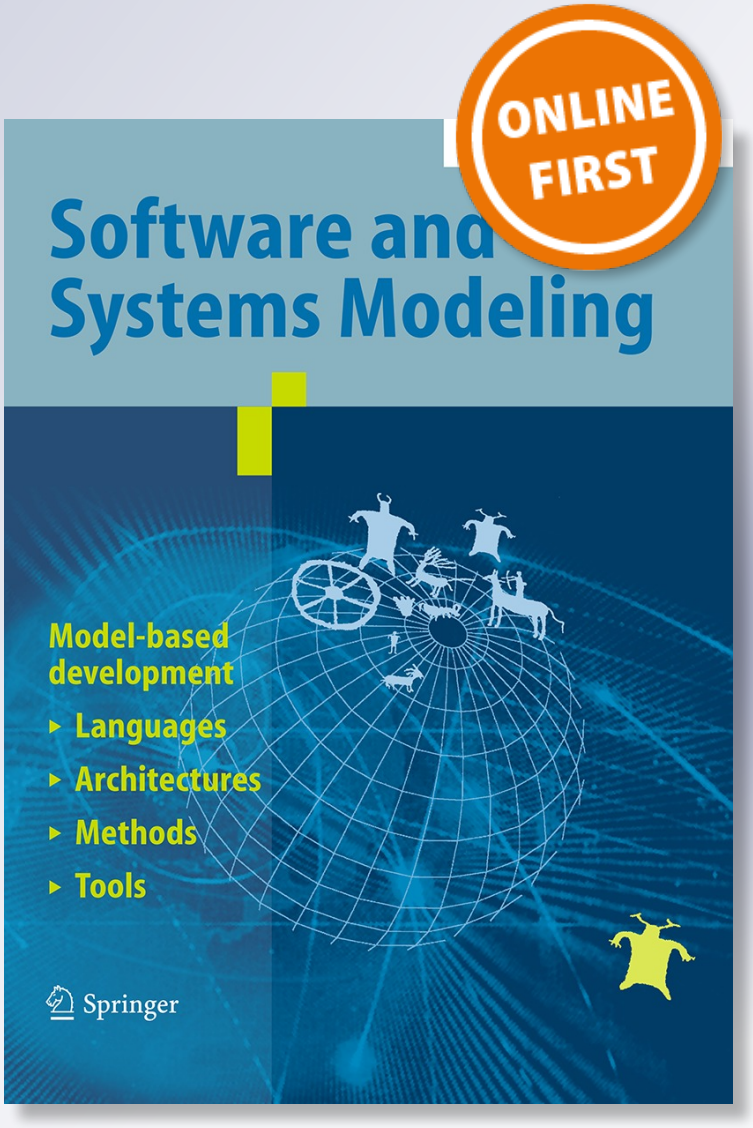
A demonstration-based model transformation approach to automate model scalability

Yu Sun, Jeff Gray & Jules White

Software & Systems Modeling

ISSN 1619-1366

Softw Syst Model
DOI 10.1007/s10270-013-0374-0



Springer

Softw Syst Model
DOI 10.1007/s10270-013-0374-0

REGULAR PAPER

A demonstration-based model transformation approach to automate model scalability

Yu Sun · Jeff Gray · Jules White

Received: 11 September 2011 / Revised: 3 August 2013 / Accepted: 7 August 2013

Abstract An important aspect during software development is the ability to evolve and scale 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. 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 simulate the model replication processes. By recording a user's operations, an inference engine analyzes the user's demonstration and generalizes 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

Communicated by Prof. Martin Gogolla.

Y. Sun (✉)
Department of Computer and Information Sciences,
University of Alabama at Birmingham, Birmingham, AL, USA
e-mail: yu.sun.cs@gmail.com

J. Gray
Department of Computer Science, University of Alabama,
Tuscaloosa, AL, USA
e-mail: gray@cs.ua.edu

J. White
Department of Electrical and Computer Engineering, Virginia Tech,
Blacksburg, VA, USA
e-mail: julesw@vt.edu

1 Introduction

Software systems often need to evolve in order to accommodate new features, to process larger workloads or to handle other scaling issues [5]. Scaling a software system, as a key type of software evolution, is crucial for a system's long-term success. Although new software engineering tools and methodologies are being developed consistently, ensuring good software scalability has always been a challenging issue. For instance, with the increasing adoption of

Published online: 14 September 2013

Model-Driven Engineering (MDE) [1,28], 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 [34] and avionics domains [12]) has promoted models to an important role in software development. The models representing the designs and systems in complex domains will continue to require tools and techniques to address issues of model evolution, particularly in the area of model scalability, which is the activity of building a complex model from a base model by adding, replicating, or modifying its model elements, connections, or substructures [22]. The need for model scalability often emerges in the context of domain-specific modeling when model engineers need to explore various design alternatives in a system or software model (e.g., scaling a model of a network of communicating nodes from a base of 20 nodes to a larger system with 300 nodes). The amount of manual effort to scale system and software models is often prohibitive due to the many connections and nodes that must be created do address a scalability need.

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 [19] 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 [15] 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 [39] and to monitor and control the infrastructure at runtime [33]. In order to allocate additional infrastructure to handle larger workloads, the underlying deployment models must be scaled. Thus, model scalability [22] 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 [12]. Although manually editing and scaling models is the most direct approach, it is often 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 [22], especially when the scalability tasks crosscut the hierarchy of the system structure. Therefore, model scalability is different from other types of model transformation tasks where manual transformation is still feasible (e.g., it is often practical to perform a manual refactoring on a UML model). The process of scaling models is often too challenging to accomplish manually, and therefore can benefit immensely from automation.

Model transformation has proven to be an effective approach to automate model scalability tasks [22]. 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) [8]. Compared with other types of endogenous model transformations, model scalability generally requires many more operations to add or remove elements, and the transformation execution engine should be capable of executing a certain transformation repeatedly to evolve the model to any desired scale [12]. Many 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 state that is more complex.

Although MTLs are powerful and expressive approaches to automate several 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 [27] 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 [20], which makes comprehension more difficult. In the context of MDE, domain experts without a programming or computer science background can participate in building and using software models. However, 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. In addition to the benefit of end-user usability, we also aim to retain the benefits of using models, being applicable to any modeling languages and flexible to extend. The approach described in this paper extends our previous work, model transformation By demonstration (MTBD) [32], which 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 and the associated tool (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. It is worth noting that our new extensions are not only applicable to model scalability tasks, but they have also been used in other scenarios such as model refactoring, model layout configuration, and aspect-oriented modeling. In this paper, we focus on the model scalability task specifically, rather than a general model transformation context, in order to accommodate the demanding scalability challenges in the large-scale model editing and maintenance activities, as well as better highlighting the advantages of our approach over the alternative approaches of manual scaling or MTL-based approaches.

The rest of the paper is organized as follows. Background information and related work is first discussed in Sect. 2. Three model scalability scenarios in different domains are presented in Sect. 3 to motivate the need to support software evolution by automating model scalability. In Sect. 4, the original MTBD project combined with its limitations in dealing with model scalability is 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 Sect. 5. Section 6 evaluates the new approach, pointing out its advantages and limitations, with Sect. 7 offering concluding remarks.

2 Related work

Software scalability in computer systems has been well recognized and defined. Bondi [5] provided a comprehensive analysis of 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 [14] and proposed the use of a model transformation language to automate model scalability tasks [14,22]. They point out that model scalability can often be specified as an endogenous model transformation, and other 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 Sect. 2.1. In Sect. 2.2, we overview some innovative approaches that have the potential to simplify the automation of model scalability tasks using approaches similar to our own work on MTBD.

2.1 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 transformation is often restricted by the APIs offered by a specific modeling tool. 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 [30].

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 [38] 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 [12,23,26]. A specialized transformation language provides a set of constructs for explicitly specifying the behavior of the transformation, which typically can 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 [18] and ECL [22] 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. Typical MTLs in this category are GreAT [3] and VIATRA [4]. 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 Sect. 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.

2.2 Innovative model transformation approaches that can simplify model scalability tasks

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

Model Transformation By Example (MTBE) [36] 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 [36], 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 [37]. Similarly, Strommer and Wimmer implemented an Eclipse prototype to enable generation of ATL rules from the semantic mappings between domain models [31,40]. Instead of using logic programming engines, their inference and reasoning process is based on pattern matching.

However, the current state of MTBE research still has some limitations and may not be the best approach 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 likely to be a time-consuming task that may not provide the level of detail to infer all of the scalability needs. 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 one-to-one between two different domains, but they are not as practical when it comes to endogenous model transformations where one-to-multiple or multiple-to-multiple mappings between the source and target models are involved, which presents limitations in supporting model scalability evolution activities.

Brosch et al. [6,7] introduced a method for specifying composite operations within the user's modeling language and environment of choice. The user models the composite operation by example, changing a source model into the desired 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 precondition 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 used potentially to support automating model scalability tasks. However, the limitations with this approach are as follows: (1) Even though the refinement process is not on the level of generated model transformation rules, some programming concepts are involved (e.g., `includesAll()`, `isEmpty()`, and some iteration control), 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.

Beyond the model transformation area, Damm and Harel have investigated system interaction modeling, which offers

important guidance to our work [9]. They extended the Message Sequence Chart (MSC) to Live Sequence Chart (LSC) by introducing more expressive, generic, and executable mechanisms to capture more behavioral requirements. For instance, LSC provides different types of loops for the specification of iteration, which is similar to our generic operations. Because LSC has a deep formal and theoretical foundation, we believe that the MTBD demonstration process might be specified formally in a similar manner or even converted to LSC under certain conditions, which means that we could potentially reuse the formalism of LSCs to consolidate the formal specification of MTBD.

3 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. For each of the three examples in Sects. 3.1, 3.2, and 3.3, 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 Sect. 3.4. The specific approach for using MTBD to address the needs arising from these examples will be given in Sect. 4.

3.1 Adding new event types: evolving software design models

Stochastic reward nets (SRNs) [25] can be used for evaluating the reliability of complex distributed systems. The Stochastic Reward Net Modeling Language (SRNML) was developed to describe SRN models of large distributed systems [22], in order to design and model performance-based system properties such as the schedulability, performance, and time profiles. For example, the SRN model defined by SRNML in Fig. 1 depicts mechanisms to handle synchronous event demultiplexing and dispatching when applying the reactor pattern [29] in middleware for network services. In [22], we showed how scalabilty could be addressed using a traditional MTL; in this paper, we show how the MTBD approach can be used as an alternative for describing scalability tasks.

The reactor pattern handles service requests to a service handler from one or multiple input events concurrently. Whenever an event comes in, the service handler demultiplexes the incoming event to its associated event handler. Thus, an SRN model consists of two parts: the event types handled by a reactor and the associated execution snapshot. The execution snapshot specifies 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 Fig. 1, the original model has two event types, 1 and 2. Each event type is represented by a set of elements including its arrival transition (e.g., *A1*), to queuing place (e.g., *B1*) and finally service (e.g., *Sr1*) through the immediate transitions (e.g., *Sn1*). The immediate transition is enabled when a snapshot is taken. The arc from the queuing place back to the arrival transition is used to avoid the firing of a transition in special cases. The bottom of the original model in Fig. 1 represents 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*).

Scalability Scenario in SRNML (Example 3.1) The scalability challenges of SRN models are triggered when new event types and the corresponding connections with event handlers are added. As shown in the bottom of Fig. 1, when two new event types (3 and 4) need to be modeled, two new sets of event types and connections (i.e., from *A*3 to *Sr3*, from *A*4 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. Thus, adding new event types always results in creating new sets of model elements and making connections between the new event type to every existing event type. Section 5.1 will explain the detailed scaling process step by step.

3.2 Enlarging event services: evolving software implementation models

The Event QoS Aspect Language (EQAL) [10] is a domain-specific modeling language (DSML) to configure publisher–subscriber event services for large-scale distributed systems. Publisher–subscriber is an effective mechanism that can be used in event-based communications, in order to reduce software dependencies, enhance system composability and evolution, and enable anonymous and asynchronous communications among application components. Several EQAL model transformations have been implemented that take EQAL models as input and generate publisher–subscriber service configuration files, component property descriptions, and part of the underlying code to support system

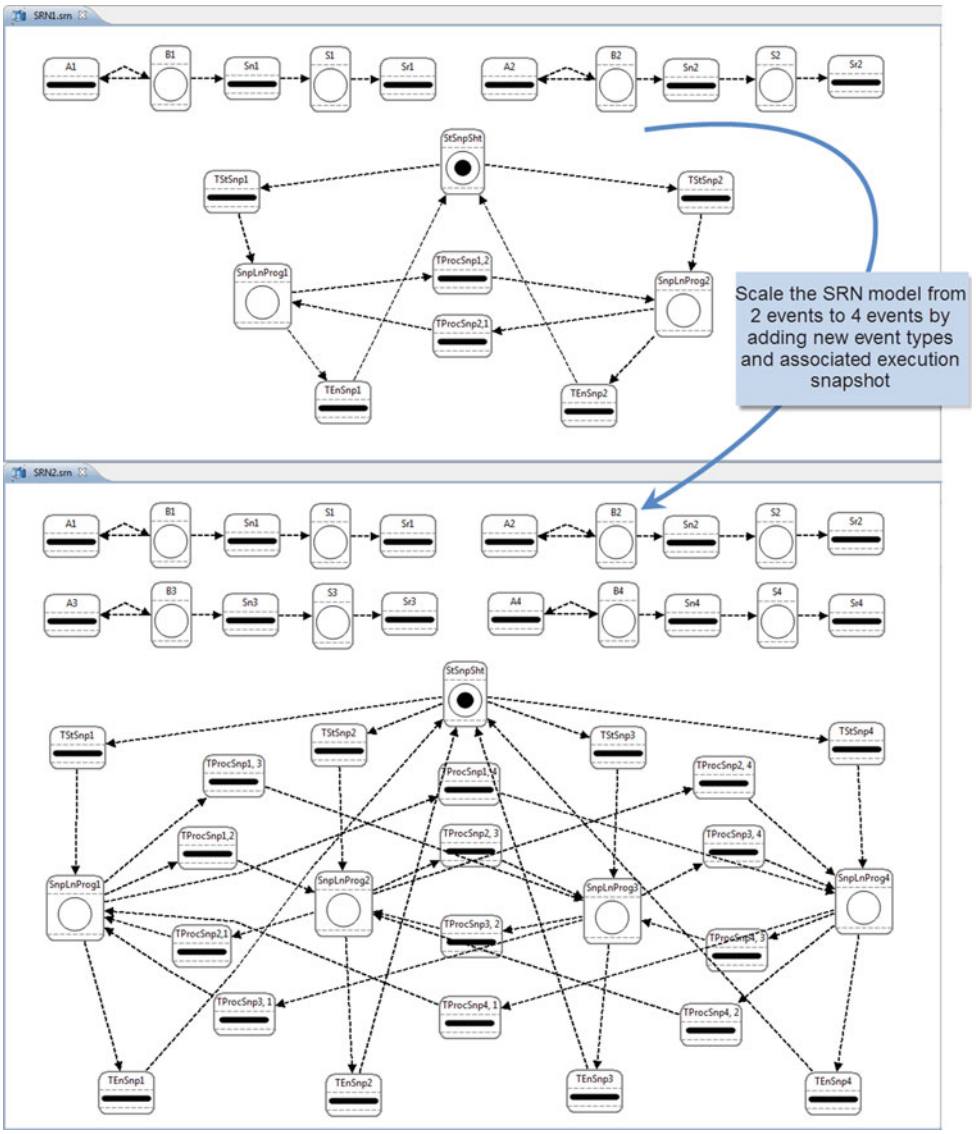


Fig. 1 An SRN model before (*top*) and after (*bottom*) scaling

implementation. We previously used a model transformation language to perform scalability tasks [14], but focus in this paper on how such tasks can be demonstrated.

The top of Fig. 2 illustrates a model defined by EQAL to specify a federation of event channels in different sites, which allows sharing of filtering messages 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*. The *EventSupporter* acts as a publisher to generate events to be transmitted, while the *EventConsumer* receives events through hook operations. The *EventChannel* accepts the events from *EventSupporter* and delivers them to *EventConsumer* through*Gateways*.

Scalability scenario in EQAL (Example 3.2) One complex scalability issue in EQAL arises when scaling a small federation of event services to a large one, which involves creating or replicating a large number of publishers, subscribers, and the associations. The bottom of Fig. 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 amongthem. In addition,

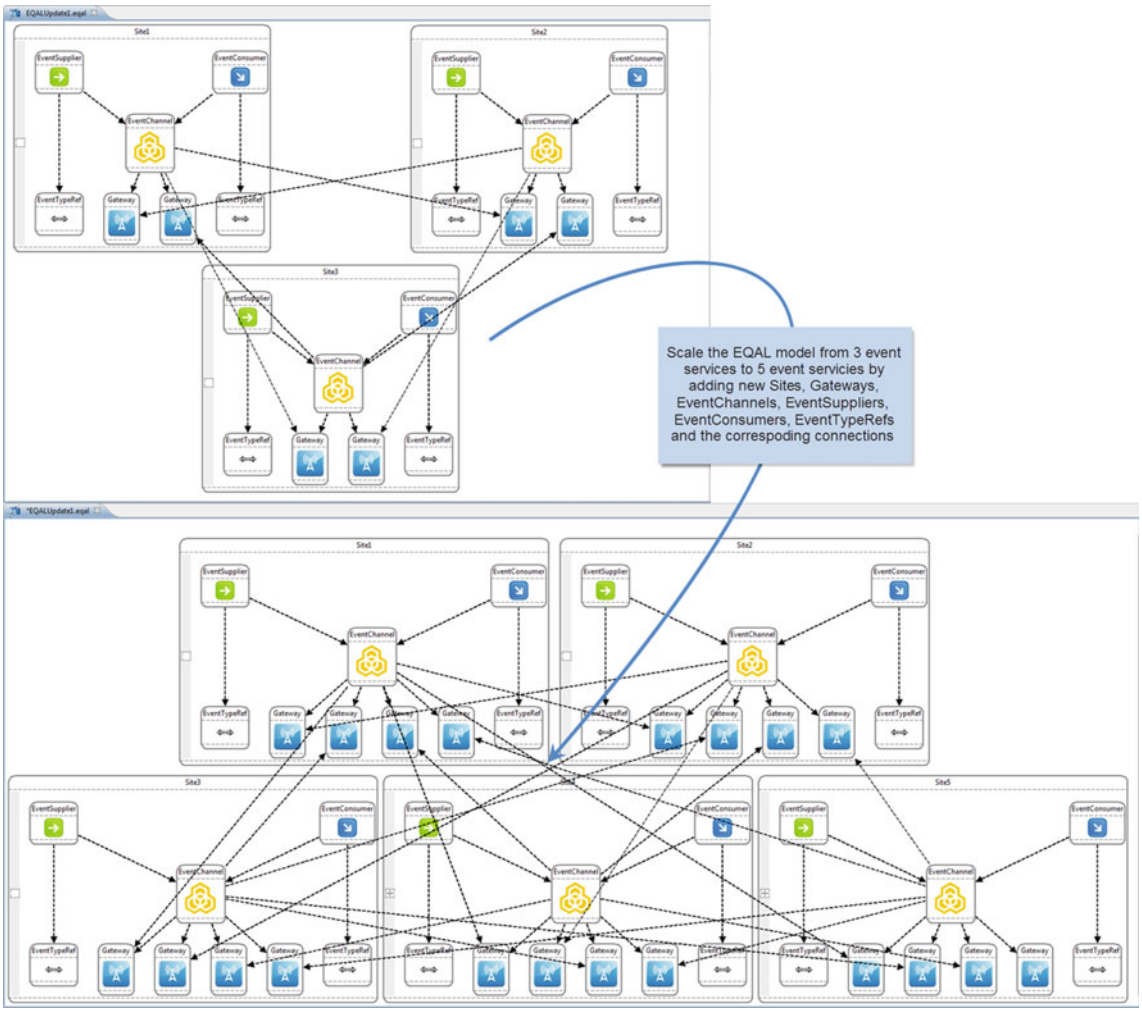


Fig. 2 An EQAL model before (*top*) and after (*bottom*) scaling

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*.

3.3 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 [17]. 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.

The Cloud Computing Management Modeling Language (C2M2L) [33] 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 Fig. 3 shows a diagram of an EJB cloud application deployed in Amazon EC2 [2], 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. Our previous work considered the

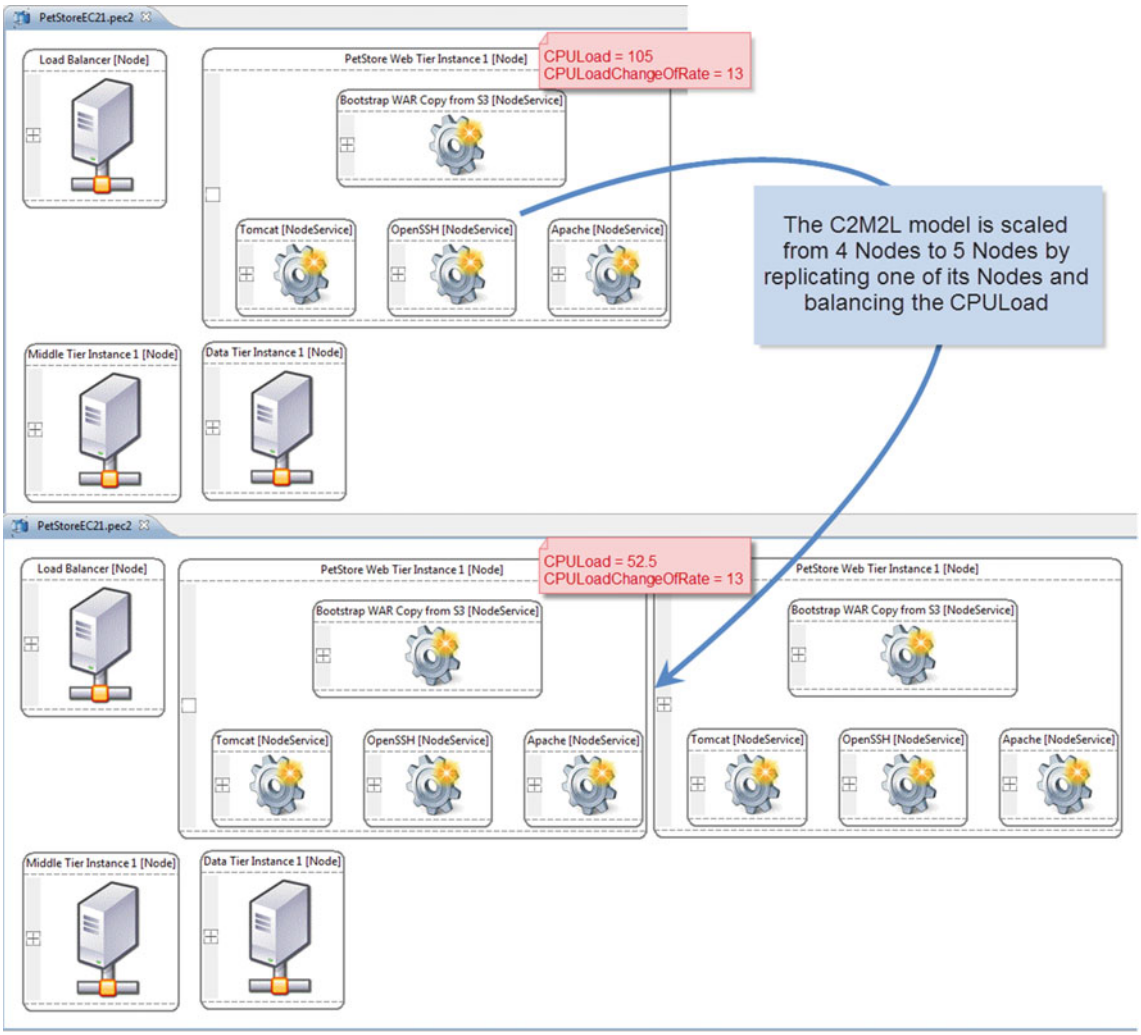


Fig. 3 A C2M2L model before (*top*) and after (*bottom*) scaling

adoption of a model transformation language to perform scalability tasks on C2M2L models [33]. In this paper, we focus on the opportunities to demonstrate such tasks without the need for a model transformation language.

Scalability Scenario in C2M2L (Example 3.3) 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 Fig. 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 are essential to ensure that applications are running correctly and smoothly.

3.4 Challenges of model scalability current practice

Scalability issues often emerge in systems and software models due to the need to explore various design alternatives (e.g., what happens when the number of base nodes scales from 150 to 300, and how is the model brought to a correct state to understand the tradeoffs and requirements for such a new design change?). For each of the model scalability scenarios presented thus far in this paper, 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 two-event SRN model, creating one event service for a 3-Site EQAL model, or replicating two new *Nodes* in a C2M2L model). The smaller examples presented in these figures were chosen to illustrate the basic needs of a scalability task. However, it becomes extremely challenging to scale each of these scenarios manually 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, such as: (1) locating the correct part of a model to be scaled, (2) creating proper elements and connections, (3) precisely replicating elements and connections, (4) setting up correct properties, and (5) making accurate connections between existing elements and newly created ones. Some of these examples (i.e., from Sects. 3.1 and 3.2) have been automated in the past using a model transformation language (MTL) [14,22,33]. However, the end users of these languages (e.g., domain experts, such as cloud computing administrators) might not have experience in using MTLs. Although it is possible for end users and model transformation experts to work together to solve the problems, ensuring a correct communication between the two sides is not always an easy task. Therefore, a simpler approach is needed to assist end users in specifying their own model scalability scenarios.

4 Automating model scalability using MTBD

In this section, we first give a brief introduction to the ideas behind MTBD (Sect. 4.1). Then, the key limitations that initially prevented MTBD from being applied to model scalability tasks are identified (Sect. 4.2), followed by the enhancements we have made to address these limitations (Sect. 4.3).

4.1 Overview of MTBD

MTBD is a model transformation approach motivated by the difficulties that domain experts experience when they try to learn model transformation languages and understand metamodel definitions. We first introduced the core idea of MTBD in [32]. The basic idea (illustrated in Fig. 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 (*User Demonstration*). A recording and inference engine has been developed to capture all user operations performed during the demonstration (*Operation Recording*). After the recording process has completed, the inference engine optimizes

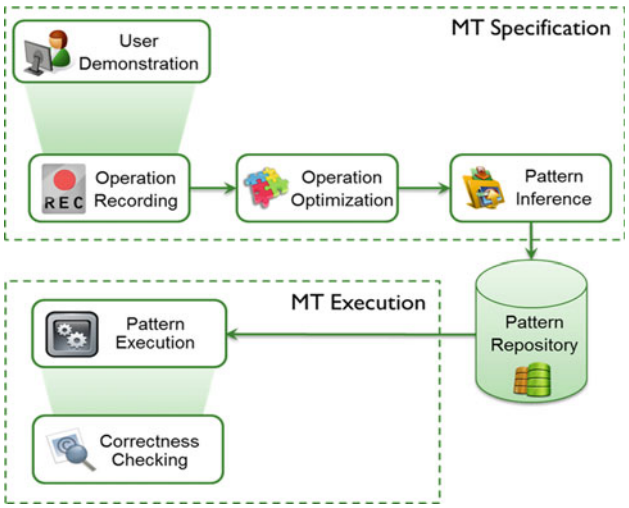


Fig. 4 Overview of MTBD (adapted from [32])

the recorded operations (*Operation Optimization*) and infers a transformation pattern that specifies the precondition of the transformation and the sequence of actions needed to realize the transformation (*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 (*Pattern Execution*). During the execution of a transformation pattern, constraint checking ensures that the execution does not violate the metamodel definition of the domain.

The idea of MTBD has been implemented as an Eclipse plug-in for the Generic Eclipse Modeling System (GEMS) [11], resulting in a tool called MT-Scribe. The project page for MT-Scribe [24] contains sample videos and the open source implementation. Without using any model transformation languages or the need to understand metamodels, domain experts who are not computer scientists are able to demonstrate endogenous model transformations and execute generated transformation patterns in an 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.

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 Fig. 5).

After the demonstration is completed, a transformation pattern can be inferred. This pattern specifies the precondition as shown in List 2, which specifies the required elements

List 1 Operations performed in the demonstration

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

List 2 Precondition—elements needed and corresponding metatypes

Elements needed for operations	Element	MetaType
elem1.elem2	elem1	ModelRoot
elem1.elem2.elem3	elem2	Site
	elem3	Gateway

List 3 Transformation actions

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

for this transformation with their containment relationship as well as the metatypes. In this case, the precondition is a *Site* in the *ModelRoot* which contains one *Gateway*. The transformation actions (i.e., adding a new *Gateway* and changing its name) are listed in List 3. These lists are abstract representations of how the pattern is saved, which are invisible to end users. We use generic names (e.g., "elem") to indicate that the elements have been generalized from the concrete demonstration examples.

A user may then apply this pattern to any other EQAL model. The engine will traverse the model and match the precondition using a backtracking 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`" (Fig. 6 shows the pattern applied to an EQAL model containing six *Sites*).

4.1.1 Operation recording

Considering a model transformation process as a function, the goal of MTBD is to allow users to express domain knowledge regarding a function (i.e., domain function), D, by demonstrating a concrete application of the domain function. That is, the user describes 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 Sect. 3.2 captured a domain function, D, 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.

MTBD captures domain functions as transformations that can be applied to models that adhere to the metamodel of the target domain. The first step in MTBD is for a user to apply the domain function to a model so that the MTBD engine can capture the set of model modifications. The process begins by the user or an external signal initiating a recording process. During the recording process, the user demonstrates the domain function on a specific model instance.

The domain function D takes an initial model M as input and produces a new model, M', as output (i.e., $M' = D(M)$). Although it is possible that M and M' are not conformant to the same metamodel, this paper explicitly focuses and enforces this assumption of an endogenous transformation. In MTBD, a model is a typed directed graph that can contain cycles. A model M consists of one or more *Nodes* (i.e., a typed object element—E), *Connections* (i.e., the link connecting two *Nodes*—C), and *Attributes* (i.e., the properties associated with *Nodes* or *Connections*—A). The models that we are transforming are based on EMF as used by GEMS.

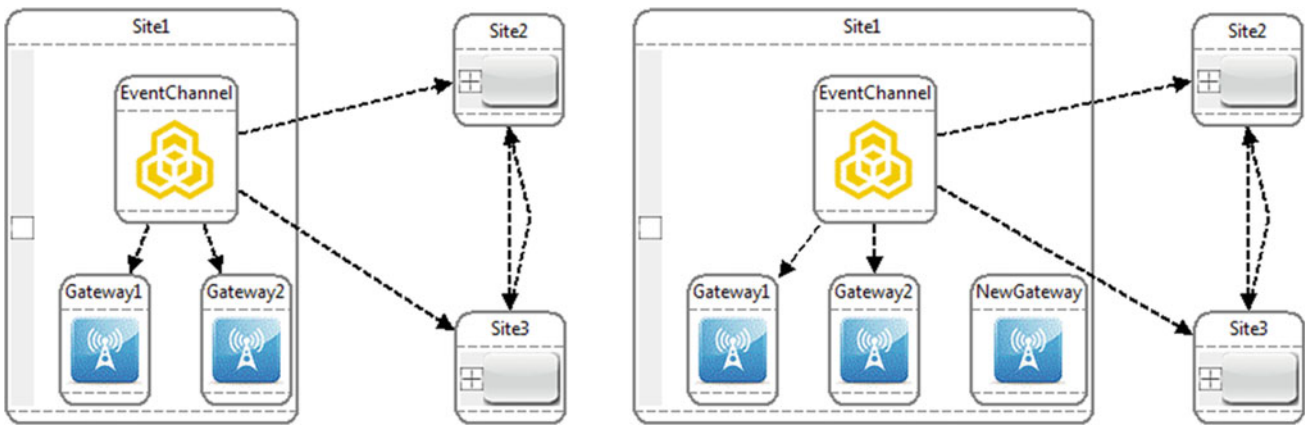


Fig. 5 The EQAL model before (*left*) and after (*right*) the demonstration

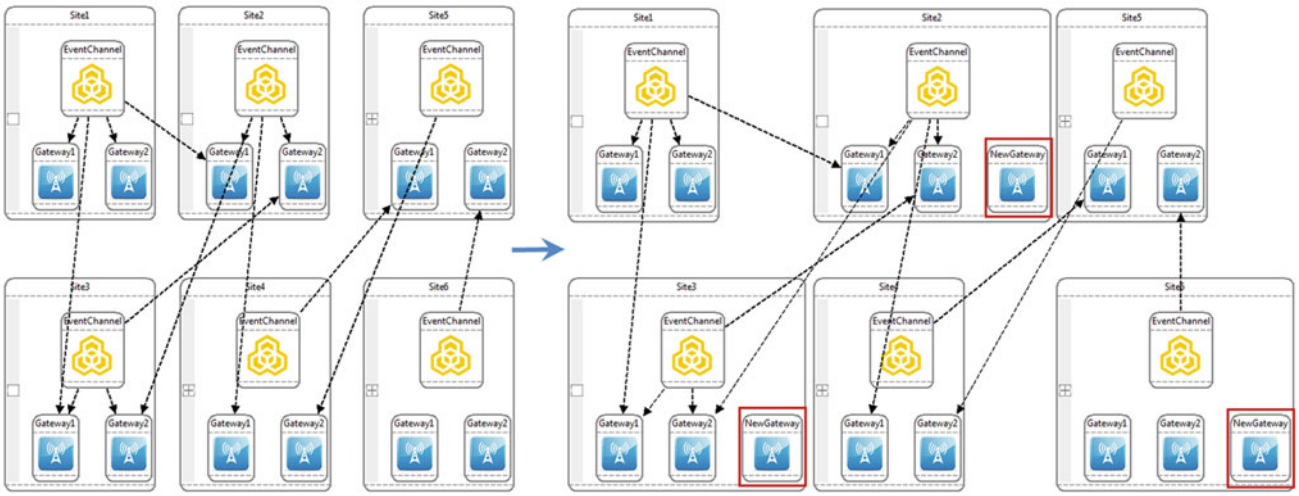


Fig. 6 An EQAL model before (*left*) and after (*right*) applying the inferred scaling transformation pattern

4.1.2 Pattern inference

After the recording process, the MTBD engine processes a series of model modifications, T, that express the application of the domain function 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 recorded model modifications, T, are expressed in terms of a specific model, M. However, the transformation must be generalized and expressed in terms of the general metamodel notations captured in the metamodel, rather than a specific model's elements. MTBD generalizes T as T' through an inference process. The inference step produces a model transformation, which we describe as a tuple:

$$Transformation = \langle P', T' \rangle \tag{1}$$

where P' is a set of generalized preconditions 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, P' describes the domain knowledge regarding the circumstances in which D can be applied, and T' defines what to do when these circumstances are met. For example, in the cloud computing example from Sect. 3.3, 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.

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*, is described as a tuple representing the types of model elements that are encountered if traversing from a specific element to the root of the model:

$$Pe =< Pe1, Pe2, \ldots, PeN > \tag{2}$$

Assume that an element $E1$ in the Model M is 4 levels away from the root. A hierarchical precondition, *Pe*, would be a 4-tuple and describes the types of ancestors of $E1$, where *Pe0* is the type of e, *Pe1* is the type of e's parent, *Pe2* is the type of e's parent's parent, and so forth. In order for this precondition to hold in an arbitrary model, an element must exist with type *Pe0*, where each ancestor K levels further up in the tree of model elements exactly matches the corresponding type *PeK*. For example, the parent of the parent of an element $E1$ would need to match the type *Pe2*.

A connection precondition is another form of a structural precondition. Connection preconditions dictate the C where *Pse* specifies a precondition, such as a hierarchical precondition, that must be met for an element to be considered the source element of a connection to be modified; *Pte* is a precondition that must be met for an element to be considered

the target element of the connection; and Tc is the type of connection that must exist between the elements that satisfy the source and target structural preconditions.

The inference function evaluates each change that was recorded from the user's application of the domain function to M. From these changes, structural preconditions are extracted as follows:

- **Added Elements** For each model element $E2$ that is added to an element $E1$ in the model, a precondition, Pe, is created that describes $E1$ and its ancestors.
- **Removed Elements** If an element, $E1$, is removed from the model, a precondition, Pe, is created describing $E1$.
- **Added Connections** Each new connection, $C1$, which is added from model element E1 to E2, produces a new precondition, Pc. The source precondition, Pse, of Pc is created by generating a hierarchical precondition describing $E1$ and its ancestors. The target precondition, Pte, of Pc is created by generating a hierarchical precondition for $E2$. Finally, because a new connection is being created and an existing connection of a specific type is not needed, Tc is set to a type that matches any or no existing connections between the elements.
- **Removed Connections** Each deleted connection, $C1$, that previously started from model element $E1$ and ended at model element $E2$ produces a new precondition, Pc. The Pse and Pte preconditions of Pc are generated as described above for added connections. Tc is created by generating a precondition describing the type of $C1$.
- **Changed Attributes** If an element, $E1$, has an attribute value changed, a precondition, Pe, is created. Pe describes the type of $E1$ and its ancestors.

Attribute preconditions Attribute preconditions specify the required values of properties on elements that a transformation will be applied to. An attribute precondition, Ac, is specified as a tuple:

$$Ac =< Pe, Expr > \tag{3}$$

where Pe 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. 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.

4.1.3 Manual inference refinement

The goal of MTBD is to generate a transformation, T', that faithfully represents the domain function. However, in many circumstances, the model that the function is demonstrated on may lack sufficient information to infer preconditions accurately. For example, in the cloud computing example from Sect. 3.3, 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 the domain function. The optional manual inference step allows the user to view and modify the inferred transformation and preconditions produced by the inference. The following section describes in detail the need for a manual refinement step.

4.2 Limitations of original MTBD to support model scalability

Although the example in Sect. 4.1 is simple, it shows the potential for assisting 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 executed correctly. In the previous example, the precondition inferred (List 2) indicates 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 additional 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 Fig. 7); or users may want to add the new *Gateway* only if the *Site*

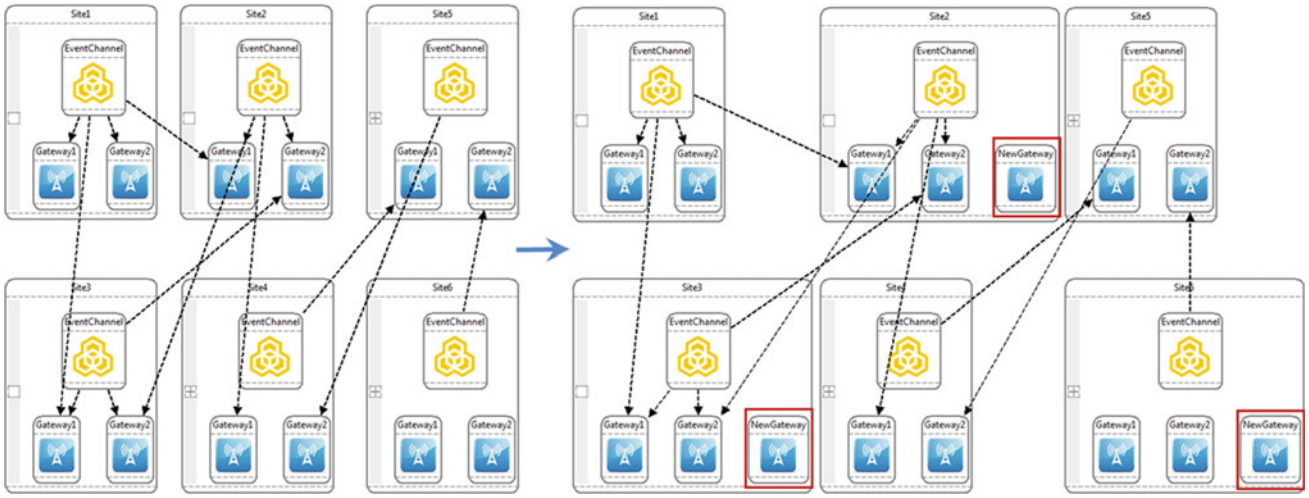


Fig. 7 Scaling specific locations based on preconditions

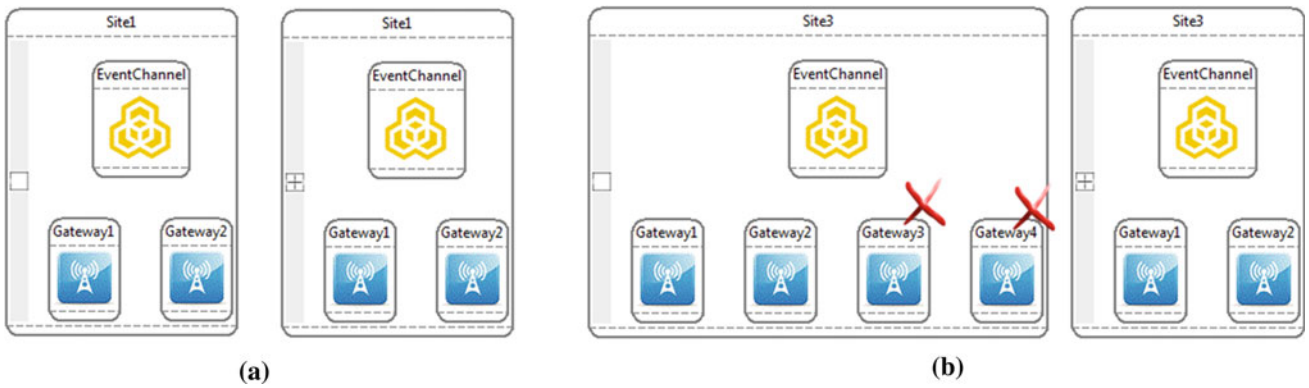


Fig. 8 The inferred transformation actions are not generic. **a** Replicate a *Site* in a demonstration. **b** The inferred pattern failed to replicate all *Gateways*

has no outgoing and incoming connections from it. These kinds of specific precondition requirements are needed frequently 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 3.3 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.

The inferred transformation actions are not generic In addition to 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 may not be generic enough to reflect a user's real transformation intention. For instance, a user may want to replicate a *Site* (e.g., *Site1* in Fig. 8a) that contains an *EventChannel* and two *Gateways*. This would require that the operations in List 4 be performed in the demonstration.

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 Fig. 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 Fig. 8b is the *Site* created after executing the inferred replication pattern). This situation emerges 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* of 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

List 4 Operations performed to replicate a *Site*

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

satisfy the weakest precondition due to a lack of sufficient *Gateways* to execute the two replicating operations in the demonstration.

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 3.1 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 3.1 requires the creation of a new 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 3.1 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 MT-Scribe, when applying a generated transformation pattern, only a single pattern could be selected to execute just 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 in 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.

4.3 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. The extensions made to MT-Scribe to address the model scalability needs are summarized in this section.

A user refinement step to specify preconditions Inferring the specific preconditions from only the demonstration can be 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 inference 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 (Fig. 9).

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 find the elements easily for which he or she wants 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 Sect. 4.1, it can be observed that the reason an inferred transformation pattern

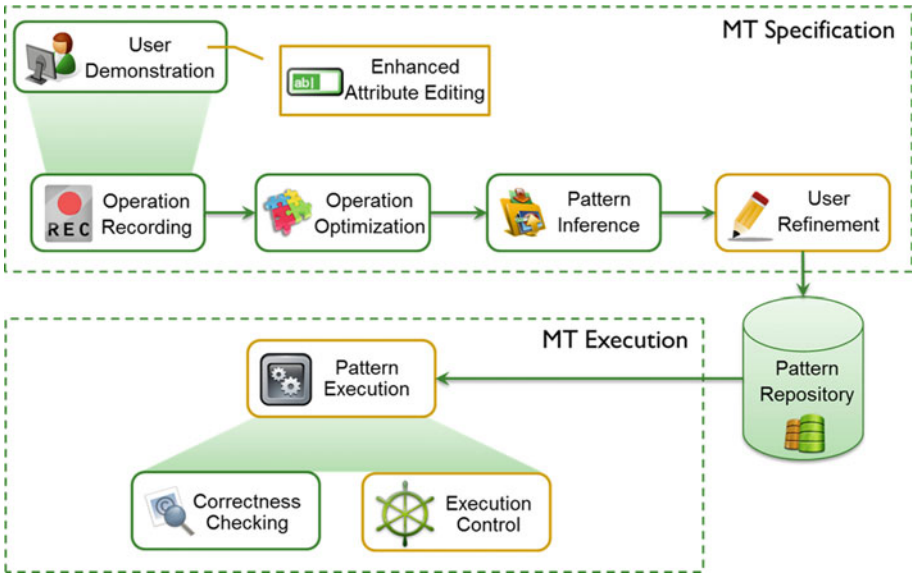


Fig. 9 Overview of extended MTBD

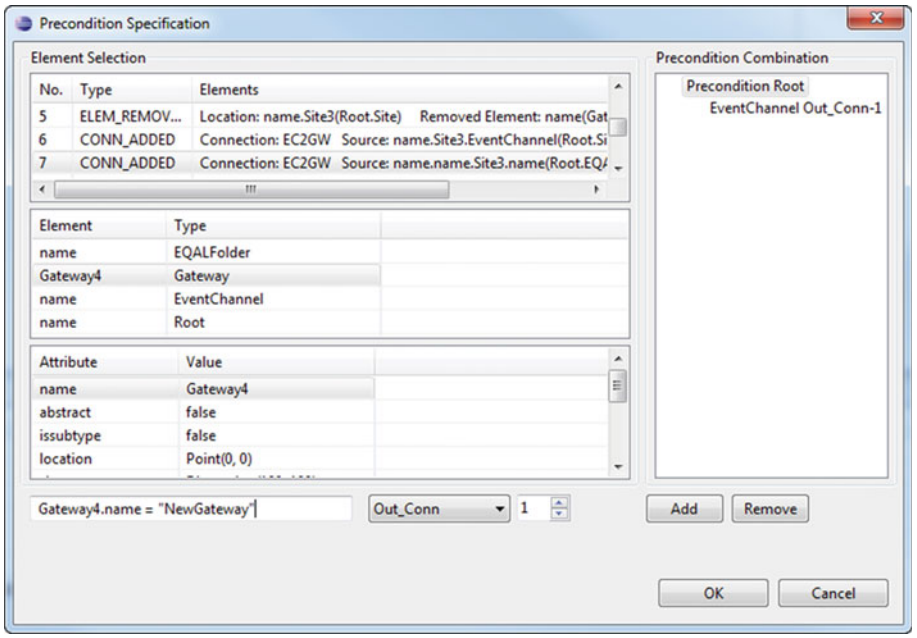


Fig. 10 Precondition specification dialog in MT-Scribe

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 operations need to be generic (i.e., an operation needs to be iterated and repeatedly executed for different times according to the number of available elements), a demonstration only needs to be done once, followed by clear identification of the operation(s) as generic and repeatable.

Figure 11 shows the generic operations identification dialog. It simply lists all the operations performed during the demonstration process. Users may identify the generic operations 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 3.2, instead of performing the operations listed in List 4, the user should demonstrate the tasks as specified in List 5.

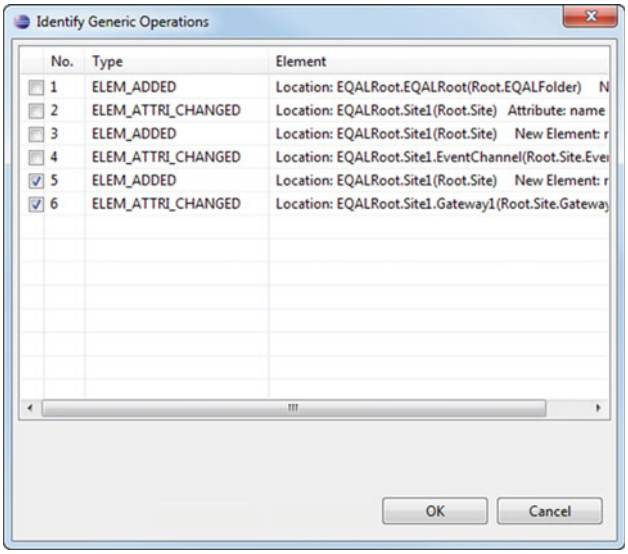


Fig. 11 Generic operations identification dialog

List 5 Demonstrate generic operations only once

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

After performing the above demonstration, the user must then mark operations 5 and 6 as generic (as shown in Fig. 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 continue to 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 two 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 (e.g., `subString()`, `indexOf()`) can be used to specify the computation. In the new implementation of MT-Scribe, we applied the dynamic language Groovy [16] 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 number of 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 Sect. 3.

5 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 Sect. 3. 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 once, scale multiple times."

Given a model scalability task, the main steps of a solution are as follows: (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.

Based on the new extended MTBD framework shown in Fig. 9, a complete iteration using MTBD to solve a model scalability task is as follows. First, users start with the demonstration to simulate the one degree scaling task (*User Demonstration*). During the demonstration, users are expected to perform operations not only on adding or replicating model elements and connections, but also on their attributes (*Enhanced Attribute Editing*). At the same time, the event listener monitors all the operations occurring in the model editor and collects the information for each operation in sequence (*Operation Recording*). After the demonstration, the engine optimizes the recorded operations to eliminate any duplicated or meaningless actions (*Operation Optimization*). With an optimized list of recorded operations, the transformation can be inferred by generalizing the behavior in the demonstration (*Pattern Inference*). Users are also enabled to refine the generated transformation pattern by providing more feedback for the precondition of the desired transformation scenario from two perspectives—structure and attributes, or identifying generic operations to be executed repeatedly according to the available model elements and connections (*User Refinement*). After the user refinement, the transformation pattern will be finalized and stored in the pattern repository for future use (*Pattern Repository*). The final patterns in the repository can be executed on any model instances. The execution starts with matching the precondition in the new model instance and then carrying out the transformation actions on the matched locations of the model (*Pattern Execution*). The MTBD inference engine also validates the correctness of the models after each execution process (*Correctness Checking*). Users can choose where to execute the pattern, a sequence of patterns to execute, and the execution times (*Execution Control*).

The remainder of this section provides solutions to the scalability examples of Sect. 3 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 [24]. At the end of this section, we also briefly present the main implementation challenges and approaches used in the tool.

5.1 Scaling SRN models

By analyzing the scalability needs of Example 3.1, 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 Fig. 12:

t1. Create a new set of places, transitions, and connections for the new event type. Specify proper names for each element 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 provide this demonstration, we choose the two-event SRN model as shown in the top of Fig. 1. Then, we manually edit the model and demonstrate the three sub-tasks. To demonstrate t1, the operations identified in List 6 are performed.

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 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 Fig. 12 (sub-task 1).

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 Fig. 12. Figure 12 (sub-task t2) shows the model after these operations.

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 Sect. 4.3, 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 inference engine will generate the correct transformation pattern to repeat these operations when needed. The operations performed are shown. We select *SnpLnProg2* as the existing snapshot place and demonstrate the creation of snapshot transitions *TProcSnp2,3* and *TProcSnp3,2*.

When specifying the name attributes, complex String composition can be given using the Java API, as done in operations 29 and 31 (the only concept that the user needs to

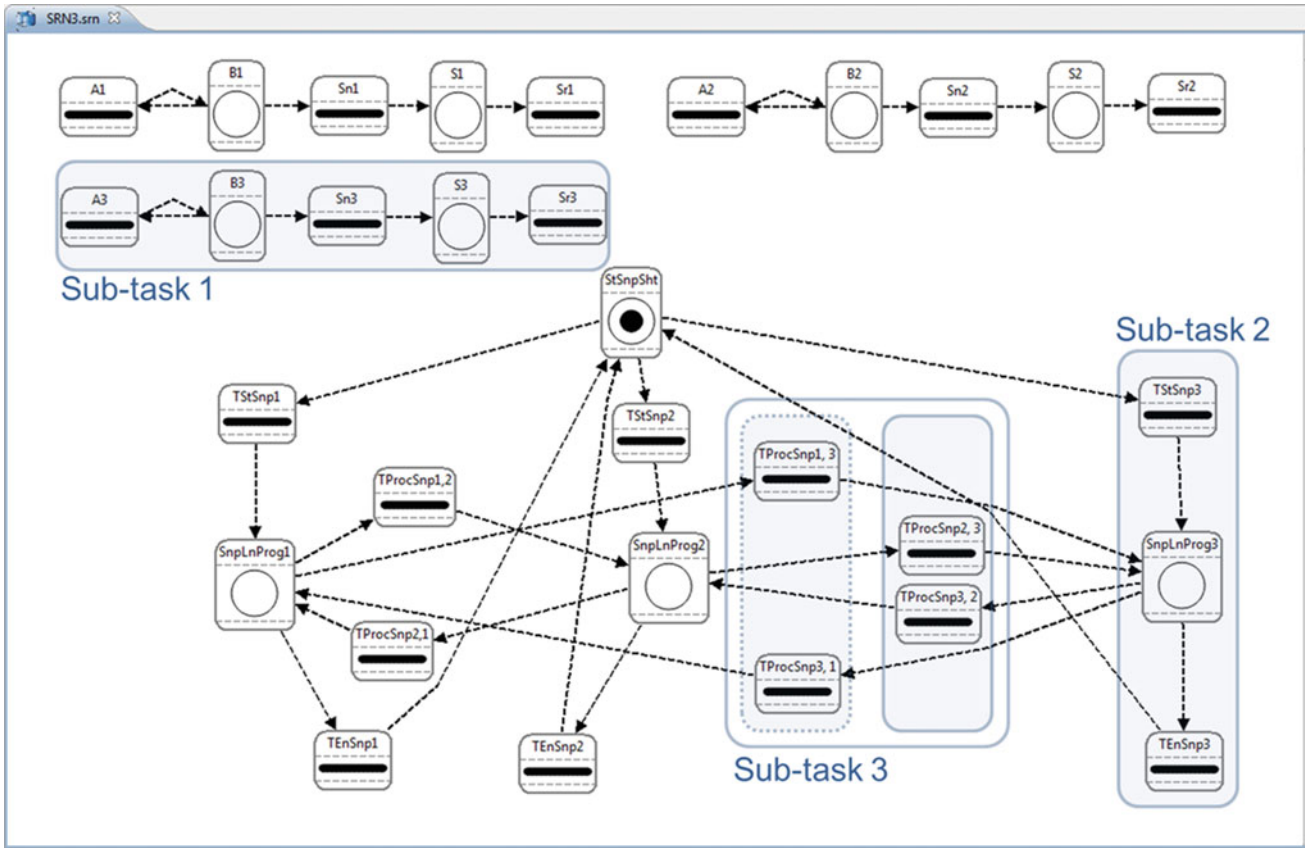


Fig. 12 The process of scaling a SRN model from two events to three events

List 6 Operations for sub-task t1 of Example 3.1

Sequence	Operation performed
1	Add a *Transition* in *SRNRoot*
2	Create an artificial name with the value: *EventName = "3"*
3	Set the new*Transition.name = "A" + EventName = "A3"*
4	Add a *Place* in *SRNRoot*
5	Set the new *Place.name = "B" + EventName = "B3"*
6	Add a *Transition* in *SRNRoot*
7	Set the new *Transition.name = "Sn" + EventName = "Sn3"*
8	Add a *Place* in *SRNRoot*
9	Set the new *Place.name = "S" + EventName = "S3"*
10	Add a *Transition* in *SRNRoot*
11	Set the new *Transition.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*

understand is the meaning of subString). 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 Fig. 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 Fig. 1 is the result of adding two event types using the inferred pattern (Lists 7, 8).

5.2 Scaling the EQAL models

Example 3.2 focuses on increasing event services. The scaling process of adding one more event service includes four sub-tasks, as illustrated in Fig. 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.

List 7 Operations for sub-task t2 of Example 3.1

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

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.*

We give the demonstration on the model instance shown in the top of Fig. 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 a generic operation, 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 make the connection in one of the *Sites*, and then, identify them as generic. The operations performed for t1 are shown in List 9.

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.

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.

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).

After the demonstration, generic operations are identified, and preconditions are given in the user refinement step through the interface shown in Fig. 10. 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 Fig. 2 is the result of applying the inferred pattern to scale the model by adding three new event services.

5.3 Scaling the C2M2L models

Replicating a *Node* in Example 3.3 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*.

List 8 Operations for sub-task 3 of Example 3.1

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

* Represents generic operations to be identified

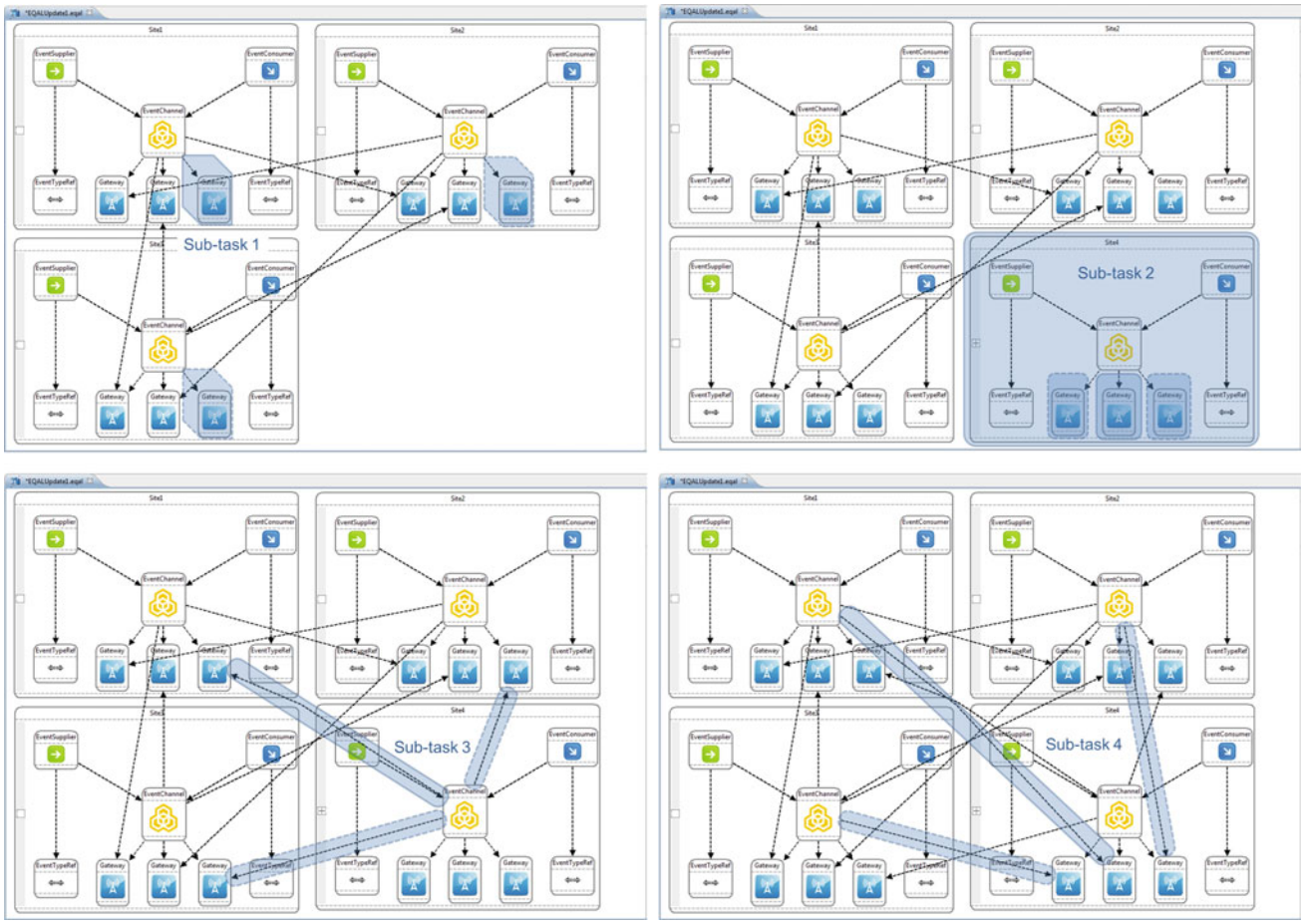


Fig. 13 The process of scaling an EQAL model from three event services to four

List 9 Operation for sub-task t1 of Example 3.2

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

* Represents generic operations to be identified

List 10 Operations for sub-task t2 of Example 3.2

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

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.

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

List 11 Operations for sub-task t3 of Example 3.2 ("p" represents the precondition)

Sequence	Operation performed
15*	Connect *Site.EventChannel* and *Site1.Gateway*
	^{p1}*Site.EventChannel.outgoingConns* = 0
	^{p2}*Site.EventChannel.incomingConns* = 0
	^{p3}*Site1.Gateway.outgoingConns* = 0
	^{p4}*Site1.Gateway.incomingConns* = 1

List 12 Operations for sub-task t4 of Example 3.2

Sequence	Operation performed
16*	Connect *Site1.EventChannel* and *Site.Gateway*
	^{p1}*Site.Gateway.outgoingConns* = 0
	^{p2}*Site.Gateway.incomingConns* = 1

5.4 Implementation challenges

The implementation of MTBD is realized in MT-Scribe [24], which is an Eclipse plugin based on GEMS [11]. With the main functionalities of MTBD being the specification of a model transformation and the execution of a model transformation, the main challenges in the implementation of MT-Scribe are how to preserve all the information about precondition and transformation actions in a pattern and how to match the pattern in order to carry out the execution.

GEMS provides a set of rich APIs to access and manipulate the model, as well as several event listeners to capture the actions performed by the user in the editing environment, so that we can directly record all the needed information about a demonstration. The inference process takes the list of recorded operations as the input and generates two tables as the output of the initial transformation pattern. The first table records the precondition, with the key of the table as the required model element with its containment relationship as shown in the left of List 2 and the value as its corresponding metatypes. This table defines the required structural precondition, and the attribute preconditions for each element are converted as a list attached to each table entry. The transformation actions are specified in the second table in sequential order, with the key as the operation type and the value to be the required operands based on the precondition table entries. The two tables correlate with each other and serve as the representation for the transformation pattern during the whole specification process. With the new inputs taken from the user refinement step, the associated table entries will be updated or extended accordingly.

List 13 Operations for sub-task t1 of Example 3.3

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

List 14 Operations for sub-task t2 of Example 3.3

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 finalized pattern (i.e., two tables) is persisted by serializing the table objects so that they can be loaded directly into the execution engine later. The execution engine is responsible for a series of tasks: (1) load the pattern, (2) match the precondition, (3) execute each transformation action and check the correctness, and (4) rollback and restore the initial state if an error occurs. The main challenge lies on the precondition matching process. We traverse all the available model elements and connections in the current model instance and use each of them as the candidate for the required element in the precondition table. The candidate is matched when its metatype and the containment relationship conform to the structural precondition, as well as when the list of attribute preconditions are satisfied. If there is no match, a backtracking algorithm is applied to try out the next combination. The actual execution of each action is implemented using the Command pattern so that undo can be performed easily if a rollback is needed.

6 Evaluation

In this section, the extended MTBD approach is first evaluated using the three desired characteristics of model scalability described in [22]. 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.

6.1 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 initially in [22].

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 use MT-Scribe 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 in 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 is not familiar with model transformation languages, the idea of user extension by altering the model transformation is not even possible.

6.2 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 [22]. In our earlier work, we used a model transformation engine called C-SAW, which processed a transformation language called the Embedded Constraint Language (ECL) [13]. We used ECL to perform the same model scalability tasks that were introduced in Sects. 3.1 and 3.2 (in this paper, Sect. 3.3 describes an example that was not considered in the previous scalability studies that focused on using MTLs). 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 [22], 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

```
//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);
}
```

Fig. 14 An excerpt of a transformation rule written in ECL to accomplish sub-task t1 of Example 3.2

```
//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;
}
```

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

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, Fig. 14 shows part of the model transformation rules written in ECL to implement sub-task t1 of Example 3.2. To add a Gateway, the necessary objects should be declared first, followed by calling creational APIs to construct 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.

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 Fig. 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.

We have not conducted a formal human-based user study on the comparison between the two approaches. However,

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

Model scalability example	MTBD	ECL rules
Example 3.1	35 operations	170 SLOC
	1 generic operation refinement	
Example 3.2	16 operations	124 SLOC
	2 precondition refinement	
	1 generic operation refinement	

Table 1 lists some of the results of the comparative effort, indicating that [22] used over 170 lines of ECL code to address Example 3.1. 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 3.2 using ECL requires 32 lines of ECL [14], while our MTBD-based scalability solution required 16 direct editing operations, one generic operation identification and two precondition refinements.

Table 2 The comparison of development time to solve model scalability tasks using MTBD and a model transformation language (ECL); performed by experts in both cases

Example	MTBD	ECL rules
Example 3.1	1.5 s for demonstration 1 min for refinement	30 min
Example 3.2	40 s for demonstration 2 min for refinement	25 min

To better measure the effort, we asked two users to accomplish the two tasks using these two approaches so that we could measure the time spent using each approach. The user who used MTBD was a trained MTBD user, while the other user who wrote the transformation rules is one of the ECL coauthors. The result in Table 2 shows that using MTBD to solve these two examples is 10 times faster than writing C-SAW rules. This fast development cycle enables users to quickly test patterns and make modifications. For instance, in order to determine whether a generated pattern is too general or not, users need to execute the pattern and test the result. In some cases, modifications have to be made to fix errors followed by re-executing the new pattern. MTBD shortens these development iterations. This comparison is made between computer scientists who actually understand both approaches; the MTL approach with C-SAW is not even possible with most end users who are not computer scientists.

Table 3 describes the prerequisite knowledge for each approach. For MTBD, the development environment is integrated in the modeling tool itself; thus, users only need to know a few additional buttons to use MTBD. The demonstration is performed using the same model editing operations, so the only new things to learn are the UIs for pattern refinements. For ECL, the development environment is a plug-in to GME [21], and it requires a regular textual editor to specify the transformation rules. Learning ECL requires learning the new syntax and semantics of the language, including a number of new keywords, program structures, and functions. Finally, we identified the possible errors that could happen in the development process using the two approaches. For MTBD, users might perform incorrect editing operations or give invalid expressions in the precondition specification and refinement. The base pattern and minimum preconditions are automatically generated at the background, so there is no way that users can make mistakes on this part. For C-SAW or nearly any other textual or visual transformation programming language, the transformation rules could suffer from general programming issues such as the syntax errors and semantics bugs in the logical expression of the transformation task.

6.3 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 specified easily 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 (Table 4).

Therefore, since it is not easy to make MTBD a fully complete replacement to a well-defined model transforma-

Table 3 The comparison of prerequisite knowledge to solve model scalability tasks using MTBD and a model transformation language (ECL)

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

Table 4 The comparison of possible errors to solve model scalability tasks using MTBD and a model transformation language (ECL)

MTBD	ECL rules
1. Demonstrate with the wrong editing operation	1. General programming language syntax error
2. The incorrect specification on the attribute precondition expressions or generic operations	2. Semantics error or bugs on the transformation logic

tion 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 were 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.

In addition, the current inference is based on a single demonstration from users, rather than a series of demonstrations for different scenarios. Although a single demonstration requires much less effort from users, it often contains limited information about the desired scenario, restricting the accuracy of the transformation pattern being inferred (e.g., negative preconditions cannot be inferred from a single positive demonstration). Thus, the desired number of demonstrations given by users as input to the inference engine is another issue that needs to be investigated further. Apart from the insufficient information on a precondition in a single demonstration, the generic operation refinement step also has some limitations. In the current execution engine, we have explicitly excluded the container elements from the candidate pool after each matching process in order to avoid the infinite loop of adding new elements when the add operation is identified as generic. The drawback is that a single generic adding operation cannot be used to add multiple elements in the same container. Additionally, the generic operation will be executed until no more operands are available, which means that we cannot control the generic operations to be matched and executed for a certain number of times, or based on certain conditions.

The correctness of the inferred transformation pattern currently depends on the correctness of the demonstration. An issue has emerged on how to help users to make sure whether the generated pattern is desired. Finally, the question about how to debug the transformation process at an end-user level is another task to be considered in order to ensure the quality of the models after being scaled. We have already initiated a new line of work that has developed a prototype debugger for MT-Scribe [35].

The limited evaluation and assessment of the approach is also an area of future work that we are planning to explore. The comparisons made in Sect. 6.2 were performed with just two users who were experienced in both techniques. A more detailed study between end users in targeted domains, who may also not be computer scientists, is also needed to assess the additional benefits of MTBD as applied to scalability tasks. Although the scalability tasks described in this paper are generally not even possible with end users using traditional approaches, we would still like to understand more of the gaps in the capability of MT-Scribe to better assist such users. To this end, we have planned a new experimental human-based study that will be reported on the MTBD web page when completed [24].

7 Conclusion

This paper introduced 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 [14,22], we believe that MTBD offers several advantages supporting ease of use for end users who are domain experts, but do not have programming experience. The demonstration focus allows users to be ignorant of both the details of the transformation language and the structure of the metamodel for the language being used. The paper presented a new approach to MTBD and summarized the changes that were needed to evolve a tool (MT-Scribe) that supported an earlier version of our approach 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, as well as showing the new extensions made in MT-Scribe. 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. Of course, this approach is also applicable in other model transformation areas such as aspect-oriented modeling and model refactoring. However, we believe that highlighting model scalability helps to promote the overall benefits of MDE for adoption in new domains that require frequent engineering changes to be explored in the context of scalability.

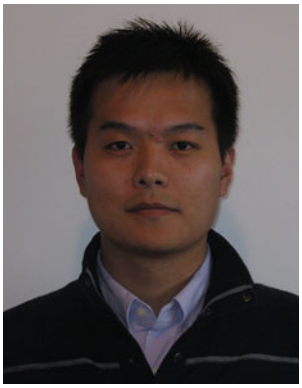
Acknowledgments This work was supported by NSF CAREER Award CCF-1052616.

References

1. Agrawal, A., Karsai, G., Lédeczi, Á.: An end-to-end domain-driven software development framework. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA): Domain-driven Track, pp. 8–15, Anaheim, CA (2003)
2. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/ (2012)
3. Balasubramanian, D., Narayanan, A., Buskirk, C., Karsai, G.: The graph rewriting and transformation language: GreAT. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **1**, 8 (2006)
4. Balogh, Z., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: Symposium on Applied

Computing (SAC), pp. 1280–1287, Dijon, France, April 2006 (2006)
5. Bondi, A.: Characteristics of scalability and their impact on performance. In: 2nd International Workshop on Software and Performance, pp. 195–203, Ottawa, Ontario, Canada (2000)
6. 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. In: International Conference on Model Driven Engineering Languages and Systems (MoDELS), pp. 271–285, Spring-Verlag, LNCS 5795, Denver, CO., October 2009 (2009)
7. Brosch, P., Seidl, M., Wieland, K., Wimmer, M., Langer, P.: The operation recorder: specifying model refactorings by-example. In: International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA): Tool Demonstration, pp. 791–792, Orlando, FL, October 2009 (2009)
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
9. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. Formal Methods Syst. Des. **19**(1), 45–80 (2001)
10. Edwards, G., Deng, G., Schmidt, D., Gokhale, A., Natarajan, B.: Model-driven configuration and deployment of component middleware publish/subscribe Services. In: Generative Programming and Component Engineering (GPCE), pp. 337–360, Vancouver, BC, October 2004 (2004)
11. Generic Eclipse Modeling System (GEMS). http://www.eclipse.org/gmt/gems/
12. Gray, J., Zhang, J., Lin, Y., Wu, H., Roychoudhury, S., Sudarsan, R., Gokhale, A., Neema, S., Shi, F., Bapty, T.: Model-driven program transformation of a large avionics framework. In: Generative Programming and Component Engineering (GPCE), pp. 361–378, Springer, LNCS 3286, Vancouver, BC, October 2004 (2004)
13. Gray, J., Lin, Y., Zhang, J.: Automating change evolution in model-driven engineering. IEEE Comput. Special Issue Model Driven Eng. **39**(2), 51–58 (2006)
14. 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), pp. 295–308, Springer, LNCS 3713, Montego Bay, Jamaica, October 2005 (2005)
15. Gray, J., Tolvanen, J., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-specific modeling. In: Handbook of Dynamic System Modeling, Chapter 7, pp. 7–1 through 7–20. CRC Press, Boca Raton (2007)
16. Groovy. http://groovy.codehaus.org/
17. Hayes, B.: Cloud computing. Commun. of the ACM **51**(7), 9–11 (2008)
18. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1/2), 31–39 (2008)
19. 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)
20. 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. In: International Conference on Model-Driven Engineering Languages and Systems (MoDELS), pp. 528–542, Springer, LNCS 4199, Genova, Italy, October 2006 (2006)
21. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. IEEE. Comput. **34**(11), 44–51 (2001)
22. Lin, Y., Gray, J., Zhang, J., Nordstrom, S., Gokhale, A., Neema, S., Gokhale, S.: Model replication: transformations to address model scalability. Softw. Pract. Experience **38**(14), 1475–1497 (2008)
23. Mens, T., Gorp, P.: A taxonomy of model transformation. Workshop on Graph and Model Transformation, vol. 152, pp. 125–142, Talinn, Estonia (2005)
24. MTBD Project Page. http://students.cis.uab.edu/yusun/mtbd/
25. Muppala, J., Ciardo, G., Trivedi, K.: Stochastic reward nets for reliability prediction. Commun. Reliab. Maintainab. Serv. **1**(2), 9–20 (1994)
26. OMG, Revised Submission for MOF 2.0 Query/View/Transformations RFP (ad/2002-04-10), OMG Document ad/2005-07-01 (2005)
27. Object Management Group, Object Constraint Language Specification. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL (2012)
28. Schmidt, D.: Model-driven engineering. IEEE. Comput. **39**(2), 25–32 (2006)
29. Schmidt, D., Stal, M., Rohnert, H., Buschman, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, vol. 2. Wiley, New York (2000)
30. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw Special Issue Model Driven Softw. Dev. **20**(5), 42–45 (2003)
31. Strommer, M., Wimmer, M.: A framework for model transformation by-example: concepts and tool support. In: 46th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS), pp. 372–391, Zurich, Switzerland, July 2008 (2008)
32. Sun, Y., White, J., Gray, J.: Model transformation by demonstration. In: Model Driven Engineering Languages and Systems (MoDELS), pp. 712–726, Spring-Verlag, LNCS 5795, Denver, CO., October 2009 (2009)
33. Sun, Y., White, J., Gray, J., Gokhale, A.: Model-driven automated error recovery in cloud computing. In: Model-Driven Analysis and Software Development: Architectures and Functions, IGI Global, Hershey, PA (2009)
34. Sun, Y., Gray, J., Bulheller, K., von Baillou, N.: A model-driven approach to support engineering changes in industrial robotics software. In: Model Driven Engineering Languages and Systems (MoDELS), pp. 368–382, Springer, LNCS 7590, Innsbruck, Austria, October 2012 (2012)
35. Sun, Y., Gray, J.: Model transformation by demonstration debugger: end-user support for debugging model transformation execution. In: European Conference on Modeling Foundations and Applications (ECMFA), pp. 86–100, Springer, LNCS 7949, Montpellier, France, July 2013 (2013)
36. Varró, D.: Model transformation by example. In: Model-driven engineering languages and systems (MoDELS), pp. 410–424, Springer, LNCS 4199, Genova, Italy, October 2006 (2006)
37. Varró, D., Balogh, Z.: Automating model transformation by example using inductive logic programming. In: Symposium on Applied Computing (SAC), pp. 978–984, Seoul, Korea, March 2007 (2007)
38. W3C, XSLT Transformation version 1.0. http://www.w3.org/TR/xslt (1999)
39. White, J., Czarnecki, K., Schmidt, D., Lenz, G., Wienands, C., Wuchner, E., Fiege, L.: Automated model-based configuration of enterprise java applications. In: Enterprise Distributed Object Computing (EDOC), pp. 301–312, Annapolis, Maryland, October 2007 (2007)
40. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: Hawaii International Conference on System Sciences (HICSS), pp. 285, Big Island, HI, January 2007 (2007)

Author Biographies



Yu Sun received his PhD from the University of Alabama at Birmingham in 2011. His research focuses on model-driven engineering, and specifically supporting model evolution process in a user-centric approach. Yu currently works for a startup company—PAR Works, as the director of engineering. The company provides high-precision 3D augmented reality technology for mobile platforms.

Jeff Gray is an Associate Professor in the Department of Computer Science at the University of Alabama. He co-directs the Software Modeling Lab, which focuses on research in areas related to model-driven engineering, DSLS, and software maintenance. He also conducts research and outreach in computer science education, with particular emphasis on grades K-12.

Jules White is an Assistant Professor of Computer Science in the Department of Electrical Engineering and Computer Science at Vanderbilt University. His research has won 4 Best Paper Awards. He has also published over 80 papers. Dr. White's research focuses on securing, optimizing, and leveraging data from mobile cyber-physical systems. His mobile cyber-physical systems research spans four key focus areas: (1) mobile security and data collection, (2) high-precision mobile augmented reality, (3) mobile device and supporting cloud infrastructure power and configuration optimization, and (4) applications of mobile cyber-physical systems in multi-disciplinary domains, including energy-optimized cloud computing, smart grid systems, healthcare/manufacturing security, next-generation construction technologies, and citizen science. His research has been transitioned to industry, where it won an Innovation Award at CES 2013, was a finalist for the Technical Achievement at Award at SXSW Interactive, and was a top 3 for mobile in the Accelerator Awards at SXSW 2013.