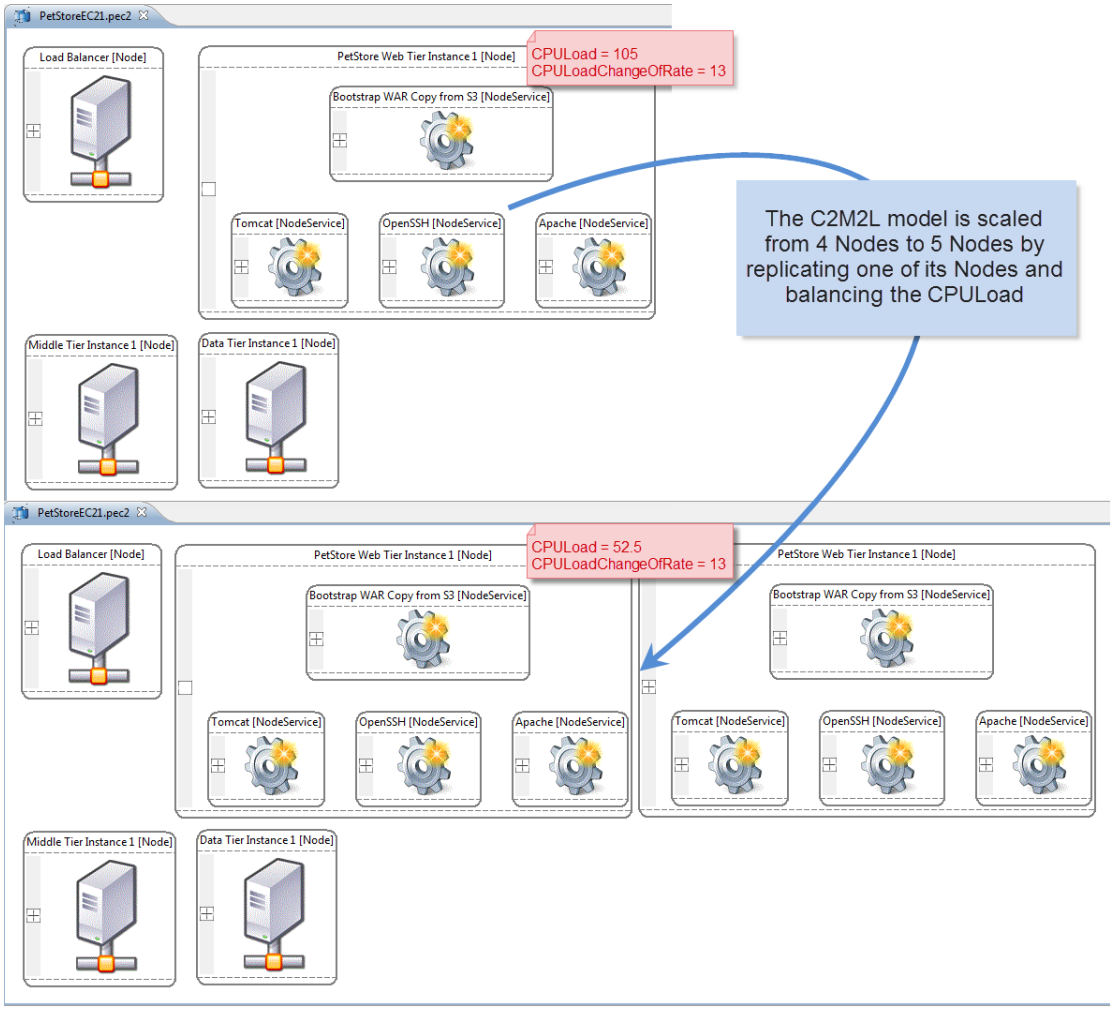


Figure 3. A C2M2L model before (top) and after (bottom) scaling

Scalability Scenario in C2M2L (Example 2.3): 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* and *CPULoadRateOfChange* of a certain *Node* are both out of the normal range, more *Nodes* containing the same *NodeServices* and configuration need to be replicated in order to balance the work load. As shown in the

bottom of Figure 3, one more *Web Tier Instance Node* is replicated to handle the increasing workload of the original single *Node*. To accomplish this task, creating the same *Node* and *NodeServices* are needed, as well as setting up all the properties to be the same as the previous *Node*, except balancing the *CPULoad* of both *Nodes*. In this scenario, the *CPULoad* and *CPULoadRateOfChange* properties must be checked before scaling, so that the new *Node* will be added only when the existing *Node* is really out of the normal range. This management task becomes challenging when a large number of application *Nodes* are running in the cloud. Automating the detection of overloaded *Nodes* and replicating them promptly is essential to ensure applications are running correctly and smoothly.

2.4 Challenges of Model Scalability Current Practice

For each of these model scalability scenarios, it is possible to edit the model manually to scale it from a simple state to another simple state (e.g., adding two new events in a 2-event SRN model, creating one event service for a 3-Site EQAL model, or replicating two new *Nodes* in a C2M2L model). However, it becomes extremely challenging to manually scale each of these scenarios from to a complex state when there are a large number of new elements that need to be added and connected. This challenge comes not only from the quantity of the required editing operations, but also the required accuracy and correctness, because a model scalability scenario often involves various types of error-prone activities: locating the correct part of a model to be scaled, creating proper elements and connections, precisely replicating elements and connections, setting up correct properties, and making accurate connections between existing elements and newly created ones. Some of these examples (i.e., from Sections 2.1 and 2.2) have been automated in the past using a model transformation language (MTL) [3][6]. However, the model end-users (e.g., domain experts, such as cloud computing administrators) might not have experience in using MTLs. A simpler approach is needed that assists general end-users in specifying model scalability scenarios.

3 Automating Model Scalability Using MTBD

In this section, we first give a brief introduction to the ideas behind Model Transformation By Demonstration (MTBD) (Section 3.1). Then, the key limitations that initially prevented MTBD from being applied to model scalability tasks are identified (Section 3.2), followed by the enhancements we have made to address these limitations (Section 3.3).

3.1 Overview of MTBD

MTBD is a model transformation approach motivated by the difficulties doman experts have with learning model transformation languages and understanding metamodel definitions. The basic idea (Figure 4) is that instead of manually writing transformation rules in a specific model transformation language, users demonstrate how a model transformation should be done by directly editing (e.g., add, delete, connect, update) a concrete model instance to simulate the desired model transformation process (i.e., User Demonstration). A recording and inference engine has been developed to capture all user operations performed during the demonstration (i.e., Operation Recording). After the recording process has completed, the engine optimizes the recorded operations (i.e., Operation Optimization) and infers a transformation pattern that specifies the precondition of the transformation and the sequence of actions needed to realize the transformation (i.e., Pattern Inference). This pattern can be reused by automatically matching the precondition in any model instance and replaying the actions to execute the intended model transformation (i.e., Pattern Execution). During the execution of a transformation pattern, constraint checking ensures that the execution does not violate the metamodel definition of the domain.

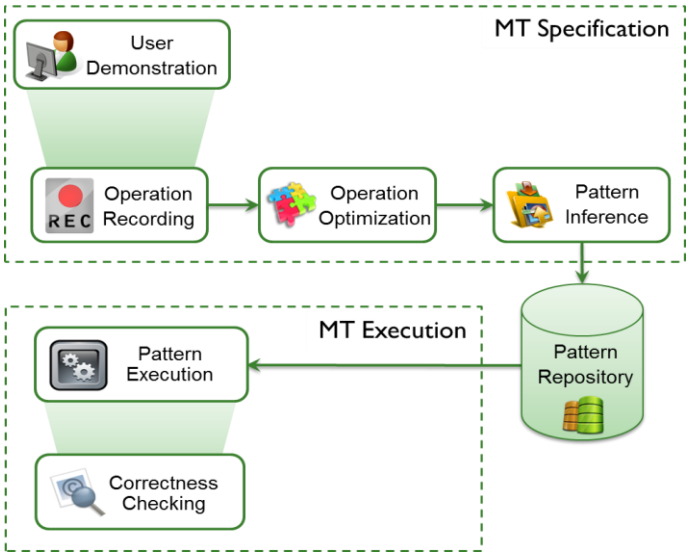


Figure 4. Overview of MTBD

The idea of MTBD has been implemented as an Eclipse plug-in for the Generic Eclipse Modeling System (GEMS) [19] called MT-Scribe. Without using any model transformation languages or the need to understand metamodels, general users are able to demonstrate endogenous model transformations and execute generated transformation patterns in a simple and automated manner. Similarly, this approach can be used to demonstrate how to scale models and infer corresponding patterns. This section presents a simple example based on

EQAL to illustrate the basic idea of using MTBD to support automated model scalability. More details about MTBD can be found in [10].

Assume that for each *Site* in an EQAL model, we desire to add one more *Gateway* (called `NewGateway`). To accomplish this task using MTBD, a user needs to demonstrate the scalability task by finding a single *Site*, adding a *Gateway* to it, followed by changing the name of the new *Gateway*. Operations in List 1 represent the user-demonstrated actions that are performed in the demonstration (a *Site* called *Site1* is selected in the demonstration, see Figure 5).

List 1. Operations performed in the demonstration

Sequence	Operation Performed
1	Add a *Gateway*in *EQALRoot.Site1*
2	Set *EQALRoot.Site1.Gateway.name* = "`NewGateway`"

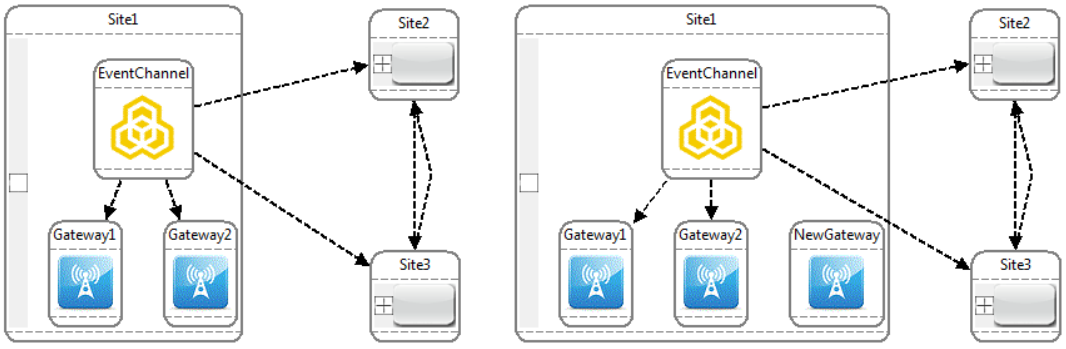


Figure 5. The EQAL model before (left) and after (right) the demonstration

After the demonstration is completed, a transformation pattern can be inferred. This pattern specifies the precondition (List 2 – any *Site* in the model root), and the transformation actions (List 3 – adding a new *Gateway* and changing its name). These lists are abstract representations of how the pattern is saved, which are invisible to end-users.

List 2. Precondition – elements needed and corresponding metatypes

Elements Needed for Operations
elem1.elem2
elem1.elem2.elem3

Element	MetaType
elem1	ModelRoot
elem2	Site
elem3	Gateway

List 3. Transformation actions

Sequence	Transformation Action
1	Add elem3 in elem1.elem2
2	Set elem1.elem2.elem3.name = "`NewGateway`"

A user may then apply this pattern to any other EQAL model. The engine will traverse the model and match the precondition using a back-tracking algorithm, after which the transformation actions will be executed. In this example, all the *Sites* in the model will automatically have a new *Gateway* added with the name being "`NewGateway`" (Figure 6 shows the pattern applied to an EQAL model containing six *Sites*).

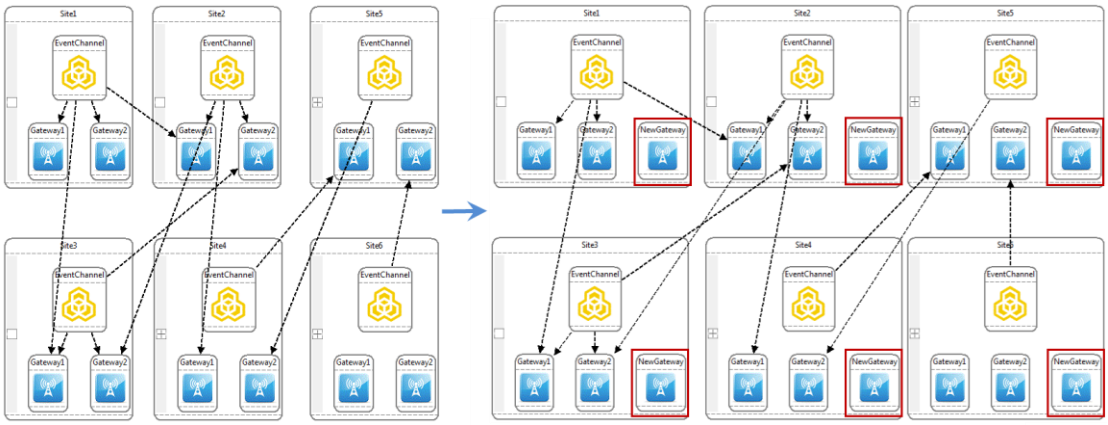


Figure 6. An EQAL model before (left) and after (right) applying the inferred scaling transformation pattern

3.2 MTBD Formal Description

Using the description of MTBD provided in Section 3.1, we can build a formal model of the process. Our formal model of MTBD is based on a 5-tuple:

$$Mtbd = \left\langle \vec{\Delta}_m, TG(M_i, \vec{\Delta}_m), \varpi(M_i, \vec{\Delta}_m), \phi(\vec{P}, T), \xi(M_j, \vec{P}', \vec{T}') \right\rangle \quad (1)$$

where:

1. M_i is a model conformant to the metamodel $Meta_i$
2. M_j is a model also conformant to the metamodel $Meta_i$
3. $\vec{\Delta}_m$ is a sequence of model modifications recorded during a user demonstration of a transformation on the model, M_i.
4. $TG(M_i, \vec{\Delta}_m)$ is a generalization function that produces a model transformation, T, that can be applied to any model conformant to $Meta_i$. The transformation is produced by generalizing the series of modifications, $\vec{\Delta}_m$, that were applied to M_i.
5. $\varpi(M_i, \vec{\Delta}_m)$ is an inference function that extracts the preconditions, $\vec{P}$, needed in order to generalize and apply the modifications, $\vec{\Delta}_m$, to another model.

6. $\phi(\vec{P},T)$ is an optional manual transformation and precondition refinement function that allows the domain expert to modify the transformation and preconditions inferred by *TG* and ϖ. This function produces a refined transformation, T', and set of preconditions $\vec{P}'$.

7. $\xi(M_j,\vec{P}',\vec{T}')$ is a transformation function that applies the refined generalized transformation, T', to a model, M_j, if the refined preconditions $\vec{P}'$ are met by M_j.

3.2.1 Operation Recording

The goal of MTBD is to allow users to express domain knowledge regarding a function, $K(M_i)$. That is, the user is describing a domain-specific function that can be applied to a model in order to achieve a domain-specific goal. For example, the EQAL example in Section 2.2 captured a domain function that expressed how to scale up a publisher/subscriber model by adding *Sites*, *EventChannels*, and *Gateways*. A critical component of MTBD is that the domain function (transformation) is expressed in terms of the notations in the modeling language and not the notations used to describe the metamodel, $Meta_i$.

MTBD captures domain functions as transformations that can be applied to models that adhere to the metamodel, $Meta_i$, of the target domain. The first step in MTBD is for a user to apply the domain function, $K(M_i)$ to a model, so that the MTBD engine can capture the set of model modifications, $\vec{\Delta}_m$. The process begins by the user or an external signal initiating a recording process. During the recording process, the user applies the domain function, $K(M_i)$, to the model, M_i:

$$K(M_i) \Rightarrow M_j \tag{2}$$

$$K : Meta_i \rightarrow Meta_i \tag{3}$$

The domain function takes an initial model, M_i, as input, and produces a new model, M_j, as output. Although it is possible that M_i and M_j are not conformant to the same metamodel, $Meta_i$, this paper explicitly focuses and enforces this assumption. Equation 3 shows that the domain function must represent an endogenous model transformation that maps a model in one metamodel domain to a model in the same metamodel domain.

3.2.2 Pattern Inference

After the recording process, the MTBD engine possesses a series of model modifications, $\vec{\Delta}_m$ that express the application of the domain function, $K(M_i)$ to a specific model. The next

step of MTBD is to use pattern inference to generalize and describe the domain function as a model transformation. A critical aspect of this process is that the transformation must be expressed in terms of the general metamodel notations captured in, $Meta_i$, rather than a specific model's elements, M_i. The inference step produces a model transformation, which we describe as a tuple:

$$Transformation = \langle \vec{P}, T \rangle \quad (4)$$

where $\vec{P}$ is a precondition that must be met in order to apply the transformation, and T is the set of generalized model modifications that transform the source model to the desired target model. In terms of the domain function, $\vec{P}$ describes the domain knowledge regarding the circumstances in which $K(M_i)$ can be applied and T defines what to do when these circumstances are met. For example, in the cloud computing example from Section 2.3, $\vec{P}$ is the precondition that the rate of change of CPU load is above a set threshold and T represents the modifications to the system needed in order to scale up the number of virtual machine instances to handle the load.

3.2.2.1 Precondition Inference

The preconditions can be subdivided into two types:

1. **Structural preconditions** that govern the types of elements, the containment relationships, and connection relationships that must exist within the model. For example, in the EQAL motivating scenario, there must be an element of type *Site* contained within an element of type *Root*.
2. **Attribute preconditions** specify the required values of attributes on the model elements. For example, in the cloud computing scenario, the *CPULoadRateOfChange* attribute of a *Node* element must be above a specified threshold.

Structural Preconditions. The structural preconditions take the form of assertions on the hierarchy or connection relationships that must be present in the model. A hierarchical precondition, Pe_i, is described as a vector:

$$Pe_i = T_0, T_1, \ldots, T_n \quad (5)$$

where T_0 is the type of an element that is directly modified by one or more operations in $\vec{\Delta}_m$, T_1 is the type of the parent of T_0, T_2 is the type of the parent of T_1, and so forth to the root element. In order for this precondition to hold in an arbitrary model, M_j, an instance of the type T_0, contained within an element of type T_1, must exist. More formally, given an element, e_i, in a model M_j that conforms to the metamodel $Meta_i$, a hierarchical precondition, Pe_i, is

satisfied by e_i if:

$$V(e_i, T_i) = (type(e_i) == T_i) \wedge (V(e_{i+1}, T_{i+1})) \quad (6)$$

$$Pe_i(e_i) = \begin{cases} (V(e_i, T_0) == true), true \\ otherwise, false \end{cases} \quad (7)$$

A connection precondition is another form of a structural precondition. Connection preconditions dictate the associations that must be present in the model. For example, in the EQAL example, a transformation that removes a connection between two *Sites* must have a precondition that there exist two *Sites* that are connected. A connection precondition, Pc_i, is defined as a 3-tuple:

$$Pc_i = < Pe_j, Pe_k, T_l > \quad (8)$$

where Pe_j specifies a structural precondition that must be met for an element to be considered the source element of a connection to be modified; Pe_k is a precondition that must be met for an element to be considered the target element of the connection; and T_l is the type of connection that must exist between the elements that satisfy the source and target structural preconditions. In order for a connection, c_i, between two elements, e_i and e_j, to satisfy Pc_i:

$$Pc_i = \begin{cases} Pe_j(e_i) \wedge Pe_k(e_j) \wedge (type(c_i) == T_l), true \\ Pe_j(e_j) \wedge Pe_k(e_i) \wedge (type(c_i) == T_l), true \\ otherwise, false \end{cases} \quad (9)$$

The inference $\varpi(M_i, \vec{\Delta}_m)$ function evaluates each change in Δ_m that occurred. From these changes, structural preconditions are extracted as follows:

- **Added Elements.** For each model element, e_j, that is added to the model as a child of e_i, a precondition, Pe_i, is created. The type vector for Pe_i captures the types of elements that are visited from traversing from e_i to the root of the model. T_0 is set to the type of e_i.
- **Removed Elements.** If an element, e_i, is removed from the model, a precondition, Pe_k, is created. The type vector for Pe_k captures the types of elements that are visited from traversing from e_i to the root of the model. T_0 is set to the type of e_i.
- **Added Connections.** Each new connection, c_j, that is added from model element e_i to e_j, produces a new precondition, Pc_i. The type vector for the source element, Pe_j, captures the types of elements that are visited from traversing from the source element to the root of the model. The type vector for the target element, Pe_k, captures the types of elements that are visited from traversing from the target

element to the root of the model. T_l is set to *0* to indicate that no existing connection is required between the elements that satisfy Pe_j and Pe_k.

- **Removed Connections.** Each deleted connection, c_j, that previously started from model element e_i and ended at model element e_j, produces a new precondition, Pc_i. The type vector for the source element, Pe_j captures the types of elements that are visited from traversing from the source element to the root of the model. The type vector for the target element, Pe_k, captures the types of elements that are visited from traversing from the target element to the root of the model. T_l is set to the type of c_j.
- **Changed Attributes.** If an element, e_i, has an attribute value changed, a pre-condition, Pe_k, is created. The type vector for Pe_k captures the types of elements that are visited from traversing from e_i to the root of the model. T_0 is set to the type of e_i.

Attribute Preconditions. Attribute preconditions specify the required values of properties on elements that a transformation will be applied to. The attribute preconditions, *Ac*, are specified as tuples:

$$Ac_i =< Pe_i, Expr > \quad (10)$$

where Pe_i is a structural precondition specifying the source model element to which the attribute precondition must be checked. The *Expr* component specifies a mathematical expression over the attributes of an element that satisfies Pe_i. Currently, the attribute must be a primitive value and only arithmetic primitives (e.g., addition, multiplication, division, and subtraction) are supported.

Attribute preconditions are difficult to infer automatically. Simple algorithms can extract preconditions that specify an exact value of one or more element attributes. However, these algorithms are often too exclusive and generate preconditions that require exact matching of all attribute values. Ideally, attribute preconditions are specified as expressions from domain knowledge covering the affected elements. Manual inference refinement is used to capture this type of attribute precondition.

3.2.3 Manual Inference Refinement

The goal of MTBD is to generate a transformation, *T*, that faithfully represents the domain function $K(M_i)$. However, in many circumstances, the model that the function is demonstrated on, M_i, may lack sufficient information to infer preconditions accurately. For example, in the cloud computing example from Section 2.3, the cloud computing model does not have any

information related to the CPU rate of change threshold at which scaling should occur. In this type of situation, the domain expert must be able to refine the inferred preconditions, by providing a CPU rate of change threshold value, in order to ensure that *T* accurately captures $K(M_i)$. The optional manual inference function, $\phi(\vec{P},T)$, allows the user to view the inferred transformation and preconditions produced by *TG* and ϖ. The following section describes in detail the need for a manual refinement step.

3.3 Limitations of Original MTBD to Support Model Scalability

Although the example in Section 3.1 is simple, it shows the potential for assisting general end-users in using MTBD to automate model scalability. However, this example is too simple to illustrate its real practicality. In fact, some key limitations existed in our previous implementation of MT-Scribe that prevented the MTBD concepts from being applied to complex model scalability tasks in practice.

Specific and restricted specification of preconditions. To scale a model, a precise precondition is needed to specify exactly where to execute the model transformation. However, in the original implementation of MT-Scribe only the weakest precondition can be inferred from the demonstration, such that there was no way for the end-user to provide more restricted conditions. A model satisfying the weakest precondition is defined as the model containing the minimum sufficient elements for each operation to be correctly executed. In the previous example, the precondition inferred (List 2) is that a *Site* must exist in the *Root*, so that a *Gateway* can be added in this *Site*, and the name of the new *Gateway* can be updated later.

The weakest precondition is insufficient in practice. In many cases, more specific restrictions are often required to provide more control on where to scale a model precisely. For example, users may want to add the new *Gateway* in the *Site* only if a certain attribute of the *Site* satisfies a specific condition (e.g., *Site.capacity >= 100* as shown in Figure 7); or users may want to add the new *Gateway* only if the *Site* has no outgoing and incoming connections from it. These kinds of specific precondition requirements are frequently needed in model scalability tasks. Scaling a model by adding or replicating model elements or connections often requires the end-user to select specific locations to scale, rather than simply enlarging all the places that could fit and execute the recorded operation in a demonstration (e.g., Example 2.3 requires the creation of new *Nodes* only when the *CPULoad* and *CPULoadRateOfChange* are both out of the normal range). Therefore, enabling users to specify more restricted and specific preconditions was the first need for extending MT-Scribe.

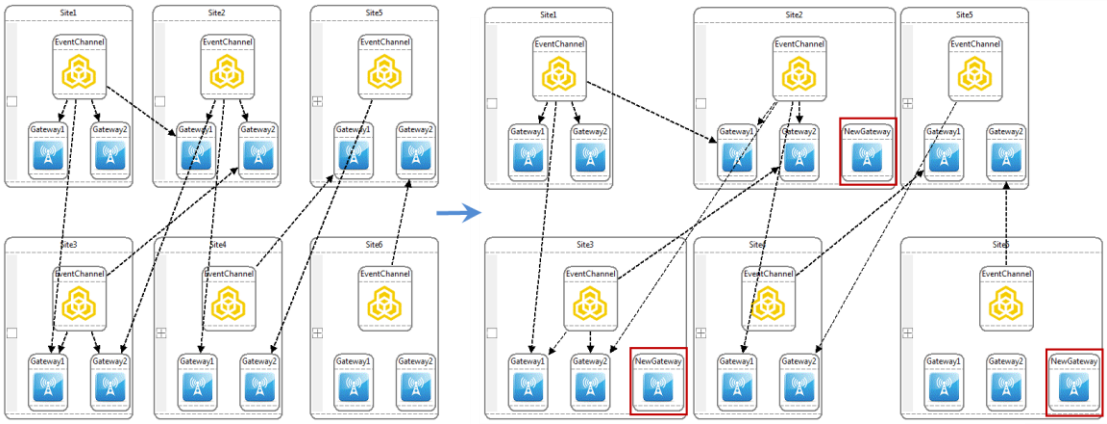


Figure 7. Scaling specific locations based on preconditions

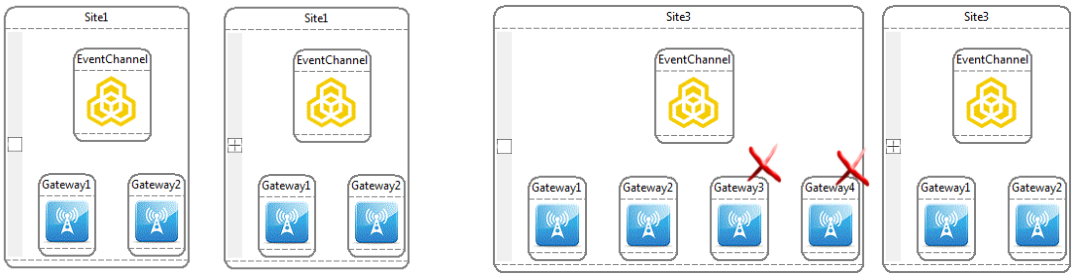
The inferred transformation actions are not generic. Besides the precondition, another part of the inferred transformation pattern is the list of transformation actions, which are extracted from the recorded operations. However, the inferred actions are specific to a user's demonstration, which means that the sequence, the number and the type of inferred actions are exactly the same as the recorded operations. The consequence is that it is not generic enough to reflect a user's real transformation intention. For instance, a user may want to replicate a *Site* (e.g., *Site1* in Figure 8a) that contains an *EventChannel* and two *Gateway*s. This would require that the operations in List 4 be performed in the demonstration.

List 4. Operations performed to replicate a *Site*

Sequence	Operation Performed
1	Add a *Site* in *EQALRoot*
2	Set *Site.name = EQALRoot.Site1.name*
3	Add an *EventChannel* in the new *Site*
4	Set *EventChannel.name = EQALRoot.Site1.EventChannel.name*
5	Add a *Gateway* in the new *Site*
6	Set *Gateway.name = EQALRoot.Site1.Gateway1.name*
7	Add a *Gateway* in the new *Site*
8	Set *Gateway.name = EQALRoot.Site1.Gateway2.name*

The real intention of this demonstration is to make an exact copy of *Site1*, including all the elements contained. However, the inferred transformation pattern only works correctly if the *Site* to be replicated contains exactly the same number of elements as the *Site* in the demonstration – one *EventChannel* and two *Gateways*. If there are more than two *Gateways* (e.g., *Site3* in the left of Figure 8b), only two of them (i.e., *Gateway1* and *Gateway2*) will be replicated, and *Gateway3* and *Gateway4* will not be copied (e.g., *Site3* in the right of Figure

8b is the *Site* created after executing the inferred replication pattern), because in the demonstration, the user only performed the necessary operations to add two *Gateways*, although his or her real intention was to copy all the available *Gateways*. If there are less than two *Gateways* (e.g., one *Gateway1* is in the *Site*), the pattern will also fail to replicate the *Site*, because this *Site* does not satisfy the weakest precondition due to a lack of sufficient *Gateways* to execute the two replicating operations in the demonstration.



a. Replicate a *Site* in a demonstration

b. The inferred pattern failed to replicate all *Gateways*

Figure 8. The inferred transformation actions are not generic

The inability to infer generic actions may lead to a major problem when dealing with model scalability tasks. The number of specific elements or connections varies frequently in different scaling situations and the number will usually increase after each scaling process (e.g., Example 2.1 requires the creation of transitions between the new snapshot place and each of the existing snapshots, but the number of existing snapshots varies). Because of this, a specific and non-generic inferred transformation obviously cannot handle each scenario readily. Therefore, we needed to extend MT-Scribe to enable the inference of more generic transformation actions.

More diverse options are required in attribute transformation. Enabling attribute transformation (e.g., transforming a specific attribute from one value to another value through arithmetic or string computations) in a user-friendly manner is an important innovation in MTBD. However, only simple computations such as basic arithmetic (i.e., +, -, *, /) and string concatenation were supported in earlier versions of MT-Scribe. To perform model scalability tasks, other operations are needed. For instance, the name of a certain element should be constructed based on a substring of the name of another element in the base model (e.g., Example 2.1 requires the creation of a new of snapshot transition by combing the names of the source and target snapshot places, such as *TProcSnp1,3*). However, obtaining the substring was not possible in previous versions of MT-Scribe. In other cases, the value of a certain attribute should be decided from the user's input (e.g., Example 2.1 requires the name of the new event to be obtained by the end-user), which is independent of any attributes existing in the model. This required the addition of interactive user input to MT-Scribe.

More options are needed to control the execution of transformation patterns. In the original version of our tool, when applying a generated transformation pattern, only a single pattern could be selected to execute only once. However, in the context of model scalability, scaling a base model to a complex model requires repeated execution of a transformation to avoid manual execution of the transformation multiple times. Additionally, to handle complex scalability requirements, more than one transformation pattern is needed to work in sequence to achieve the desired result. Therefore, users should be able to select and execute multiple patterns together in a composed pipeline sequence, realizing the execution of a transformation chain.

3.4 New Extensions and Features to MTBD

To address these limitations in the previous version of MT-Scribe, and adapt it to model scalability requirements, several new features and extensions have been made.

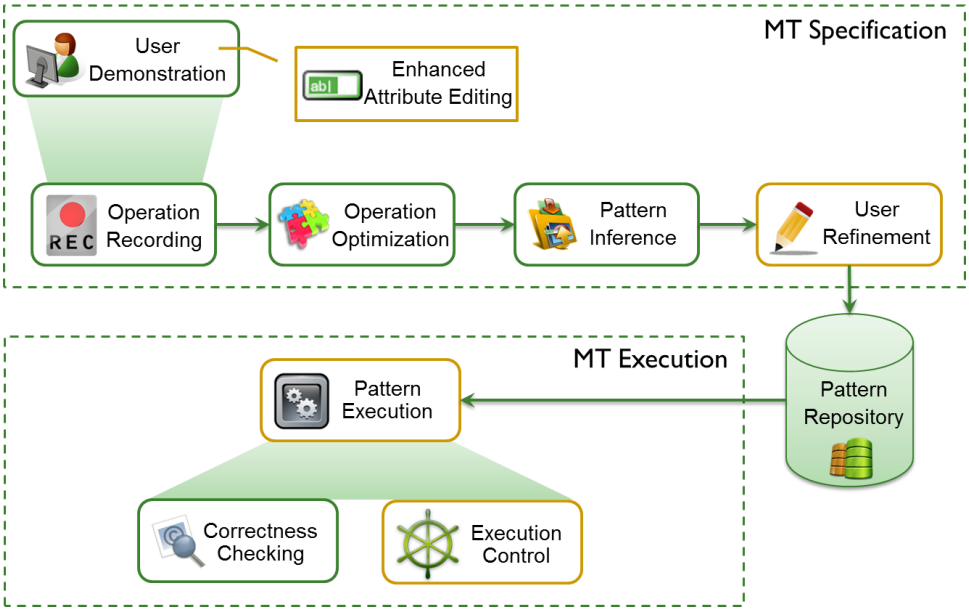


Figure 9. The overview of extended MTBD

A user-refinement step to specify preconditions. Inferring the specific preconditions from only the demonstration is difficult and inaccurate, because the performed operations only reflect the actions with very limited information about the precondition. Therefore, additional feedback should be given by users so that the engine can refine the generated pattern. In order to maintain the simplicity of MTBD, a user-friendly interface has been implemented to enable user selection of a specific element and specification of the desirable preconditions, without having to know model transformation languages or metamodel definitions.

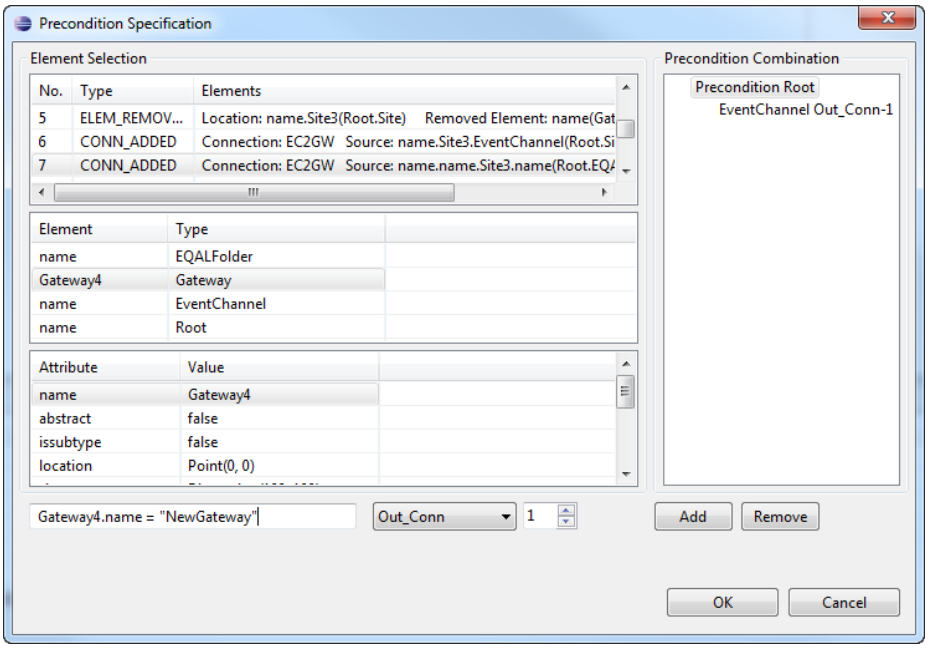


Figure 10. Precondition specification dialog

Figure 10 shows the precondition specification dialog. The upper-left lists all the recorded operations in the demonstration. By clicking on a specific operation, all the model elements involved will be listed, so that a user can easily find the elements for which they want to provide more constraints. Similarly, by clicking on a certain element, all its attributes and associated values are listed. Users can select certain attributes and type the necessary restrictions. For example, the following additions could be made: "*Site1.capacity >= 100*", "*Site1.capacity == Site2.capacity == Site3.capacity*", "*Node1.CPULoad > 80 && Node1.CPULoadChangeOfRate > 10*". Also, constraints can be given on the attributes that are not defined in the metamodel, such as the number of outgoing or incoming connections. Through this interface, users continue to work at the model instance level to give specific preconditions on the elements they considered in the demonstration. The meta-information and generic computation will be inferred and stored in the transformation pattern automatically.

A new user-refinement step to identify generic operations. From the *Site* replication example in Section 3.1, it can be observed that the reason an inferred transformation pattern does not work correctly for the *Sites* containing more than two *Gateways* is that the inferred actions are specific to the user's demonstration, failing to reflect the user's real intention (i.e., copying all *Gateways*, no matter how many there are). However, from List 4, we can see that operations (5, 6) and (7, 8) have exactly the same meaning and the same purpose (i.e., adding a new *Gateway* in the new *Site* and setting its name to be the name of an existing *Gateway* being copied). In fact, only one set is enough, and we can just repeat their execution

according to the number of available *Gateways* in the *Site* being copied. Therefore, to solve the problem, we implemented the idea that if certain operation(s) needs to be generic (i.e., needs to be executed or repeated for different times according to the number of available elements), a demonstration is only needed to be done once, followed by clearly identifying the operation(s) as generic or repeatable.

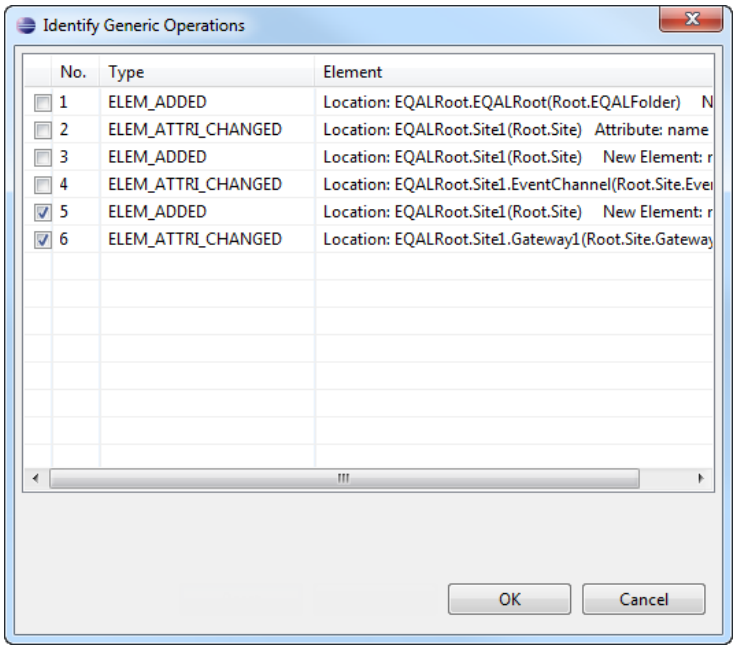


Figure 11. Generic operations identification dialog

Figure 11 shows the generic operations identification dialog. It simply lists all the operations performed during the demonstration process. Users may identify the generic operation(s) by selecting the checkbox. The new MT-Scribe inference engine will then mark the operations accordingly, and repeat them in the pattern execution according to the specific model instance. For example, to solve the problem in Example 2.2, instead of performing the operations listed in List 4, the user should do as specified in List 5.

List 5. Demonstrate generic operations only once

Sequence	**Operation Performed**
1	Add a *Site* in *EQALRoot*
2	Set *Site.name = EQALRoot.Site1.name*
3	Add an *EventChannel* in the new *Site*
4	Set *EventChannel.name = EQALRoot.Site1.EventChannel.name*
5	Add a *Gateway* in the new *Site*
6	Set *Gateway.name = EQALRoot.Site1.Gateway1.name*

After performing the above demonstration, the user must then mark operations 5 and 6 as generic (as shown in Figure 11). This sequence of demonstration and operation revision actions will generate a generic transformation pattern that is capable of replicating any *Sites* correctly, independent of the number of *Gateways*. With these enhancements, users still work at the model instance level when demonstrating the generic operations.

An enhanced attribute refactoring editor. In the earlier version of MT-Scribe, we implemented an attribute refactoring editor, which allowed users access to all the attributes existing in the current model instance. Through this editor, users could calculate the needed attributes through arithmetic or string computations during the demonstration (e.g., users could just click on a certain attribute, retrieve the value, type the computation, and calculate the new value). All of the meta-information and computation details are stored in the inferred transformation. For instance, to set the *capacity* of the new *Site* to be 2 times the capacity of *Site1* being copied, the user just clicks on the *capacity* of the *Site* being copied, and retrieves its current value (e.g., *100*). Then, the user can type "*/2*" and click on "Calculate," the result being that *50* is displayed and assigned as the *capacity* of the new *Site* while the computation "*NewSite.capacity = Site1.capacity / 2*" is stored in the transformation pattern.

In order to enhance the attribute editor, new functions have been added. More diverse expressions, such as `subString()`, `indexOf()` can be used to specify the computation. In the new implementation of MT-Scribe, we applied the dynamic language Groovy [12] to parse and calculate the expressions. All of the Java expressions and functions supported by Groovy may be used in the attribute computation.

Moreover, user input is also enabled in the attribute editing process. If a certain value is independent of any existing attributes and should be input by users, they can create a name and give its value in the demonstration, indicating that this is an input value, which is then visible in the rest of the demonstration. Later, when executing this pattern, the inference engine will automatically prompt an input box to ask the user to specify the value of this name. Thus, with the enhanced attribute refactoring editor, users have more options to specify and edit the attribute transformation in the demonstration of the scalability process.

An enhanced pattern execution controller. Users can select the pattern in the dialog to execute an inferred transformation pattern from the repository. With the enhanced execution dialog, not only the selection of multiple patterns at the same time is enabled, but also the total times for executing a selected pattern(s) can be specified. The benefit is that users can separate a complex scalability task into several subtasks, and generate several patterns, then execute them all together in sequence. The model can then be scaled by executing the patterns

for any number of times desired. In the next section, we illustrate how these new features work together to address the three model scalability examples presented in Section 2.

4 Automated Model Scalability Case Studies

In this section, we show how the concepts of MTBD can address the needs of the three motivating examples presented in Section 2. To minimize the effort of performing a scalability demonstration, we focus on a base model with a small number of elements, and demonstrate how to scale it by one degree (e.g., scale a SRN model from two events to three events; scale an EQAL model from three event services to four services). Then, by executing the inferred transformation pattern for any number of times, the model can be scaled to the desired state. In other words, the guidance of the approach can be summarized as "demonstrate one, scale multiple times."

Given a model scalability task, the main steps of a solution are: 1) analyze the process of scaling the model by one degree, so that the minimum and generalized operations needed by the scaling scenario can be clearly identified; 2) perform the demonstration of scaling the model by one degree; 3) specify preconditions and identify generic operations in the user refinement step; 4) scale the model to the desired state by executing the generated pattern for the desired number of times.

The remainder of this section provides solutions to the scalability examples of Section 2 using the new additions to MT-Scribe. Video demonstrations of each of the examples in this section are available at the MT-Scribe web page [22].

4.1 Scaling SRN Models

By analyzing the scalability needs of Example 2.1, the task of adding one more event type to an existing SRN model can be divided into the following three, sub-tasks as shown in Figure 12:

t1. 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.
t2. Create the *TStSnp* and *TEnSnp* snapshot transitions and the *SnpInProg* snapshot place, as well as the associated connections.
t3. 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 the top of Figure 1. Then, we manually edit the model and demonstrate the three sub-tasks. To demonstrate t1, the operations identified in List 6 are performed.

List 6. Operations for sub-task t1 of Example 2.1

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*

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 (e.g., the new event is called "3", so the places and transitions are named as "A3", "B3", "Sn3", etc.). The operation also indicates that the value of this name should be given by the user, which will invoke to an input box when the final generated transformation pattern is executed on other model instances. 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 top model will have a new event type, as shown in Figure 12 (Sub-task 1).

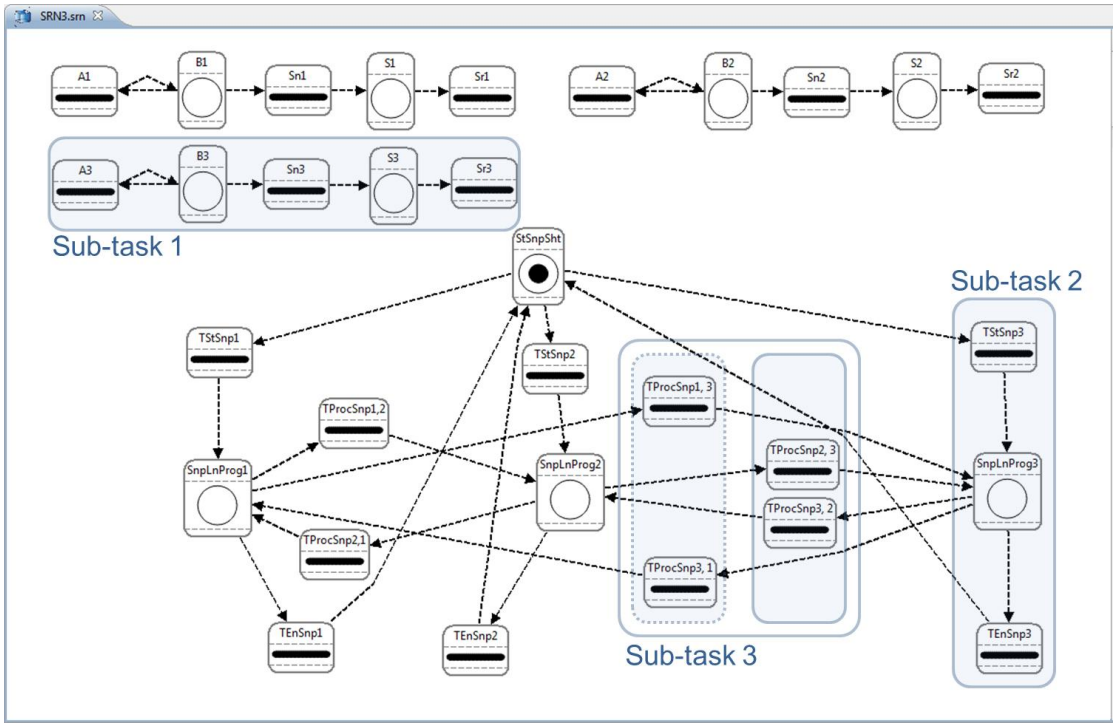


Figure 12. The process of scaling a SRN model from two events to three events

To demonstrate t2, the necessary snapshot places and transitions in sub-task 2 are added for the new event type by performing the operations indicated in List 7. Figure 12 (Sub-task t2) shows the model after these operations.

List 7. Operations for sub-task t2 of Example 2.1

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 t3, two snapshot transitions for each <existing snapshot place, new snapshot place> are created. This sub-task involves using generic operations mentioned in Section 3.3, because the number of existing snapshot places may vary in different model instances. This number will also increase after each scaling process. Therefore, in the demonstration, users only need to create two snapshot transitions for just one set of <existing snapshot place, new snapshot place>, followed by identifying these operations as generic after the demonstration, so that the engine will generate the correct transformation pattern to repeat these operations when needed. The operations performed are shown in List 8. We select *SnpLnProg2* as the existing snapshot place, and demonstrate the creation of snapshot transitions *TProcSnp2,3* and *TProcSnp3, 2*.

List 8. Operations for sub-task 3 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 (i.e., checking the generic operations in the dialog as shown in Figure 11), the inference engine automatically infers and generates the transformation pattern. After the inferred transformation 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. An execution controller has been implemented to enable execution of a pattern multiple times. The bottom of Figure 1 is the result of adding two event types using the inferred pattern.

4.2 Scaling the EQAL Models

Example 2.2 focuses on increasing event services. The scaling process of adding one more event service includes four sub-tasks, as illustrated in Figure 13:

t1. Add a new *Gateway* to each original *Site*, and connect it to its *EventChannel*.

t2. Add a new *Site*, containing an *EventChannel*, *EventSupplier*, *EventConsumer*, *EventTypeRefs*, *Gateways*, and necessary connections.

t3. Make connections from the *EventChannel* in the new *Site* to each new *Gateway* in other *Sites*.

t4. Make connections from the *EvenChannel* in each original *Site* to a new *Gateway* in the new *Site*.

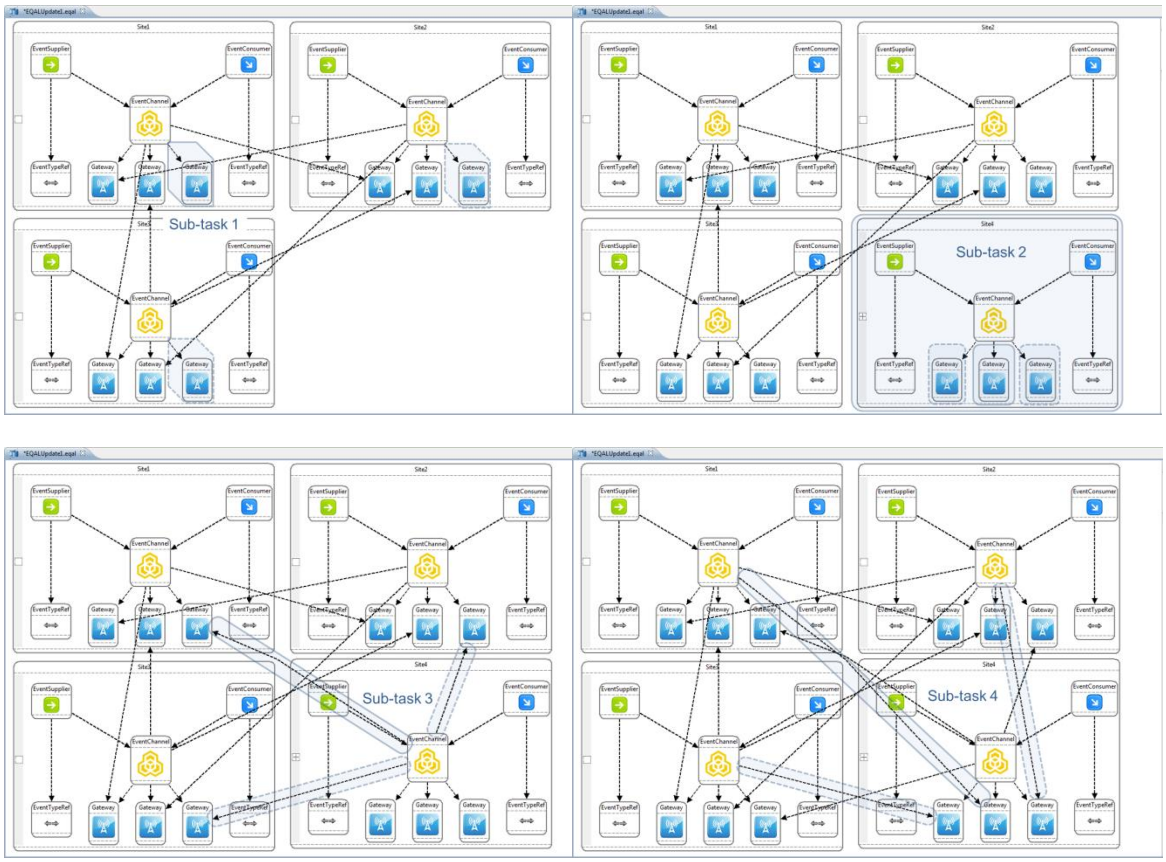


Figure 13. The process of scaling an EQAL model from three event services to four

We give the demonstration on the model instance shown in the top of Figure 2. The first sub-task t1 is to add a new *Gateway* to each original *Site* and connect it to the *EventChannel*. Obviously, this is another case of generic operations, because the number of current existing *Sites* is unfixed (e.g., there are three *Sites* in this case, but there could be more or less in other models). Therefore, in the demonstration, we only demonstrate adding one *Gateway* and making the connection in one of the *Sites*, and then, identify them as generic. The operations performed for t1 are shown in List 9.

List 9. Operation for sub-task t1 of Example 2.2

Sequence	Operation Performed
1*	Add a *Gateway* in *EQALRoot.Site1*
2*	Connect *EQALRoot.Site1.EventChannel* to *EQALRoot.Site1.Gateway*

To demonstrate t2, we need to create a new *Site*. Again, adding one *Site* and its *EventChannel*, *EventSuppiler*, *EventConsumer*, and *EventTypeRefs* are only needed once for each scaling process, while adding new *Gateways* and connecting them to the *EventChannel* in the new *Site* should be generic and correspond to the number of existing *Gateways* in the original *Site*s. List 10 shows the operations performed to add a new *Site* and its internal structure.

List 10. Operations for sub-task 2 of Example 2.2

Sequence	Operation Performed
3	Add a *Site* in *EQALRoot*
4	Add an *EventChannel* in *EQALRoot.Site*
5	Add a *EventSupplier* in *EQALRoot.Site*
6	Add a *EventConsumer* in *EQALRoot.Site*
7	Add a *EventTypeRef* in *EQALRoot.Site*
8	Add a *EventTypeRef* in *EQALRoot.Site*
9	Connect *EQALRoot.Site.EventSupplier* to *EQALRoot.Site.EventTypeRef*
10	Connect *EQALRoot.Site.EventConsumer* to *EQALRoot.Site.EventTypeRef*
11	Connect *EQALRoot.Site.EventSupplier* to *EQALRoot.Site.EventChannel*
12	Connect *EQALRoot.Site.EventConsumer* to *EQALRoot.Site.EventChannel*
13*	Add a *Gateway* in *EQALRoot.Site*
14*	Connect *EQALRoot.Site.EventChannel* to *EQALRoot.Site.Gateway*

To demonstrate t3, multiple connections have to be made to connect the new *EventChannel* in the new *Site* to each new *Gateway* in the other *Sites*. In this step, a user should not only demonstrate a single generic connecting operation, but also give additional constraints on the source and target elements of this connection, because there are so many *EventChannels* and *Gateways* available (List 11). Without a user's restriction, the inference engine may choose the wrong source and target to make the connection when the pattern is executed. The precondition on the source *EventChannel* is that it initially has no outgoing and incoming connections, because it is a newly created *EventChannel* in the new *Site*. The extra precondition on the target *Gateway* is that it has only one outgoing and no incoming connections.

List 11. Operations for sub-task t3 of Example 2.2 ("p" represents the precondition)

Sequence	Operation Performed
15*	Connect *EQALRoot.Site.EventChannel* and *EQALRoot.Site1.Gateway* p1 *EQALRoot.Site.EventChannel.outgoingConns* = 0 p2 *EQALRoot.Site.EventChannel.incomingConns* = 0 p3 *EQALRoot.Site1.Gateway.outgoingConns* = 0 p4 *EQALRoot.Site1.Gateway.incomingConns* = 1

The final sub-task t4 is to connect each original *EventChannel* to a new *Gateway* in the new *Site*. Again, the *Gateway* in the new *Site* should have only one incoming connection from its own *EventChannel* (List 12).

List 12. Operations for sub-task t4 of Example 2.2

Sequence	Operation Performed
16*	Connect *EQALRoot.Site1.EventChannel* and *EQALRoot.Site.Gateway* p1 *EQALRoot.Site.Gateway.outgoingConns* = 0 p2 *EQALRoot.Site.Gateway.incomingConns* = 1

After the demonstration, generic operations are identified, and preconditions are given in the user refinement step through the interface shown in Figure 10. Precondition specification dialog. Users give the preconditions to the elements he or she just touched in the demonstration, without being exposed to any metamodel information. The model can be scaled by adding any number of new *Sites* by applying the pattern multiple times. The bottom of Figure 2 is the result of applying the inferred pattern to scale the model by adding three new event services.

4.3 Scaling the C2M2L Models

Replicating a *Node* in Example 2.3 includes two sub-tasks:

t1. Replicate the overloaded *Node*, and balance the *CPULoad* by setting the *CPULoad* attribute for both the new *Node* and the original *Node*.
t2. Replicate all the *NodeServices* contained in the original *Node* to the new *Node*.

Replicating a model element involves creating the same type of element and setting up the same attribute values. To demonstrate t1, we can create one *Node* and set all its attributes to be the same as those in the original overloaded *Node*, except *CPULoad* (List 13). An important set of preconditions should be specified to ensure that the Node is actually

overloaded, which in this case occurs when the *CPULoad* is greater than 100 and *CPULoadRateOfChange* is greater than 10.

List 13. Operations for sub-task t1 of Example 2.3

Sequence	Operation Performed
1	Add a *Node* in *C2M2LRoot*
2	Set *Node.Name = PetStoreWebTierInstance1.Name = "PetStoreWebTierInstance1"*
3	Set *Node.AMI = PetStoreWebTierInstance1.AMI = "ami-45e7002c"*
4	Set *Node.Annotation = PetStoreWebTierInstance1.Annotation = "WebTier for PetStore"*
5	Set *Node.HeartbeatURI = PetStoreWebTierInstance1.HeartbeatURI = "http://ps01.aws.amazon.com/hb"*
6	Set *Node.HostName = PetStoreWebTierInstance1.HostName = "http://ps01.aws.amazon.com/hb"*
7	Set *Node.IsWorking = PetStoreWebTierInstance1.IsWorking = true*
8	Set *PetStoreWebTierInstance1.CPULoad (old) = PetStoreWebTierInstance1.CPULoad (old) / 2 = 105 / 2 = 52.5*
9	Set *PetStoreWebTierInstance1.CPULoad (new) = PetStoreWebTierInstance1.CPULoad (old) = 52.5* ^{p1}*PetStoreWebTierInstance1.CPULoad (old)> 100* ^{p2}*PetStoreWebTierInstance1.CPULoadRateOfChange (old)> 10*

To demonstrate t2, because the number of *NodeServices* in a *Node* varies, the operations of replicating the *NodeService* should be generic and demonstrated only once (List 14).

List 14. Operations for sub-task t2 of Example 2.3

Sequence	Operation Performed
10*	Add a *NodeService* in the new *C2M2LRoot.PetStoreWebTierInstance1*
11*	Set *NodeService.Name = PetStoreWebTierInstance1.BootstrapWarCopyFromS3.Name = "BootstrapWarCopyFromS3"*
12*	Set *NodeService.ResponseTime = PetStoreWebTierInstance1.BootstrapWarCopyFromS3. ResponseTime =0.14*
13*	Set *NodeService.ResponseTimeRateOfChange = PetStoreWebTierInstance1.BootstrapWarCopyFromS3. ResponseTimeRateOfChange = 0.001*

The generated pattern can be executed by cloud computing administrators to automatically detect the overloaded *Nodes* and replicate the necessary number of new ones.

5 Evaluation

In this section, the MTBD approach is first evaluated using the three desired characteristics of model scalability proposed by [3] and [6]. Then, we compare scalability scenarios using traditional model transformation languages to that of the MTBD approach advocated in this paper. Finally, the main limitations of MTBD are discussed at the end of this section.

5.1 Evaluation on the Desired Characteristics of a Replication Approach

This section compares the scalability solution offered by MTBD to a set of proposed desiderata described in [3] and [6].

Retain the benefits of modeling. The power of modeling comes from the ability to explore various design alternatives and perform system analysis or development at a higher level of abstraction. A model scaling technique should not inhibit this ability. For instance, a model translator can translate a model into some other artifacts (e.g., code, simulation scripts). Instead of scaling the original model, some scalability approaches may integrate the scalability task into the generation of final artifacts or other intermediate representations. The disadvantage of such an approach is that models are not the catalyst for representing the scalability result (i.e., the scalability is not represented directly in a model, but in the generated artifact), which inhibits the benefit of using models. By contrast, the MTBD approach directly operates on model instances, retaining all of the benefits of modeling.

General across multiple modeling languages. This characteristic ensures that the scaling approach should be applicable to different modeling languages. The MTBD implementation is a plug-in to GEMS, and triggered 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 scalability transformation problems, which means MTBD is a general solution that can be applied across multiple domain-specific modeling languages.

Flexible to support user extensions. The desired scaling process should allow alteration of the semantics of the replication more directly using a language that can be manipulated easily by an end-user. The current generated transformation pattern is not editable, and therefore does not allow direct extension or reuse on an existing pattern. However, we believe that the user-focus of MTBD allows a user to re-demonstrate a new task in a manner that is better than editing existing model transformation code. If the end-user has no idea of programming or model transformation languages, user extension by altering the model transformation is simply not even possible.

5.2 The Benefits of Automating Model Scalability using MTBD

The benefits of MTBD can be compared with writing model transformation rules to solve the same problems, as was done in [3] and [6]. In these earlier works, we used a model transformation engine called C-SAW, which processed a transformation language called the Embedded Constraint Language (ECL). We used ECL to perform the same model scalability tasks that were introduced in Sections 2.1 and 2.2. Although ECL is specific to model transformation tasks and is at a higher level than general-purpose programming languages, its usage still requires that a user learn the syntax of the language. The use of ECL also requires a deep understanding of the metamodel for the domain being scaled. Additionally, ECL requires understanding of basic programming concepts such as variable declaration, branch statements, and even recursion. Thus, for a general domain expert who does not have any programming language knowledge or experience, ECL is often too challenging to use as a model scalability solution. In fact, our understanding of this problem came after performing the work described in [3] and [6], leading us to the realization that a new automation approach was needed, which motivated our work on MTBD.

Comparatively, MTBD does not use any model transformation language. Users only need to perform a single case of the scaling process on a concrete model instance. Every operation or user refinement is done at the concrete model instance level, not at the metamodel level, as needed with traditional transformation languages. MTBD enables users to solve complex scaling problems while being ignorant of the underlying metamodel definition.

To better compare the efforts of automating model scalability tasks using MTLs and MTBD, Figure 14 shows part of the model transformation rules written in ECL to implement sub-task t1 of Example 2.2. To add a Gateway, the necessary objects should be declared first, followed by calling creational APIs to create the correct type of elements and the connections. However, the same task could be accomplished by only two operations in the demonstration, as shown in List 9.

```
//add one CORBA_Gateway and connect it to Event_Channel
strategy addGateWay(j: integer)
{
  declare ec, site_gw : object;
  addAtom("CORBA_Gateway", "CORBA_Gateway");
  ec := findModel("Event_Channel");
  site_gw := findAtom("CORBA_Gateway");
  addConnection("LocalGateway_EC", site_gw, ec);
}
```

Figure 14. An excerpt of transformation rule written in ECL to accomplish sub-task t1 of Example 2.2

In addition, to add the *Gateway* to each existing *Site*, and control the number of execution times, recursive calls are used in the ECL transformation rules as shown in Figure 15. In MTBD, a user simply identifies the two operations in List 9 as generic after the demonstration. The inferred transformation can then be executed as many times as needed.

```
//traverse the original sites to add CORBA_Gateways
//n is the number of the original sites
//m is the total number of sites after scaling
strategy traverseSites(n, i, m, j : integer)
{
  declare id_str : string;
  if (i <= n) then
    id_str := intToString(i);
    rootFolder().findModel("NewGateway_Federation").
    findModel("Site " + id_str).addGateWay_r(m, j);
    traverseSites(n, i+1, m, j);
  endif;
}

//recursively add CORBA_Gateways to each existing site
strategy addGateWay_r(m, j: integer)
{
  if (j<=m) then
    addGateWay(j);
    addGateWay_r(m, j+1);
  endif;
}
```

Figure 15. An excerpt of transformation rule written in ECL to enable adding a *Gateway* to each existing *Site*, while controling the number of execution times

We have not done a formal user study on the comparison between the two approaches. However, Table 1 lists some of the results of the comparative effort, indicating that [6] used over 170 lines of ECL code to address Example 2.1. We performed the same task with MTBD by demonstrating 35 editing operations on a concrete model instance, and identifying one generic operation. Similarly, the solution of Example 2.2 using ECL requires 32 lines of ECL [3], while our MTBD-based scalability solution required 16 direct editing operations, one generic operation identification and two precondition refinements.

Model Scalability Example	MTBD	ECL Rules
Example 2.1	35 operations 1 generic operation refinement	170 SLOC
Example 2.2	16 Operations 2 precondition refinement 1 generic operation refinement	124 SLOC

Table 1. The comparison of effort to solve model scalability tasks using MTBD and a model transformation language (ECL)

5.3 Current Limitations of MTBD and Future Work

When designing and implementing MT-Scribe, a tradeoff existed between simplicity and functionality, because a user's demonstration and refinement are not as expressive and accurate as the same transformation task written in an MTL. Some tasks could be easily specified by a transformation language, but turn out to be very difficult to demonstrate. For instance, scaling an element having the maximum value of a specific attribute is currently not possible in MT-Scribe. The same task could be implemented by function calls, selection or iteration facilities available in most MTLs. Although these kinds of functions could be extended to MTBD by designing some other user-friendly refinement interfaces, its simplicity after adding many user feedback steps would probably be undermined.

Therefore, since it is not easy to make MTBD a fully complete replacement to a well-defined model transformation language to support all possible model scalability tasks, our initial focus has been toward making MT-Scribe practical for most scenarios. When encountering difficulties in using MTBD to solve common model scalability problems in practice, the most needed and essential features and functions will be selected and added into MT-Scribe by designing user-friendly and user-centric interfaces and mechanisms that are capable of implementing the desired function. By such an incremental and selective extension process, we believe a proper balance can be achieved between simplicity, functionality, and practicality.

6 Related Work

Software scalability in computer systems has been well-recognized and defined. Bondi [1] provided a comprehensive analysis on the characteristics of software scalability and the impact on performance. However, automating scalability on models in the context of MDE has not been widely investigated. Gray et al. investigated model scalability [3] and proposed the use of ECL to automate model scalability tasks [3][6]. They point out that model scalability is an endogenous model transformation task and other model transformation languages (MTLs) and tools can be used to automate model scalability tasks. In this summary of related work, we analyze the traditional model transformation approaches that can be used to automate model scalability in Section 6.1. In Section 6.2, we overview some innovative approaches that can potentially simplify the automation of model scalability tasks using approaches similar to our own work on MTBD.

6.1 Traditional model transformation approaches that can support automating model scalability

One of the most direct ways to automate model scalability tasks is to use General-purpose Programming Languages (GPLs). Most modeling tools provide APIs that assist in the direct manipulation of an internal representation of the model instance. The model scalability procedures can be encoded in a GPL, such as Java and C++, which developers are generally comfortable and familiar with, avoiding extra training to write transformations. However, the power of transformations is often restricted by the APIs. Furthermore, GPLs lack the high-level abstractions to specify models and scaling transformation rules, making the GPL-based transformations difficult to write, understand, and maintain [4].

Because many modeling tools support importing and exporting model instances in the form of XMI, it is possible to use the existing XML tools such as XSLT [23] to scale models outside of the modeling tool infrastructure. Although XSLT is specifically used to transform models and has a higher level of abstraction compared with GPLs, it is tightly coupled to XML, forcing the specification of transformations using concepts at a lower level of abstraction. In addition, transformations performed outside of a modeling tool exert a potential risk that the models being transformed cannot be correctly imported or exported with future versions of the tool.

Currently, the most mature approach to automate model scalability tasks is to specify the transformation rules by using specialized MTLs [5]. A specialized transformation language provides a set of constructs for explicitly specifying the behavior of the transformation, which can typically be written more concisely than GPL and XML-based transformation approaches. There are two major types of MTLs in this category: textual hybrid MTLs and graphical MTLs. The former type usually combines both declarative and imperative constructs to perform a transformation. Declarative constructs are used to specify source and target patterns as direct mapping rules, and imperative constructs are used to implement sequences of instructions (e.g., explicitly specifying how the scaling process should be realized). ATL [8] and ECL [6] are examples of textual hybrid MTLs. By comparison, graphical MTLs convert the task of scaling a model into a graph transformation problem by utilizing graph matching and rewriting techniques. A typical graphical MTL usually defines a transformation rule as a LHS (left-hand side) graph representing the source model and a RHS (right-hand side) graph representing the target model. Then, the engine automatically matches the LHS graph in a model and changes it into the desired RHS graph. Compared with textual hybrid MTLs, it is easier to define specific model patterns using graphs, leading to a simplification of the transformation rules in many cases. However, graphical MTLs are not as expressive as textual definitions, resulting in less powerful functionality in some model scalability scenarios. Typical MTLs in this category are GreAT [24], and VIATRA [25]. However, whether a MTL has a high level of abstraction, graphical or textual, its usage on

automating model scalability always suffers from the challenges mentioned in Section 1 (i.e., the steep learning curve and need to understand the details of the underlying metamodel), preventing a wide range of end-users from contributing to model scalability tasks using their expertise.

6.2 Innovative model transformation approaches that can potentially simplify the model scalability tasks

Some innovative model transformation approaches have been proposed and developed as alternatives to MTLs. These new approaches share a similar goal of making the specification of model transformation easier and more user-friendly, requiring less knowledge of MTLs and metamodels. These innovations provide strong potential to simplify the automation of model scalability tasks.

Model Transformation By Example (MTBE) [16] is an innovative approach to address the challenges inherent from using model transformation languages. Instead of writing transformation rules manually, MTBE enables users to define a prototypical set of interrelated mappings between the source and target model instances, and then the metamodel-level transformation rules can be inferred and generated semi-automatically. In this context, users work directly at the model instance level and configure the mappings without knowing any details about the metamodel definition or the hidden concepts. With the semi-automatically generated rules, the simplicity of specifying model transformations is greatly improved. As first introduced by Varró [16], the prototypical transformation rules of MTBE can be generated partially from the user-defined mappings by conducting source and target model context analysis. Varró later proposed a way to realize MTBE by using inductive logic programming [26]. Similarly, Strommer and Wimmer implemented an Eclipse prototype to enable generation of ATL rules from the semantic mappings between domain models [17][27]. Instead of using logic programming engines, the inference and reasoning process is based on pattern matching.

However, the current state of MTBE research still has some limitations and is not very appropriate to automate model scalability tasks. The semi-automatic generation often leads to an iterative manual refinement of the generated rules; therefore, the model evolution designers are not isolated completely from knowing the transformation languages and the metamodel definitions. In addition, the inference of transformation rules depends on the given sets of mapping examples. In order to obtain a complete and precise inference result, one or more representative examples must be available for users to setup the prototypical mappings, but seeding the process with the proper scalability examples is not always an easy task. Furthermore, current MTBE approaches focus on mapping the corresponding domain

concepts between two different metamodels without handling complex attribute transformations. Therefore, it is impossible to automate the configuration of attributes in the scaling process, which is commonly required in practice. Furthermore, current MTBE approaches fit the exogenous model transformation concept very well to map the concepts between two different domains, but they are not very practical when it comes to endogenous model transformations, which presents limitations in supporting model scalability evolution activities.

Brosch et al. introduced a method for specifying composite operations within the user's modeling language and environment of choice [18][28]. The user models the composite operation by-example, changing a source model into the desirable target model. By comparing the source and target states, the specific changes can be summarized by a model difference algorithm. After giving additional specification of the pre-condition and post-condition, an Operation Specification Model (OSM) can be generated that represents the composite operation scenario and can be used to generate other transformation artifacts. Similar to MTBE, users can work on the concrete model instance level without knowing about the metamodel to define composite operations through examples. Although user refinement (e.g., specification of pre- and post- conditions) is also needed to make the generated transformation complete and accurate, the refinement is done at the example level through the given interfaces rather than at the generated transformation rule when using MTBE. In addition, the composite operation focuses on endogenous model transformation, which could be potentially used to support automating model scalability tasks. However, the major limitations with this approach are: 1) Even though the refinement process is not on the level of generated model transformation rules, some programming concepts are involved, making this process dependent on technical skills that some domain experts may not possess; 2) Attribute transformation has not been considered and implemented, which shares the same problem as MTBE; 3) In the generation of artifacts for a certain scenario, a manual binding process is required to map the elements in the OSM to the new concrete model. Although a user-friendly interface has been developed to simplify the procedure, the manual binding process would become a problem when a large number of model elements and connections are present in a scaling scenario.

7 Conclusion

This paper introduces a demonstration-based model transformation approach to automate model scalability tasks in order to support software evolution. Compared with our previous work on using model transformation languages to scale models [3][6], we believe that MTBD

offers several advantages supporting ease of use. The demonstration focus allows users to be ignorant of both the details of the transformation language, as well as the structure of the metamodel for the language being used. The paper presented a new formal model of MTBD and summarized the changes that were needed to evolve an earlier version of our approach (MT-Scribe) to address challenges related to precondition extraction, attribute refactoring, and transformation site matching. Three case studies were used to demonstrate the application of our improved technique in order to address a variety of scalability scenarios. We believe that scalability issues will become more prominent as the concepts of MDE are integrated further into development processes. The key contribution of this paper is an approach for helping to overcome the challenges associated with scaling models in MDE processes.

Acknowledgement

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

References

1. Bondi, A.: Characteristics of scalability and their impact on performance. 2nd International Workshop on Software and Performance, Ottawa, Ontario, Canada, pp. 195-203 (2000)
2. Schmidt, D.: Model-Driven Engineering. IEEE Computer, vol. 39 no. 2, pp. 25-32 (2006)
3. Gray, J., Lin, Y., Zhang, J., Nordstrom, S., Gokhale, A., Neema, S., Gokhale, S.: Replicators: Transformations to address model scalability. In Proceedings of the International Conference on Model-Driven Engineering Languages and Systems (MoDELS), Springer-Verlag LNCS 3713, Montego Bay, Jamaica, October 2005, pp. 295–308 (2005)
4. Sendall, S., Kozaczynski, W.: Model transformation - The Heart and Soul of Model-Driven Software Development. IEEE Software, Special Issue on Model Driven Software Development, 20(5):42–45 (2003)
5. 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)
6. Lin, Y., Gray, J., Zhang, J., Nordstrom, S., Gokhale, A., Neema, S., Gokhale, S.: Model Replication: Transformations to Address Model Scalability. Software: Practice and Experience, vol. 38, no. 14, pp. 1475-1497 (2008)
7. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal, v.45 n.3, p.621-645 (2006)
8. 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)
9. OMG, Revised Submission for MOF 2.0 Query/View/Transformations RFP (ad/2002-04-10), OMG Document ad/2005-07-01 (2005)

10. Sun, Y., White, J., Gray, J.: Model Transformation by Demonstration. Model Driven Engineering Languages and Systems (MoDELS), Spring-Verlag LNCS 5795, Denver, CO, October 2009, pp. 712-726 (2009)
11. Edwards, G., Deng, G., Schmidt, D., Gokhale, A., Natarajan, B.: Model-Driven Configuration and Deployment of Component Middleware Publish/Subscribe Services. Generative Programming and Component Engineering (GPCE), Vancouver, BC, October 2004, pp. 337-360 (2004)
12. Groovy. http://groovy.codehaus.org/
13. Muppala J., Ciardo G., Trivedi K.: Stochastic Reward Nets for Reliability Prediction. Communications in Reliability, Maintainability and Serviceability, vol. 1, no. 2, pp. 9-20. (1994)
14. Mens, T., Gorp, P.: A Taxonomy of Model Transformation. Workshop on Graph and Model Transformation, vol. 152, Talinn, Estonia, pp. 125-142 (2005)
15. Agrawal, A., Karsai, G., Lédeczi, Á.: An End-to-End Domain-Driven Software Development Framework. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) – Domain-driven Track, Anaheim, CA, pp. 8-15 (2003)
16. Varró, D.: Model Transformation by Example. Model-Driven Engineering Languages and Systems (MoDELS), Springer-Verlag LNCS 4199, Genova, Italy, October 2006, pp. 410–424 (2006)
17. Strommer, M., Wimmer, M.: A Framework for Model Transformation by-example: Concepts and Tool Support. 46th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS), Zurich, Switzerland, July 2008, pp. 372–391 (2008)
18. 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), Spring-Verlag LNCS 5795, Denver, CO, October, 2009, pp. 271-285 (2009)
19. Generic Eclipse Modeling System (GEMS), http://www.eclipse.org/gmt/gems/
20. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-specific Design Environments. IEEE Computer, 34(11), pp. 44-51 (2001)
21. Schmidt, D., Stal, M., Rohnert, H., Buschman, F.: Pattern-Oriented Software Architecture – Volume 2: Patterns for Concurrent and Networked Objects. John Wiley and Sons (2000)
22. MTBD Project Page. http://www.cis.uab.edu/softcom/mtbd
23. W3C, XSLT Transformation version 1.0.http://www.w3.org/TR/xslt (1999)
24. 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)
25. 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)
26. 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)

27. 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)
28. Brosch, P., Seidl, M., Wieland, K., Wimmer, M., Langer, P.: The Operation Recorder: Specifying Model Refactorings By-example. International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA) – Tool Demonstration, Orlando, FL, October 2009, pp. 791-792 (2009)
29. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting Metamodels to Ontologies - a Step to the Semantic Integration of Modeling Languages. International Conference on Model-Driven Engineering Languages and Systems (MoDELS), Springer-Verlag LNCS 4199, Genova, Italy, October 2006, pp. 528-542 (2006)
30. Object Management Group, Object Constraint Language Specification. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL. (2010)
31. Sun, Y., White, J., Gray, J., Gokhale, A.: Model-Driven Automated Error Recovery in Cloud Computing. Model-driven Analysis and Software Development: Architectures and Functions, IGI Global, Hershey, PA (2009)
32. White, J., Czarnecki, K., Schmidt, D., Lenz, G., Wienands, C., Wuchner, E., Fiege, L.: Automated Model-based Configuration of Enterprise Java Applications. Enterprise Distributed Object Computing (EDOC), October, 2007, pp. 301-312, Annapolis, Maryland (2007)
33. Gray, J., Tolvanen, J., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-Specific Modeling. Handbook of Dynamic System Modeling, CRC Press, Chapter 7, pages 7-1 through 7-20 (2007)
34. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute, Technical Report CMU-SEI-90-TR21, Carnegie Mellon University. (1990)
35. Hayes, B: Cloud Computing. Communications of the ACM, vol. 51, no. 7, pp. 9-11. (2008)
36. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/ (2010)