Model Replication: Transformations to Address Model Scalability

Yuehua Lin[1], Jeff Gray[1], Jing Zhang[1], Steve Nordstrom[2],
Aniruddha Gokhale[2], Sandeep Neema[2], and Swapna Gokhale[3]

[1] Dept. of Computer and Information Sciences, University of Alabama at Birmingham
Birmingham AL 35294-1170
{liny, gray, zhangj} @ cis.uab.edu

[2] Institute for Software Integrated Systems, Vanderbilt University
Nashville TN 37235
{steve-o, gokhale, sandeep} @ isis.vanderbilt.edu

[3] Dept. of Computer Science and Engineering, University of Connecticut
Storrs CT 06269
ssg @ engr.uconn.edu

SUMMARY

In Model-Driven Engineering, it is often desirable to evaluate different design alternatives as they relate to scalability issues of the modeled system. A typical approach to address scalability is model replication, which starts by creating base models that capture the key entities as model elements and their relationships as model connections. A collection of base models can be adorned with necessary information to characterize a specific scalability concern as it relates to how the base modeling elements are replicated and connected together. In current modeling practice, such model replication is usually accomplished by scaling the base model manually. This is a time-consuming process that represents a source of error, especially when there are deep interactions between model components. As an alternative to the manual process, this paper presents the idea of automated model replication through a model transformation process that expands the number of elements from the base model and makes the correct connections among the generated modeling elements. The paper motivates the need for model replication through case studies taken from models supporting different domains.

Keywords: *model transformation, scalability, domain-specific modeling*

1. INTRODUCTION

Model-Driven Engineering (MDE) has emerged as a new approach to software development by raising the specification of software to higher level models [32]. A powerful justification for the use of models concerns the flexibility of system analysis, i.e., system analysis can be performed while exploring various design alternatives. This is particularly true for distributed real-time and embedded (DRE) systems, which have many properties that are often conflicting (e.g., battery consumption versus memory size), where the analysis of system properties is often best provided at higher levels of abstraction [17]. A typical form of design exploration involves experimenting with model structures by expanding different portions of models and analyzing the result on scalability [14, 15]. For example, a network engineer may create

various models to study the effect on network performance when moving from 2 routers to 8 routers, and then to 32 routers. We define model scalability as the ability to build a complex model from a base model by replicating its elements or substructures and adding the necessary connections. To support model scalability requires extensive support from the host modeling tool to enable rapid change evolution within the model representation [15]. However, it is difficult to achieve model scalability in current modeling practice due to the following challenges:

(1) Large-scale system models often contain many modeling elements: From our personal experience, models can have multiple thousands of coarse grained components (others have reported similar experience, please see [18]). Modeling these components using traditional manual model creation techniques and tools can approach the limits of the effective capability of humans. Particularly, the process of modeling a large DRE system with a domain-specific modeling language (DSML) [16], or a tool like MatLab, is different than traditional class-based UML modeling. In DRE systems modeling, the models consist of instances of all entities in the system, which can number into several thousand instances from a set of types defined in a metamodel (e.g., thousands of individual instantiations of a sensor type in a large sensor network model). Traditional UML models (e.g., UML class diagrams) are typically not concerned with the same type of instance-level focus, but instead specify the entities and their relationship of a system at design time (such as classes). This is not to imply that UML-based models do not have scalability issues (in fact, the UML community has recognized the importance of specifying instance models at a large-scale [8]), but the problem is more acute with system models built with DSMLs. The main reason is that system models are usually sent to an analysis tool (e.g., simulation tool) to explore system properties such as performance and security, they need to capture a system by including the instances of all the entities (such as objects) that occur at run-time, which leads to their larger size and nested hierarchy [16].

(2) Manually scaling up models is laborious, time consuming and prone to errors: To examine the effect of scalability on a system, the size of a system model (e.g., the number of the participant model elements and connections) needs to be increased or decreased frequently. The challenges of scalability affect the productivity of the modeling process, as well as the correctness of the model representation. As an example, consider a base model consisting of a few modeling elements and their corresponding connections. To scale a base model to hundreds, or even thousands of duplicated elements would require a lot of mouse clicking and typing within the associated modeling tool [15]. Furthermore, the tedious nature of manually replicating a base model may also be the source of many errors (e.g., forgetting to make a connection between two replicated modeling elements). A manual process to replication significantly hampers the ability to explore design alternatives within a model (e.g., after scaling a model to 800 modeling elements, it may be desired to scale back to only 500 elements, and then back up to 700 elements, in order to understand the impact of system size).

To address these challenges, this paper makes a contribution to model scalability by using a model transformation approach to automate replication[1] of base models. In MDE, model transformation is a process that converts one or more models to other artifacts (e.g., models, XML files or code). In this paper, we call a transformation for model replication a *replicator*, which changes a model to address scalability concerns. In our approach, large-scale system models are automatically created from smaller, baseline specification models by applying model transformation rules that govern the scaling and replication behavior associated with stepwise refinement of models [5]. The main benefit is increased productivity and accuracy in generating system models of various scale.

[1] The term "replication" has specific meaning in object replication of distributed systems and in database replication. In the context of this paper, the term is used to refer to the repetition of modeling elements or structures among models to address scalability concerns.

1.1. Supporting Technologies used in Our Approach to Model Scalability

The implementation of the scalability approach described in this paper is tied to a specific set of tools, but we believe the general idea can be applied to many toolsuite combinations. The modeling tool and model transformation engine used in our work are summarized in this section. The purpose of the paper is not to describe these in detail, but an introduction may help to understand the subsequent sections of the paper.

Model-Integrated Computing: A specific form of MDE, called Model-Integrated Computing (MIC) [35], has been refined at Vanderbilt University over the past decade to assist in the creation and synthesis of computer-based systems. A key application area for MIC is those domains (such as embedded systems areas typified by automotive and avionics systems) that tightly integrate the computational structure of a system and its physical configuration. In such systems, MIC has been shown to be a powerful tool for providing adaptability in frequently changing environments. The Generic Modeling Environment (GME) [1, 20] is a metamodeling tool based on MIC that can be configured and adapted from meta-level specifications (called the modeling paradigm) that describe the domain. Each metamodel describes a DSML, which is similar to a grammar that specifies a programming language [16]. When using the GME, a modeling paradigm is loaded into the tool to define an environment containing all the modeling elements and valid relationships that can be constructed in a specific domain. A model compiler can be written and invoked from within the GME as a plug-in to synthesize a model into some other form (e.g., translation to code, refinement to a different model, or simulation scripts). All of the DSMLs presented in this paper are defined and developed within the GME.

C-SAW - A Model Transformation Engine: The paper advocates automated model transformation to address scalability concerns. The Constraint-Specification Aspect Weaver (C-SAW) [2] is the model transformation engine used in the case studies in Section 3. Originally, C-SAW was designed to address

crosscutting modeling concerns [13], but has evolved into a general model transformation engine. C-SAW is a GME plug-in and is compatible with any metamodel; thus, it is domain-independent and can be used with any modeling language defined within the GME. The Embedded Constraint Language (ECL) is the language that we developed for C-SAW to specify transformations. The ECL is featured in our case studies on model transformation (Section 3) and briefly explained in Figures 5, 7, and 8.

1.2 Scope of the Paper

The rest of the paper is organized as follows: Section 2 provides an outline of the technical challenges of model replication and a summary of techniques that we have observed from our own experience. Two case studies of model scalability using replicators are provided in Section 3, with related work in model transformation summarized in Section 4. The conclusion offers summary remarks and a brief description of future work that addresses current limitations.

2. ALTERNATIVE APPROACHES TO MODEL SCALABILITY

This section provides a discussion of key characteristics of model replication to support scalability. An overview of existing replication approaches is presented and a comparison of each approach is made with respect to the desired characteristics. The section offers an initial justification of the benefits of model transformation to support scalability of models using replicators.

2.1 Desired Characteristics of a Replication Approach

An approach that supports model scalability through replication should have the following desirable characteristics: 1) retains the benefits of modeling, 2) is general across multiple modeling languages, and 3) is flexible to support user extensions. Each of these characteristics (C1 through C3) is discussed further in this subsection.

C1. Retains the benefits of modeling: As stated in Section 1, the power of modeling comes from the opportunity to explore various design alternatives and the ability to perform analysis (e.g., model checking and verification of system properties [17]) that would be difficult to achieve at the implementation level but easy at the model level. Thus, a model replication technique should not perform scalability in such a way that analysis and design exploration is inhibited. This seems to be an obvious characteristic to desire, but we have observed replication approaches that remove these fundamental benefits of modeling.

C2. General across multiple modeling languages: A replication technique that is generally applicable across multiple modeling languages can leverage the effort expended in creating the underlying transformation mechanism for a specific modeling language. A side benefit of such generality is that a class of users can become familiar with a common replicator technique, which can be applied to many modeling languages.

C3. Flexible to support user extensions: Often, large-scale system models leverage architectures that are already well-suited toward scalability [32]. Likewise, the modeling languages that specify such systems may embody similar patterns of scalability, and may lend themselves favorably toward a generative and reusable replication process. Further reuse can be realized if the replicator supports multiple types of scalability concerns in a templatized fashion (e.g., the name, type, and size of the elements to be scaled are parameters to the replicator). The most flexible type of replication would allow alteration of the semantics of the replication more directly using a language that can be manipulated easily by an end-user. In contrast, replicator techniques that are hard-coded restrict the impact for reuse.

2.2 Existing Approaches to Support Model Replication

From our past experience in MDE, the following two techniques represent approaches to model replication used in common practice: 1) an intermediate phase of replication within a model compiler, 2) a domain-specific model compiler that performs replication for a particular modeling language.

A1. Intermediate stage of model compilation: A model compiler translates the representation of a model into some other artifacts (e.g., source code, configuration files, or simulation scripts). As a model compiler performs its translation, it typically traverses an internal representation of the model through data structures and APIs provided by the host modeling tool. One of our earlier ideas for scaling large models considered performing the replication as an intermediate stage of the model compiler. Prior to the generation phase of the compilation, the intermediate representation can be expanded to address the desired scalability. This idea is represented in Figure 1, which shows the model scaling as an internal task within the model compiler that directly precedes the artifact generation.

This approach is an inadequate solution to replication because it violates all three of the desired characteristics enumerated in Section 2.1. The most egregious violation is that the approach destroys the benefits of modeling. Because the replication is performed as a pre-processing phase in the model compiler, the replicated structures are never rendered back into the modeling tool itself to produce scaled models such that model engineers can further analyze the model scaling results. Thus, analysis and design alternatives are not made available to an end-user who wants to further evaluate the scaled models. Additionally, the pre-processing rules are hard-coded into the model compiler and intermixed with other concerns related to artifact generation. This coupling offers little opportunity for reuse in other modeling languages. In general, this is the least flexible of all approaches that we considered.

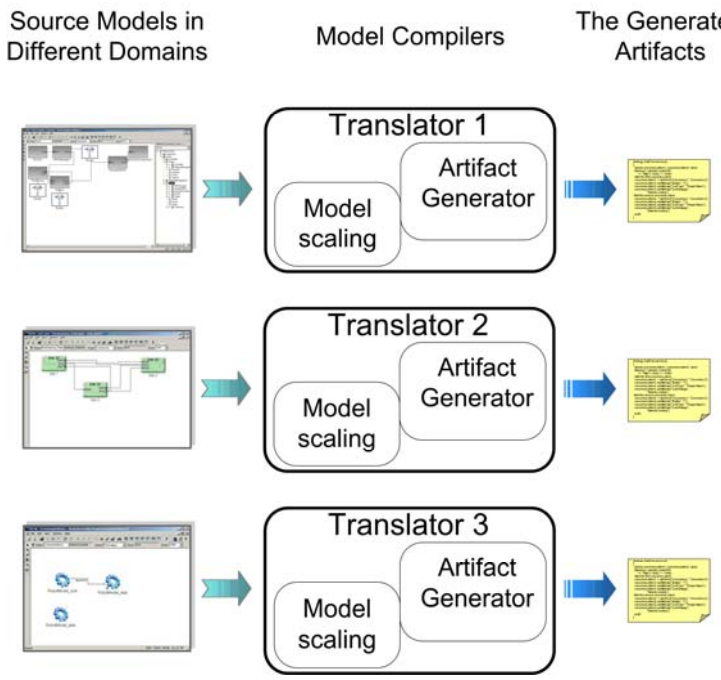


Figure 1: Replication as an Intermediate Stage of Model Compilation (A1)

A2. Domain-specific model compiler to support replication: This approach to model scalability constructs a model compiler that is capable of replicating the models as they appear in the tool such that the result of model scaling is available to the end-user for further consideration and analysis. Such a model compiler has detailed knowledge of the specific modeling language, as well as the particular scalability concern. Unlike approach A1, this technique preserves the benefits of modeling because the end result of the replication provides visualization of the scaling, and the replicated models can be further analyzed and refined. Figure 2 illustrates the domain-specific model replicator approach, which separates the model scaling task from the artifact generator in order to provide end-users an opportunity to analyze the scaled models. However, this approach also has a few drawbacks. Because the replication rules are hard-coded into the domain-specific model replicator, the developed replicator has limited use outside of the intended modeling language. Thus, the generality across modeling languages is lost.

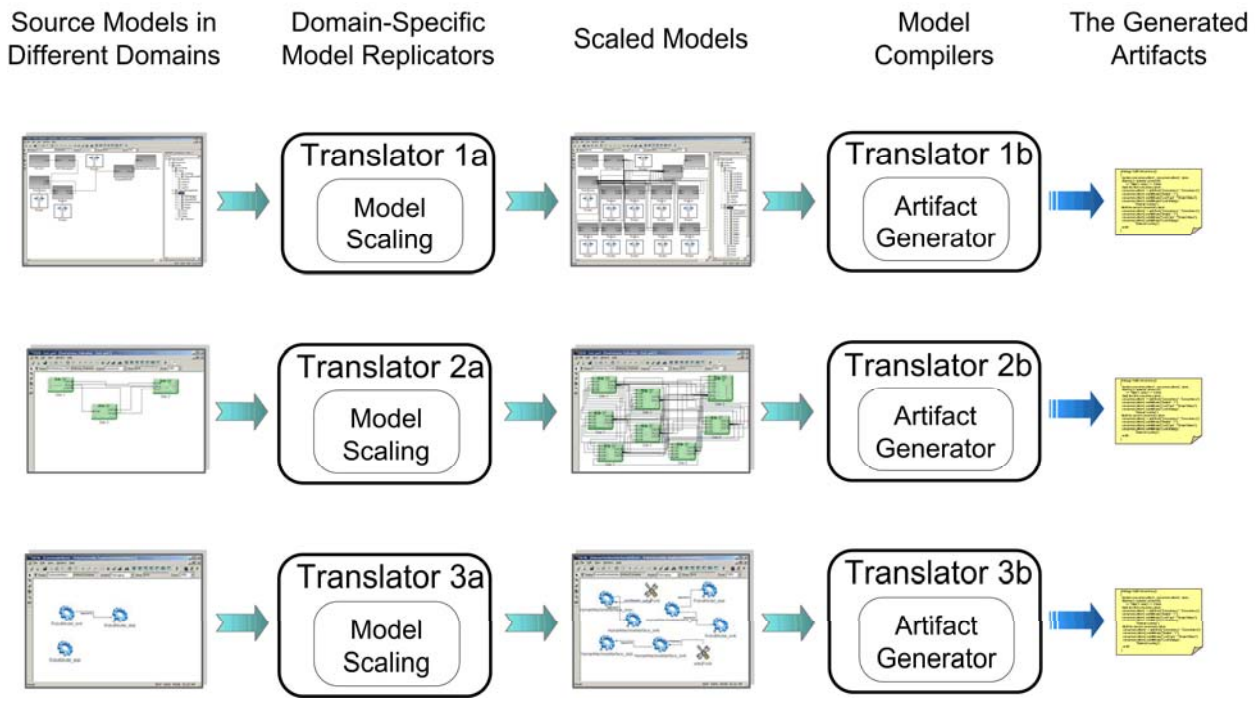


Figure 2: Replication as a Domain-Specific Model Compiler (A2)

These first two approaches have drawbacks when compared against the desired characteristics of Section 2.1. The next section presents a more generalized solution based on a model transformation language.

2.3 Replication with a Model Transformation Language

A special type of model compiler within the GME is a plug-in that can be applied to any metamodel (i.e., it is domain-independent). The C-SAW model transformation engine is an example of a plug-in that can be applied to any modeling language. The type of transformations that can be performed by C-SAW are model-to-model refinement transformations within the same metamodel. C-SAW executes as a model compiler and renders all transformations (as specified in the ECL) back into the host modeling tool. A model transformation written in ECL can be altered very rapidly to analyze the

effect of different degrees of scalability (e.g., the effect on performance when the model is scaled from 256 to 512 nodes).

This third approach to replication (designated as A3) advocates the use of a model transformation engine like C-SAW to perform the replication (please see Figure 3 for an overview of the technique). This technique satisfies all of the desirable characteristics of a replicator: by definition, the C-SAW tool is applicable across many different modeling languages, and the replication strategy is decoupled from other concerns (e.g., artifact generation) and specified in a way that can be easily modified through a higher-level transformation language. These benefits improve the capabilities of hard-coded rules as observed in the approaches described in A1 and A2. With a model transformation engine, a second model compiler is still required for each domain as in A2 (see "Model Compiler" in Figure 3), but the scalability issue is addressed independently of the modeling language.

The key benefits of approach A3 can be seen by comparing it to A2. It can be observed that Figures 2 and 3 are common in the two-stage process of model replication followed by artifact generation with a model compiler. The difference between A2 and A3 can be found in the replication approach. In Figure 2, the replication is performed by three separate model compilers that are hard-coded to a specific domain (Translator 1a, Translator 2a, and Translator 3a), but the replication in Figure 3 is carried out by a single model transformation engine that is capable of performing replication on any modeling language. Approach A3 provides the benefit of a higher level scripting language that can be generalized through parameterization to capture the intent of the replication process. Our most recent efforts have explored this third technique for model replication on several existing modeling languages, as described in the next section.

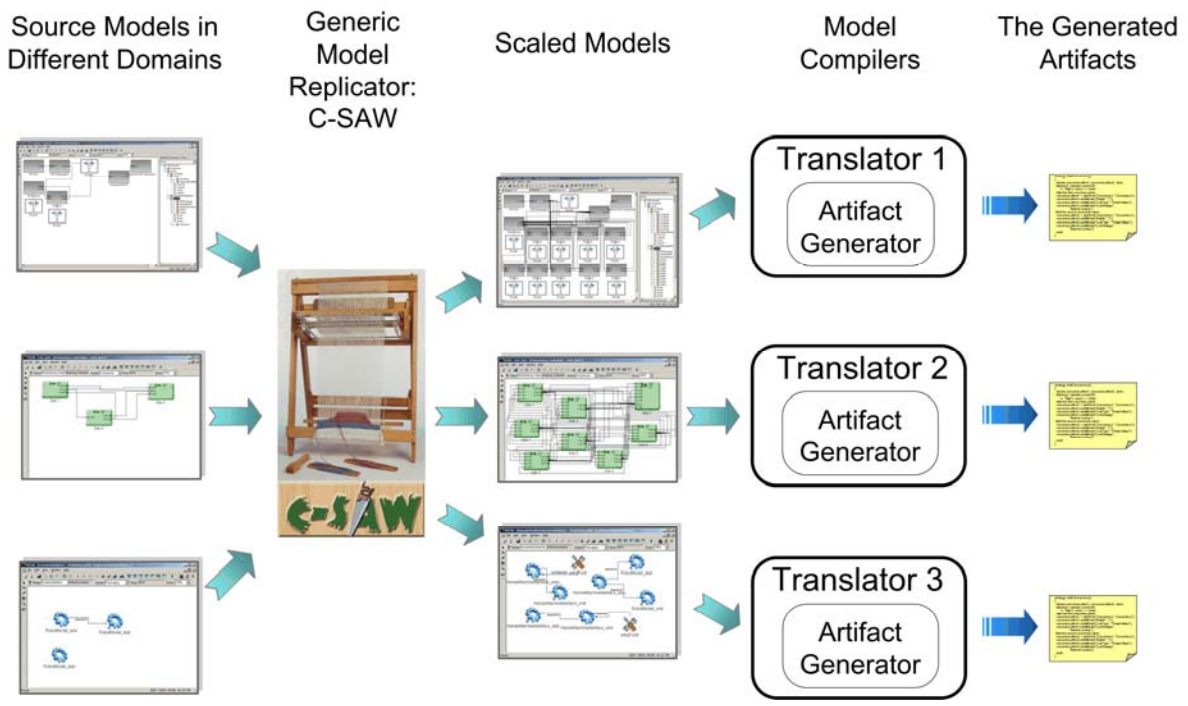


Figure 3: Replication using a Model Transformation Engine (A3)

3. MODEL SCALABILITY WITH MODEL REPLICATORS

This section contains a general description of a model transformation language followed by two case studies that demonstrate the capability of transformations to address model scalability concerns through replication. These case studies capture some of our collective experiences in applying MDE across several domains of inquiry (e.g., fault tolerance in computational physics applications and performance analysis of distributed middleware).

3.1 Using ECL to Define Model Replication Processes

Model scalability is a process that constructs a complex model from a simple base model by replicating model elements or substructures and adding necessary connections. To define and execute such a process in an automated manner requires a model transformation language that provides

support to navigate, query and alter models. Particularly, model replication is a process to manipulate (e.g., create, delete, or change) model elements and connections dynamically. Such a language also needs to support sequential, conditional, repetitive and parameterized model manipulation for defining control flows and enabling data communication between procedures.

ECL supports an imperative transformation style with numerous operations that can alter the structure of the model. It builds on the low-level APIs in C++ provided by GME, but assists end-users in specifying model transformations at a higher level of abstraction. It provides features such as model collection and aggregation (e.g., `rootFolder` in line 31 of Figure 5), as well as operations for model querying and navigation (e.g., `findFolder` in the same line). The ECL offers primitive model transformation operations, such as creation and deletion of model elements and connections. A specific task of model replication can be defined in an ECL specification.

An ECL transformation specification usually consists of two kinds of modular units: an aspect and a strategy, which allow users to procedurally define model changes. An aspect is the entry point of a model transformation and usually used to specify a collection of model elements. A strategy is used to specify elements of computation (e.g., transformation behaviors like adding a model or removing a model) that will be bound to specific model elements defined by an aspect specification. The terms “aspect” and “strategy” are used because ECL was originally designed for Aspect-Oriented Modeling [36], but is considered as a general model transformation language in this paper.

Correspondingly, an ECL specification for model replication includes an aspect to specify what models are subject to replication, and one or more strategies to specify replication processes. Any strategy can be called inside an aspect or another strategy. Inside a strategy, there may be sequential or conditional statements. Repetitive model creation can be implemented in ECL using recursive strategies. Parameterization is another feature of the ECL. For example, property data and control data can be

defined and passed through parameters and static local variables of strategies. Example ECL specifications are shown in the following case studies.

3.2 Case Studies

In this section, the concept of model replicators is demonstrated on two separate example modeling languages that were created in the GME for several domains. In each subsection, the DSML is briefly introduced, including a discussion of the scalability issues and how C-SAW solves the scalability problem through model transformations specified in the ECL. The chosen DSMLs are:

- System Integration Modeling Language (SIML), which has been used to model hardware configurations consisting of up to 5,000 processing nodes for high-energy physics applications at Fermi National Accelerator Lab.
- Stochastic Reward Net Modeling Language (SRNML), which has been used to describe performability concerns of distributed systems built from middleware patterns-based building blocks.

The purpose of introducing these two case studies is to illustrate how our model transformation approach supports scalability among models with various structures: SIML models contain multiple hierarchies and SRNML models usually contain dependent sub-structures.

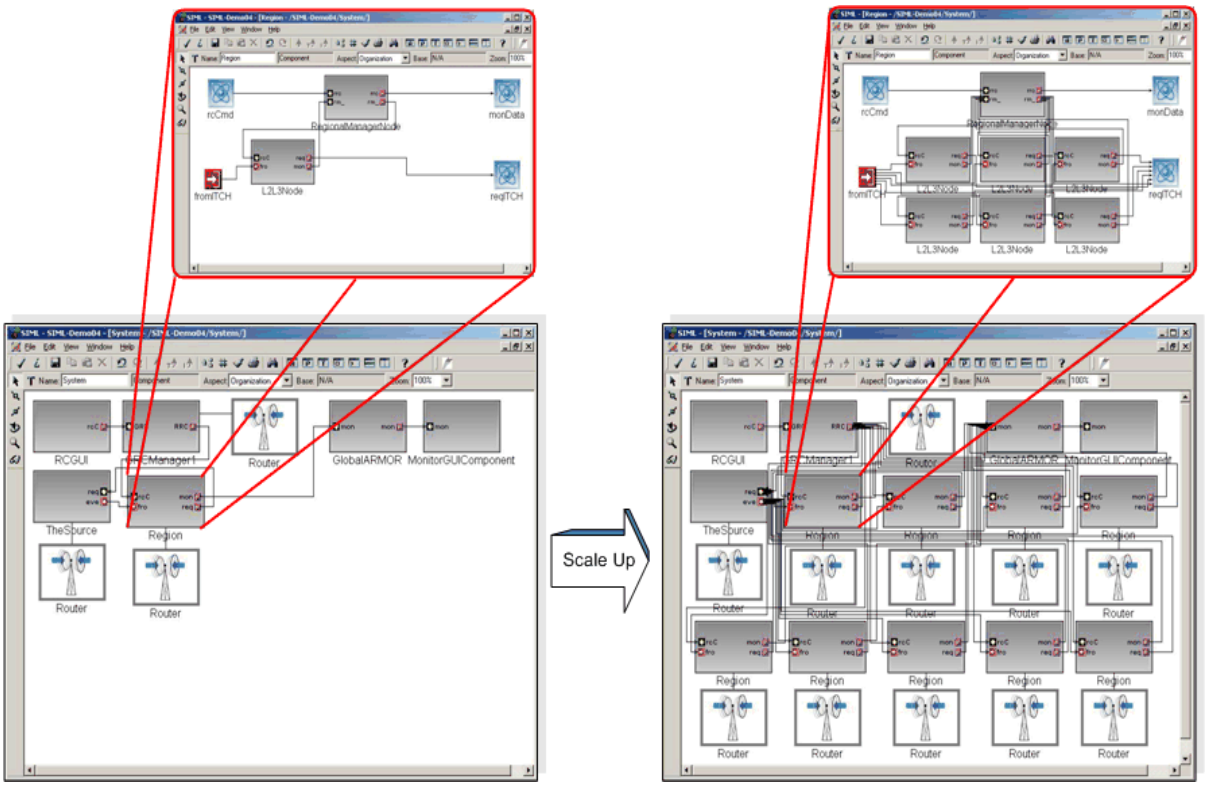


Figure 4: Visual Example of SIML Scalability

3.2.1 Scaling the System Integration Modeling Language

The System Integration Modeling Language (SIML) is a modeling language developed to specify configurations of large-scale fault tolerant data processing systems used to conduct high-energy physics experiments [34]. A system model expressed in SIML captures components and relationships at the systems engineering level. The features of SIML are hierarchical component decomposition and dataflow modeling with point-to-point and publish-subscribe communication between components. There are several rules defined by the SIML metamodel:

- A *system* model may be composed of several independent regions
- Each *region* model may be composed of several independent local process groups

- Each *local process group* model may include several primitive application models
- Each system, region, and local process group must have a representative *manager* that is responsible for mitigating failures in its area

A *local process group* is a set of processes that run the set of critical tasks to perform the system's overall function. In a data processing network, a local process group would include the set of processes that execute the algorithmic and signal processing tasks, as well as the data processing and transport tasks. A *region* is simply a collection of local process groups, and a *system* is defined as a collection of regions and possibly other supporting processes. These containment relationships lead to the multiple hierarchical structures of SIML models. A simple SIML base model is shown on the left side of Figure 4, which captures a system composed of one region and one local process group in that region (shown as an expansion of the parent region), utilizing a total of 15 physical modeling elements (several elements are dedicated to supporting applications not included in any region).

Scalability Issues in SIML: In order to plan, deploy, and refine a high-energy physics data processing system, designers manually build a multitude of different SIML models that are subject to a variety of outside and changing constraints (e.g., the current availability of hardware, software, or human resources). An example of this process would be the manual creation of separate 16-, 32-, 64-, and 128-node versions of a baseline system used for bandwidth and latency testing purposes. This would later be followed by the creation of a set of significantly larger SIML models where the final system model could incorporate as many as 2,500 local processing groups. Each of these models would undergo a variety of analysis routines to determine several key characteristics of the system. These analyses include system throughput, network/resource utilization and worst-case managerial latency (the latency between managers and subordinates is crucial in evaluating the fault tolerance of the system). The results of these analyses may vary greatly as the structure of the system model is scaled in different ways.

The number of system configurations that are created using this process is directly proportional to the time and effort allowed by system designers to create valid system models using a manual approach. In practice, SIML models have been scaled to 32- and 64-node models. However, the initial scaling in these cases was performed manually. The ultimate goal of the manual process was to scale to 2,500 nodes. After 64 nodes, it was determined that scaling to further nodes would be too tedious to perform without proper automation through improved tool support. Even with just a small expansion, the manual application of the same process would require an extraordinary amount of manual effort (much mouse-clicking and typing) to bring about the requisite changes, and increase the potential for introducing error into the model (e.g., forgetting to add a required connection). If the design needs to be scaled forward or backward, a manual approach would require additional effort that would make the exploration and analysis of design alternatives impractical. Therefore, a significant victory for design agility can be claimed if replicators can be shown to scale a base SIML model quickly and correctly into a variety of larger and more elaborate SIML models. This case study shows the benefits that result from scaling SIML models by applying an automated approach that exhibits the desired characteristics for replicators.

To decide what scaling behaviors such a replicator needs to perform, domain-specific knowledge and rules for creating a SIML model as embodied in its metamodel need to be captured. For example, there are one-to-many relationships between system and regional managers, and also one-to-many relationships between regional and local process group managers. These relationships are well-defined. Because the pattern of these relationships is known, it is feasible to write a replicator to perform automatic generation of additional local process groups and/or regions to create larger and more elaborate system models.

In general, scaling up a system configuration using SIML can involve: 1) an increase in the number of regions, 2) an increase in the number of local process groups per region, or 3) both 1 and 2.

Considering the SIML model in Figure 4, the system (which originally has one region with one local process group) is increased to nine regions with six local process groups per region. Such replication involves the following tasks:

- Replication of the local process group models
- Replication of the entire region models and their contents
- Generation of communication connections between the regional managers and newly created local managers
- Generation of additional communication connections between the system manager and new regional manager processes

The scaled model is shown in the right side of Figure 4. This example scales to just 9 regions and 6 nodes per region simply because of the printed space to visualize the figure.

ECL Transformation to Scale SIML: The scalability shown in Figure 4 can be performed by a replicator, which is a model transformation specification in ECL as shown in Figure 5. As a point of support for the effectiveness of replicators as transformations, this ECL specification was written in less than an hour by a user who was very familiar with ECL, but had studied the SIML metamodel for less than a few hours.

The ECL transformation specification is composed of an aspect and several strategies. In Figure 5, the aspect `Start` (Line 1) invokes two strategies, `scaleUpNode` and `scaleUpRegion` in order to replicate the local process group node (i.e., `L2L3Node`) within the region model and the region itself. The strategy `scaleUpNode` (Line 7) discovers the `Region` model, sets up the context for the transformation, and calls the strategy `addNode` (Line 12) that will recursively increase the number of local process group nodes. The new node instance is created on Line 18, which is followed by the construction of the communication connections between ports, regional managers and the newly created

nodes (Line 21 to Line 23). Some other connections are omitted here for the sake of brevity. Two other strategies `scaleUpRegion` (Line 29) and `addRegion` (Line 34) follow a similar mechanism.

The process of modeling systems using SIML presents the benefits of replicators by providing an automated technique that uses transformations to scale models in a concise and flexible manner. Because of the multiple hierarchies of SIML models, replications usually need to be performed on all the elements associated with containment relationships within a model. To perform a scaling task across multiple model hierarchies, ECL supports model navigation through its model querying and selection operations. A model navigation concern can be specified concisely in the ECL. For example, Line 9 is a declarative statement for finding all the region models by navigating from the root folder to the system model, which calls these three querying operations: `rootFolder()`, `findFolder()` and `findModel()`.

Also, flexibility of the replicator can be achieved in several ways. Lines 3 and 4 specify the magnitude of the scaling operation, as well as the names of the specific nodes and regions that are to be replicated. In addition to these parametric changes that can be made easily, the semantics of the replication can be changed because the transformation specification can be modified directly by an end-user. This is not the case in approaches A1 and A2 from Section 2.2 because the replication semantics are hard-coded into the model compiler.

To conclude, replicating a hierarchical model requires that a model transformation language like ECL provide the capability to traverse models, the flexibility to change the scale of replication, and the computational power to change the data attributes within a replicated structure.

```
1  aspect Start()
2  {
3    scaleUpNode("L2L3Node", 5); //add 5 L2L3Nodes in the Region
4    scaleUpRegion("Region", 8); //add 8 Regions in the System
5  }
6
7  strategy scaleUpNode(node_name : string; max : integer)
8  {
9     rootFolder().findFolder("System").findModel("Region").addNode(node_name,max,1);
10 }
11
12 strategy addNode(node_name, max, idx : integer)           //recursively add nodes
13 {
14   declare node, new_node, input_port, node_input_port : object;
15
16   if (idx<=max) then
17     node := rootFolder().findFolder("System").findModel(node_name);
18     new_node := addInstance("Component", node_name, node);
19
20     //add connections to the new node; three similar connections are omitted here
21     input_port := findAtom("fromITCH");
22     node_input_port := new_node.findAtom("fromITCH");
23     addConnection("Interaction", input_port, node_input_port);
24
25     addNode(node_name, max, idx+1);
26   endif;
27 }
28
29 strategy scaleUpRegion(reg_name : string; max : integer)
30 {
31    rootFolder().findFolder("System").findModel("System").addRegion(reg_name,max,1);
32 }
33
34 strategy addRegion(region_name, max, idx : integer)      //recursively add regions
35 {
36   declare region, new_region, out_port, region_in_port, router, new_router : object;
37
38   if (idx<=max) then
39     region := rootFolder().findFolder("System").findModel(region_name);
40     new_region := addInstance("Component", region_name, region);
41
42     //add connections to the new region; four similar connections are omitted here
43     out_port := findModel("TheSource").findAtom("eventData");
44     region_in_port := new_region.findAtom("fromITCH");
45     addConnection("Interaction", out_port, region_in_port);
46
47     //add a new router and connect it to the new region
48     router := findAtom("Router");
49     new_router := copyAtom(router, "Router");
50     addConnection("Router2Component", new_router, new_region);
51
52     addRegion(region_name, max, idx+1);
53   endif;
54 }
```

Figure 5: ECL Model Transformation to Perform the Replication Shown in Figure 4

3.2.2 Scaling the Stochastic Reward Net Modeling Language

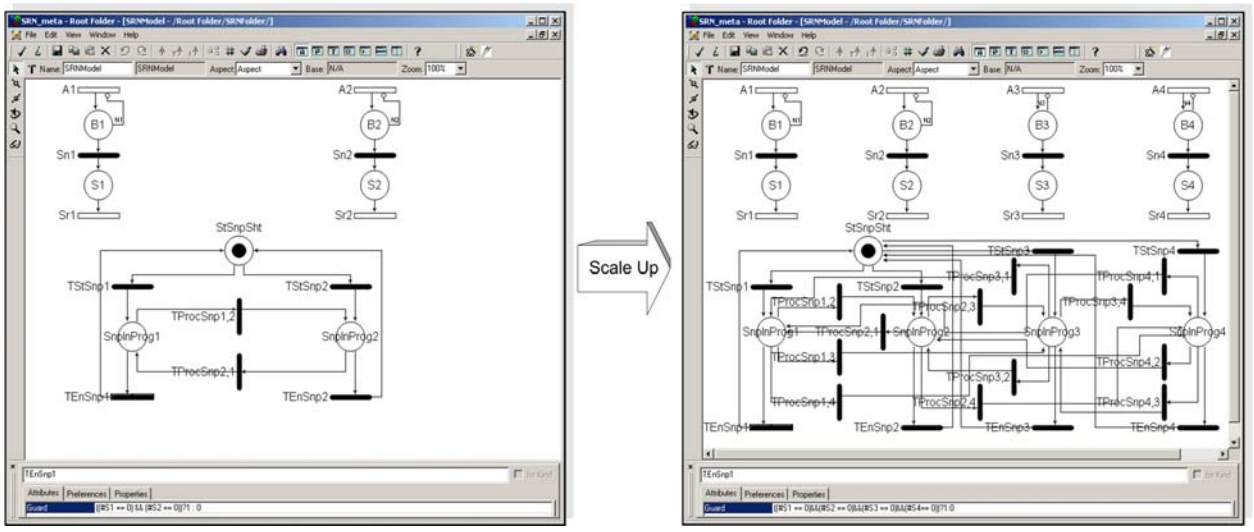
Stochastic Reward Nets (SRNs) [25] represent a powerful modeling technique that is concise in its specification and whose form is closer to a designer's intuition about what a performance model should look like. Because an SRN specification is closer to a designer's intuition of system behavior, it is also easier to transfer the results obtained from solving the models and interpret them in terms of the entities that exist in the system being modeled. SRNs have been used extensively for performance, reliability and performability modeling of different types of systems. SRNs are the result of a chain of evolution starting with Petri nets [28]. More discussion on SRNs can be found in [22, 30].

The Stochastic Reward Net Modeling Language (SRNML) [19] is a DSML we have developed in GME to describe SRN models of large distributed systems. The SRNML is similar to the goals of performance-based modeling extensions for the UML, such as the Schedulability, Performance, and Time profile [29]. The model compilers that we have developed for SRNML can synthesize artifacts required for the SPNP tool [19], which is a model solver based on SRN semantics. SRNML captures all the aspects of the evolution described above.

The SRN models in this paper, which are specified in SRNML, depict the Reactor pattern [31] in middleware for network services, which provides synchronous event demultiplexing and dispatching mechanisms. In the Reactor pattern, an application registers an event handler with the event demultiplexer and delegates to it the responsibility of listening for incoming events. On the occurrence of an event, the demultiplexer dispatches the event by making a callback to its associated application-supplied event handler. As shown in Figure 6a, an SRN model usually consists of two parts: the top half represents the event types handled by a reactor and the bottom half defines the associated execution snapshot. The execution snapshot needs to represent the underlying mechanism for handling the event types included in the top part (e.g., non-deterministic handling of events). Thus, there are implied dependent relations

between the top and bottom parts. Any change made to the top will require corresponding changes to the bottom.

Figure 6a shows the SRN model for the reactor pattern for two event handlers. The top of Figure 6a models the arrival, queuing and service of the two event types. Transitions *A*1 and *A*2 represent the arrivals of the events of types one and two, respectively. Places *B*1 and *B*2 represent the queue for the two types of events. Transitions *Sn*1 and *Sn*2 are immediate transitions that are enabled when a snapshot is taken. Places *S*1 and *S*2 represent the enabled handles of the two types of events, whereas transitions *Sr*1 and *Sr*2 represent the execution of the enabled event handlers of the two types of events. An inhibitor arc from place *B*1 to transition *A*1 with multiplicity *N*1 prevents the firing of transition *A*1 when there are *N*1 tokens in place *B*1. The presence of *N*1 tokens in place *B*1 indicates that the buffer space to hold the incoming input events of the first type is full, and no additional incoming events can be accepted. The inhibitor arc from place *B*2 to transition *A*2 achieves the same purpose for type two events.



a) base model with 2 event handlers **b) scaled model with 4 event handlers**

Figure 6: Replication of Reactor Event Types (from 2 to 4 event types)

Table 1: Enabling Guard Equations for Figure 6

Transition	Guard Function
Sn_1	((#*StSnpShot* == 1) && (#B_1 >= 1) && (#S_1 == 0)) ? 1 : 0
⋮	⋮
Sn_m	((#*StSnpShot* == 1) && (#B_m >= 1) && (#S_m == 0)) ? 1 : 0
$TStSnp_1$	(#S_1 == 1) ? 1 : 0
⋮	⋮
$TStSnp_m$	(#S_m == 1) ? 1 : 0
$TEnSnp_1$	((#S_1 == 0) && (#S_2 == 0) && **…** (#S_m == 0)) ? 1 : 0
⋮	⋮
$TEnSnp_m$	((#S_1 == 0) && (#S_2 == 0) && **…** (#S_m == 0)) ? 1 : 0
$TProcSnp_{1,2}$	((#S_1 == 0) && (#S_2 == 1)) ? 1 : 0
⋮	⋮
$TProcSnp_{1,m}$	((#S_1 == 0) && (#S_m == 1)) ? 1 : 0
⋮	⋮
$TProcSnp_{m,m-1}$	((#S_m == 0) && (#S_{m-1} == 1)) ? 1 : 0
⋮	⋮
$TProcSnp_{m,1}$	((#S_m == 0) && (#S_1 == 1)) ? 1 : 0
Sr_1	(#$SnpInProg_1$ == 1) ? 1 : 0
⋮	⋮
Sr_m	(#$SnpInProg_m$ == 1) ? 1 : 0

The bottom of Figure 6a models the process of taking successive snapshots and non-deterministic service of event handles in each snapshot. Transition *Sn*1 is enabled when there are one or more tokens in place *B*1, a token in place *StSnpSht*, and no token in place *S*1. Similarly, transition *Sn*2 is enabled when there are one or more tokens in place *B*2, a token in place *StSnpSht* and no token in place *S*2. Transitions *TStSnp*1 and *TStSnp*2 are enabled when there is a token in place *S*1, place *S*2, or both. Transitions *TEnSnp*1 and *TEnSnp*2 are enabled when there are no tokens in both places *S*1 and *S*2. Transition *TProcSnp*1,2 is enabled when there is no token in place *S*1 and a token in place *S*2. Similarly, transition *TProcSnp*2,1 is enabled when there is no token in place *S*2 and a token in place *S*1. Transition *Sr*1 is

enabled when there is a token in place *SnpInProg*1, and transition *Sr*2 is enabled when there is a token in place *SnpInProg*2. All the transitions have their own guard functions, as shown in Table 1.

Scalability Issues in SRNML: The scalability challenges of SRN models arise from the addition of new event types and connections between their corresponding event handlers. For example, the top of the SRN model must scale to represent the event handling for every event type that is available. A problem emerges when there could be non-deterministic handling of events, which leads to the complicated connections between the elements within the execution snapshot of an SRN model. Due to the implied dependencies between the top and bottom parts, the bottom part of the model (i.e., the snapshot) should incorporate appropriate non-deterministic handling depicted in the scaled number of event types. The inherent structural complexity and the complicated dependent relations within an SRN model make it difficult and impractical to scale up SRN models manually, which requires a computer-aided method such as a replicator to perform the replication automatically.

The replication behaviors for scaling up an SRN model can be formalized as computation logic and specified in a model transformation language such as ECL. As illustrated in Figure 6, Figure 6a describes a base SRN model for two event types, and Figure 6b represents the result of scaling this base model from two event types to four event types. Such scalability in SRN models can be performed with two model transformation steps. The first step scales the reactor event types (i.e., the upper part of the SRN model) from two to four, which involves creating the *B* and *S* places, the *A*, *Sn* and *Sr* transitions and associated connection arcs and renaming them, as well as setting appropriate guard functions for each new event type. The second step scales the snapshot (i.e., the bottom part of the SRN model) according to the newly added event types. Inside a snapshot, the model elements can be divided into three categories. The first category is a group of elements that are independent of each event type; the second category is a group of model elements that are associated with every two new event types; and the third category is a

group of elements that are associated to one old event type and one new event type. Briefly, these three groups of elements can be built by three sub-tasks:

1. Create the `TStSnp` and `TEnSnp` transitions and the `SnpInProg` place, as well as required connection arcs among them for each newly added event type; assign the correct guard function for each created transition; this task builds the first group.
2. For each pair of new event types, create two `TProcSnp` transitions and connect their `SnpInProg` places to these `TProcSnp` transitions; assign the correct guard function for each created transition; this task builds the second group.
3. For each pair of <old event type, new event type>, create two `TProcSnp` transitions and connect their `SnpInProg` places to these `TProcSnp` transitions; assign the correct guard function for each created transition; this task builds the third group.

ECL Transformation to Scale SRNML: In this example, only the model transformation for scaling the snapshot is illustrated. The ECL specification shown in Figure 7 performs sub-task one. It is composed of several strategies. The `computeTEnSnpGuard` (Line 1) strategy is used to re-compute the guard functions of the `TEnSnp` transitions when new event types are added. The ECL code on Lines 3 and 4 recursively concatenate the string that represents the guard function. After this string is created, it is passed to the `addEventsWithGuard` strategy (not shown here), which adds the new guard function and event to the snapshot. The `addEvents` strategy (Line 12) recursively calls the `addNewEvent` strategy to create necessary transitions, places and connections in the snapshot for the new event types with identity numbers from `min_new` to `max_new`. The `addNewEvent` strategy (Line 20) creates snapshot elements for a single new event type with identity number `event_num`. The `findAtom` operation on Line 25 is used to discover the `StSnpSht` place in the snapshot. The `TStSnp` transition is

created on Line 26 and its guard function is created on Lines 27 and 28. Next, the `SnpInProg` place and the `TEnSnp` transition are created on Lines 30 and 31, respectively. The guard function of the `TEnSnp` transition is set on Line 32. Finally, four connection arcs are created among the `StSnpSht` place, the `TStSnp` transition, the `SnpInProg` place and the `TEnSnp` transition (Lines 34 to 37).

```
1  strategy computeTEnSnpGuard(min_old, min_new, max_new : integer; TEnSnpGuardStr : string)
2  {
3       if (min_old < max_new) then
4          computeTEnSnpGuard(min_old + 1, min_new, max_new, TEnSnpGuardStr +
                              "(#S" + intToString(min_old) +  " == 0)&&");
5       else
6          addEventswithGuard(min_new, max_new, TEnSnpGuardStr + "(#S" + intToString(min_old) +
                              "== 0))?1:0");
7       endif;
8  }
9
10 ...  // several strategies not show here (e.g., addEventswithGuard)
11
12 strategy addEvents(min_new, max_new : integer; TEnSnpGuardStr : string)
13 {
14      if (min_new <= max_new) then
15              addNewEvent(min_new, TEnSnpGuardStr);
16              addEvents(min_new+1, max_new, TEnSnpGuardStr);
17      endif;
18 }
19
20 strategy addNewEvent(event_num : integer; TEnSnpGuardStr : string)
21 {
22      declare start, stTran, inProg, endTran : atom;
23      declare TStSnp_guard : string;
24
25      start := findAtom("StSnpSht");
26      stTran := addAtom("ImmTransition", "TStSnp" + intToString(event_num));
27      TStSnp_guard := "(#S" + intToString(event_num) + " == 1)?1 : 0";
28      stTran.setAttribute("Guard", TStSnp_guard);
29
30      inProg := addAtom("Place", "SnpInProg" + intToString(event_num));
31      endTran := addAtom("ImmTransition", "TEnSnp" + intToString(event_num));
32      endTran.setAttribute("Guard", TEnSnpGuardStr);
33
34      addConnection("InpImmedArc", start, stTran);
35      addConnection("OutImmedArc", stTran, inProg);
36      addConnection("InpImmedArc", inProg, endTran);
37      addConnection("OutImmedArc", endTran, start);
38 }
39 ...
```

Figure 7: ECL Model Transformation to Perform First Sub-task of Scaling Snapshot

Sub-task one actually creates independent snapshot elements for each new event type. In collaboration, sub-tasks two and three build the necessary relationships between each pair of new event types, and each pair consisting of a new event type and an old event type. Figure 8 shows the ECL specification to perform sub-task two (i.e., to build the relationship between every two new event types). The `connectTwoEvents` strategy (Line 17) creates the `TProcSnp` transition and its associated connections between two events. Then, the `connectOneNewEventToOtherNewEvents` strategy (Line 9) recursively calls the `connectTwoEvent` strategy to build relationships between two new events. Finally, the `connectNewEvents` strategy (Line 1) builds the relationships between each pair of new event types by recursively calling the `connectOneNewEventToOtherNewEvents` strategy. Inside the `connectTwoEvents` strategy, the `SnpInProg` places of the two event types are discovered on Lines 28 and 29, respectively. Then, two new `TProcSnp` transitions are created and their guard functions are set (Lines 30 through 33), followed by the construction of the connections between the `SnpInProg` places and the `TProcSnp` transitions (Lines 35 through 38).

To conclude, the introduction of new event types into an SRN model requires changes in several locations of the model. For example, new event types need to be inserted; some properties of model elements such as the guard functions need to re-computed; and the execution snapshot needs to be expanded accordingly. The difficulties of scaling up an SRN model manually is due to the complicated dependencies among its model elements and parts, which can be easily addressed by a replicator using C-SAW and its model transformation language ECL. With the expressive power of ECL, it is possible to specify reusable complicated transformation logic in a templatized fashion. The replication task is also simplified by distinguishing independent and dependent elements, and building up a larger SRN model in a stepwise manner. The result of the model replication preserves the benefits of modeling (benefit C1

from Section 2) because the result of the replication can be persistently exported to XML and sent to a Petri Net analysis tool.

```
1  strategy connectNewEvents(min_new, max_new: interger)
2  {
3       if(min_new < max_new) then
4              connectOneNewEventToOtherNewEvents(min_new, max_new);
5              connectNewEvents(min_new+1, max_new);
6       endif;
7  }
8
9  strategy connectOneNewEventToOtherNewEvents(event_num, max_new: integer)
10 {
11      if(event_num < max_new) then
12             connectTwoEvents(event_num, max_new);
13             connectNewEvents(event_num, max_new-1);
14      endif;
15 }
16
17 strategy connectTwoEvents(first_num, second_num : integer)
18 {
19      declare firstinProg, secondinProg : atom;
20      declare secondTProc1, secondTProc2 : atom;
21      declare first_numStr, second_numStr, TProcSnp_guard1, TProcSnp_guard2 : string;
22
23      first_numStr := intToString(first_num);
24      second_numStr := intToString(second_num);
25      TProcSnp_guard1 := "((#S" + first_numStr + " == 0) && (#S" + second_numStr +
                           "  == 1))?1 : 0";
26      TProcSnp_guard2 := "((#S" + second_numStr + " == 0) && (#S" + first_numStr +
                           " == 1))?1 : 0";
27
28      firstinProg := findAtom("SnpInProg" + first_numStr);
29      secondinProg := findAtom("SnpInProg" + second_numStr);
30      secondTProc1 := addAtom("ImmTransition", "TProcSnp" + first_numStr +
                               "," + second_numStr);
31      secondTProc1.setAttribute("Guard", TProcSnp_guard1);
32      secondTProc2 := addAtom("ImmTransition", "TProcSnp" + second_numStr +
                               "," + first_numStr);
33      secondTProc2.setAttribute("Guard", TProcSnp_guard2);
34
35      addConnection("InpImmedArc", firstinProg, secondTProc1);
36      addConnection("OutImmedArc", secondTProc1, secondinProg);
37      addConnection("InpImmedArc", secondinProg, secondTProc2);
38      addConnection("OutImmedArc", secondTProc2, firstinProg);
39 }
40 ...
```

Figure 8: ECL Model Transformation to Perform Second Sub-task of Scaling Snapshot

In addition to the examples discussed above, we have also developed a replication strategy for the Event Quality Aspect Language (EQAL), which has been used to configure a large collection of federated event channels for mission computing avionics applications. Replication within EQAL was reported in [14, 15].

4. RELATED WORK

The general area of related work concerns modeling research and practice that provide abilities to specify replication concerns for instance-based models and to create model elements and links dynamically to address model scalability. From our literature review, only a few researchers have proposed methodologies and techniques for such a purpose. We have not found any actual practice and experience to be reported beyond the case studies used in Section 3 of this paper.

4.1 Related Work on Model Scalability

Related work on model scalability contributes to the ability to specify and generate instance-based models with repetitive structures. The approach proposed by Milicev [24] uses extended UML Object Diagrams to specify the instances and links of a target model that is created during automatic translation; this target model is called the domain mapping specification. An automatically generated model transformer is used to produce intermediate models, which are refined to final output artifacts (e.g, C++ codes). Similar to our approach, Milicev adopts a model transformation approach whereby users write and execute transformation specifications to produce instance-based models. However, different from our approach, Milicev's work is domain-dependent because the domain mapping specifications and model transformers are domain-specific. Moreover, the target of his work is the reusability of code generators built in existing modeling tools, which introduces an intermediate model representation to bridge multiple abstraction levels (e.g., from model to code). The target of our approach is to use existing model transformation tools, which support model-to-model transformations at the same abstraction level, to perform model replication in a domain-independent manner without any effort toward extending existing model representations across different abstraction levels.

Several researchers have proposed standard notations to represent repetitive structures of modeling real-time and embedded systems, which is helpful in discovering possible model replication patterns. The MARTE RFP (Modeling and Analysis of Real-Time and Embedded systems) was issued by the OMG in February 2005. This request for proposals solicits submissions for a UML profile that adds capabilities for modeling Real-Time and Embedded Systems (RTES), and for analyzing schedulability and performance properties of UML specifications. One of the particular requests of this RFP concerns the definition of common high-level modeling constructs for factoring repetitive structures for software and hardware. Motivated by this RFP, Cuccuru et al. [8] proposed multi-dimensional multiplicities and mechanisms for the description of regular connection patterns between model elements. However, these proposed patterns are in an initial stage and have not been used by any existing modeling tools. Their proposal mainly works with models containing repetitive elements that are identical, but may not specify all the model replication situations that we identified in this paper (e.g., to represent models with a collection of model elements of the same type, which are slightly different in some properties, or have similar but not identical relationships with their neighbors). As such research matures, we believe significant results will be achieved toward representation of repetitive or even similar model structures. Such maturity will contribute to standardizing and advancing model replication capabilities.

4.2 Related Work on Model Transformation

Our approach advocates using existing model transformation techniques and tools to address model scalability, especially where modeling languages themselves lack support for dynamic creation of model instances and links. This investigation on the application of model transformations to address scalability concerns extends the application area of model transformations. The practice and experiences illustrated in this paper help to motivate the need for model scalability. However, there is no specific reason that

GME, ECL and C-SAW need to be used for the general notion of model replication promoted in this paper; we used this set of tools simply because they were most familiar to us and we had access to several DSMLs based on the combination of these tools.

We are not aware of any other research that has investigated the application of model transformations to address scalability concerns like those illustrated in this paper. However, a large number of approaches to model transformation have been proposed by both academic and industrial researchers and there are many model transformation tools available (example surveys can be found in [9, 23, 33]). The combination of such toolsuites is likely to offer similar capabilities toward specifying replicators for model scalability.

There are several approaches to model transformation, such as graphical languages typified by graph grammars (e.g., GReAT [1] and Fujaba [11]), or a hybrid language (e.g., the ATLAS Transformation Language [6] and Yet Another Transformation Language [27]). Graphical transformation languages provide a visual notation to specify graphical patterns of the source and target models (e.g., a subgraph of a graph). However, it can be tedious to use purely graphical notations to describe complicated computation algorithms. As a result, it may require generation to a separate language to apply and execute the transformations. A hybrid language transformation combines declarative and imperative constructs inside the transformation language. Declarative constructs are used to specify source and target patterns as transformation rules (e.g., filtering model elements), and imperative constructs are used to implement sequences of instructions (e.g., assignment, looping and conditional constructs). However, embedding predefined patterns can render complicated syntax and semantics for a hybrid language. Many existing model transformation languages (including those discussed above) allow transformation to be specified between two different domains (e.g., a transformation that converts a UML model into an entity-relationship model). The ECL can be distinguished from these approaches as a relatively simple and easy-

to-learn language that focuses on specifying and executing model-to-model transformation within the same domain. However, it has full expressive power for such a model replication purpose because a model replication is a specific model-to-model transformation within the same domain.

5. CONCLUSION

This paper has demonstrated the effectiveness of using a general model transformation engine to specify replicators that assist in scaling models; as such, it has promoted the use of model transformation in a new role to support model scalability. Among the approaches to model scalability, a model transformation engine offers several benefits, such as domain-independence and improvements to productivity (when compared to either the corresponding manual effort, or the effort required to write model compilers that are specific to a domain and scalability issue). The model replicator approach presented in this paper assists in increasing the number of model elements while adjusting the core dependencies within a model. The case studies presented in this paper highlight the ease of specification and the general flexibility provided across domains represented by different modeling languages (e.g., SIML and SRNML). The C-SAW web page [2] provides example videos of the model scalability case studies presented in this paper.

There are several existing limitations to the approach presented in this paper. Three limitations that affect the replicator process, which suggest areas of future investigation, are:

- *Visualization of result*: Although model transformations provide a mechanism to physically scale a base model, this may lead to undesirable consequences with respect to visualization of the result. After replication, a large amount of modeling elements may be placed in a random location on the modeling canvas (e.g., in C-SAW, all of the new elements that are added to a model are placed in the upper-left corner). Tools like the GME provide auto-layout features, but the placement from the auto-layout algorithm does not consider semantic issues of the domain (e.g.,

it may be desirable to arrange the modeling elements in a certain style and pattern based upon the underlying type of each element). It is possible to specify layout information in the ECL (i.e., the <x, y> coordinates of a modeling element are simply a property that can be set with the ECL `setAttribute` operator), but this forces the intent of the replication to be coupled with layout concerns. Furthermore, even if the layout issue is customized as part of the transformation, the sheer amount of information in the result may be overwhelming (e.g., consider the difficulty of comprehending the result of scaling to 20 event handlers in the SRNML example). The explosion of the visual presentation may suggest that some additional hierarchy is needed within the DSML itself to manage the expansion of a base model within a flat container.

- *Limitations of host modeling tool*: With respect to scalability, most modeling tools will eventually reach a limit on the size of a model that can be loaded into memory. We faced this limitation with the GME as we tried to scale some of the models from the case studies presented in this paper. For example, in the SRNML model of Figure 6, the GME would crash whenever 439 event types were reached[2] (approximately 4,400 modeling elements added at the same time). Interestingly, it was possible to add 500 event types if the replication was performed as two steps of 250. This suggests that C-SAW was adding modeling elements at a speed that eventually GME could not process. To avoid such limit, more research efforts are needed for compact representation of models with repetitive and similar structures.
- *Correctness of replication*: Transformation specifications, such as those used to specify the replicators in this paper, are written by humans and prone to error. To improve the robustness and reliability of model transformation, there is a need for testing and debugging support to assist in finding and correcting the errors in transformation specifications. Ongoing and future work on

[2] This result was confirmed on several machines.

ECL focuses on the construction of testing and debugging utilities within C-SAW to ensure the correctness of the ECL transformation specifications [21].

- *Patterns and standards of replication*: The specification of repetitive model structures may have common aspects in a specific domain or across different domains. A future area of investigation will consider transformation patterns to solve general purpose or domain-specific model replication. Undoubtedly, efforts to catalog different patterns of model replication will also improve reusability and maintainability of transformations. With respect to model transformation standardization efforts, C-SAW was under development two years prior to the initiation of OMG's Query View Transformation (QVT) request for proposal. It seems reasonable to expect that the final QVT standard would be able to describe transformations similar to those presented in this paper. For the purpose of exploring our research efforts, we have decided to continue our progress on developing C-SAW and later re-evaluate the merits of merging toward a standard.

ACKNOWLEDGMENTS

This project was supported by the DARPA Program Composition for Embedded Systems (PCES) program and is currently supported by the National Science Foundation under CSR-SMA-0509342 and CSR-SMA-0509296.

REFERENCES

1. http://escher.isis.vanderbilit.edu/tools/get_tool?GME
2. http://www.cis.uab.edu/gray/Research/C-SAW/
3. http://www.aspect-modeling.org/
4. Agrawal A, Karsai G, Lédeczi Á. An End-to-End Domain-Driven Software Development Framework. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) – Domain-driven Track*, Anaheim, CA, October 2003; 8-15.
5. Batory D, Sarvela JN, Rauschmeyer A. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, June 2004; 355-371.
6. Bézivin J, Jouault F, Valduriez P. On the Need for MegaModels. *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, Vancouver, BC, October 2004.
7. Ciardo G, Muppala J, Trivedi K. SPNP: Stochastic Petri Net Package. *Workshop on Petri Nets and Performance Models*, Kyoto, Japan, December 1989; 142-150.
8. Cuccuru A, Dekeyser JL, Marquet P, Boulet P. Towards UML2 Extensions for Compact Modeling of Regular Complex Topologies. *Model-Driven Engineering Languages and Systems (MoDELS)*, Springer-Verlag LNCS 3713, Montego Bay, Jamaica, October 2005; 445-459.
9. Czarnecki K, Helsen S. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 2006; **45**(3):621-646.
10. Edwards G, Deng G, Schmidt D, Gokhale AS, Natarajan B. Model-Driven Configuration and Deployment of Component Middleware Publish/Subscribe Services. *Generative Programming and Component Engineering (GPCE)*, Vancouver, BC, October 2004; 337-360.
11. *The FUJABA Toolsuite*, http://www.fujaba.com.
12. Gokhale A, Schmidt D, Natarajan B, Gray J, Wang N. Model-Driven Middleware. *Middleware for Communications*, (Qusay Mahmoud, editor), John Wiley and Sons, 2004.
13. Gray J, Bapty T, Neema S, Tuck J. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, October 2001; 87-93.

14. Gray J, Lin Y, Zhang J, Nordstrom S, Gokhale A, Neema S, Gokhale S. Replicators: Transformations to Address Model Scalability. *Model-Driven Engineering Languages and Systems (MoDELS)*, Springer-Verlag LNCS 3713, Montego Bay, Jamaica, October 2005; 295-308.
15. Gray J, Lin Y, Zhang J. Automating Change Evolution in Model-Driven Engineering. *IEEE Computer* (Special Issue on Model-Driven Engineering), February 2006; 41-48.
16. Gray J, Tolvanen JP, Kelly S, Gokhale A, Neema S, Sprinkle J. Domain-Specific Modeling. *CRC Handbook on Dynamic System Modeling*, (Paul Fishwick, editor), CRC Press, 2006.
17. Hatcliff J, Deng W, Dwyer M, Jung G, Venkatesh Prasad Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. *International Conference on Software Engineering*, Portland, OR, May 2003; 160-173.
18. Johann S, Egyed A. Instant and Incremental Transformation of Models. *Automated Software Engineering*, Linz, Austria, September 2004; 362-365.
19. Kogekar A, Kaul D, Gokhale A, Vandal P, Praphamontripong U, Gokhale S, Zhang J, Lin Y, Gray J. Model-driven Generative Techniques for Scalable Performabality Analysis of Distributed Systems. *IPDPS Workshop on Next Generation Systems*, Rhodes Island, Greece, April 2006.
20. Lédeczi Á, Bakay A, Maroti M, Volgyesi P, Nordstrom G, Sprinkle J, Karsai G. Composing Domain-Specific Design Environments. *IEEE Computer*, November 2001; 44-51.
21. Lin Y, Zhang J, Gray J. A Framework for Testing Model Transformations. in *Model-driven Software Development*, (Sami Beydeda, Matthias Book, and Volker Gruhn, eds.), Springer, 2005; **10**:219-236.
22. Marsan MA, Balbo G, Conte G, Donatelli S, Franceschinis G. *Modelling with Generalized Stochastic Petri Nets*, Wiley Series in Parallel Computing, John Wiley and Sons, 1995.
23. Mens T, Gorp PV. A Taxonomy of Model Transformation. *International Workshop on Graph and Model Transformation*, Tallinn, Estonia, September 2005, http://tfs.cs.tu-berlin.de/gramot/FinalVersions/PDF/MensVanGorp.pdf.
24. Milicev D. Automatic model transformations using extended UML object diagrams in modeling environments. *IEEE Transactions on Software Engineering*, April 2002; **28**(4):413-431.
25. Muppala J, Ciardo G, Trivedi K. Stochastic Reward Nets for Reliability Prediction. *Communications in Reliability, Maintainability and Serviceability*, July 1994; 9-20.

26. Neema S, Bapty T, Gray J, Gokhale A. Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems. *Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 2487, Pittsburgh, PA, October 2002; 236-251.
27. Patrascoiu O. Mapping EDOC to Web Services using YATL. *Enterprise Distributed Object Computing (EDOC)*, Monterey, CA, September 2004; 286-297.
28. Peterson J. Petri Nets. *ACM Computing Surveys*, September 1977; 223-252.
29. Petriu D, Zhang J, Gu G, Shen H. Performance Analysis with the SPT Profile. in *Model-Driven Engineering for Distributed and Embedded Systems*, (S. Gerard, J.P. Babeau, J. Champeau, eds.), Hermes Science Publishing Ltd., London, England, 2005; 205-224.
30. Rácz S, Telek M. Performability Analysis of Markov Reward Models with Rate and Impulse Reward. *International Conference on Numerical Solution of Markov Chains*, Zaragoza, Spain, September 1999; 169-180.
31. Schmidt D, Stal M, Rohnert H, Buschman F, *Pattern-Oriented Software Architecture – Volume 2: Patterns for Concurrent and Networked Objects*, John Wiley and Sons, 2000.
32. Schmidt D. Model-Driven Engineering. *IEEE Computer*, February 2006; 25-32.
33. Sendall S, Kozaczynski W. Model Transformation – the Heart and Soul of Model-Driven Software Development. *IEEE Software, Special Issue on Model-Driven Software Development*, September/October 2003; **20**(5):42-45.
34. Shetty S, Nordstrom S, Ahuja S, Yao D, Bapty T, Neema S. Systems Integration of Large-Scale Autonomic Systems using Multiple Domain Specific Modeling Languages. *Engineering of Autonomic Systems*, Greenbelt, MD, April 2005; 481-489.
35. Sztipanovits J, Karsai G. Model-Integrated Computing. *IEEE Computer*, April 1997; 10-12.
36. *Workshop Series on Aspect-Oriented Modeling*, http://www.aspect-modeling.org/