Replicators: Transformations to Address Model Scalability

Jeff Gray[1], Yuehua Lin[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
{gray, liny, 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

Abstract. In Model Integrated Computing, it is desirable to evaluate different design alternatives as they relate to issues of scalability. A typical approach to address scalability is to create a base model that captures the key interactions of various components (i.e., the essential properties and connections among modeling entities). A collection of base models can be adorned with necessary information to characterize their replication. In current practice, replication is 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 a replicator, which is a model transformation 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 replicators through case studies taken from models supporting different domains.

1. Introduction

A powerful justification for the use of models concerns the flexibility and analysis that can be performed to explore 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 [10]. A general metric for determining the effectiveness of a modeling toolsuite comprises the degree of effort required to make a change to a set of models. In previous work, we have shown how crosscutting concerns that are distributed across a model hierarchy can negatively affect the ability to explore design alternatives [9]. A

L. Briand and C. Williams (Eds.): MoDELS 2005, LNCS 3713, pp. 295-308, 2005.

form of alternative exploration involves experimenting with model structures by scaling up different portions of models and analyzing the result on scalability. This paper makes a contribution to model scalability and describes an approach that can be used to enable automated replication[1] to assist in rapidly scaling a model.

Scalability of modeling tools is of utmost concern to designers of large-scale DRE systems. From our personal experience, models can have multiple thousands of coarse grained components (others have reported similar experience, please see [11]). Modeling these components using traditional model creation techniques and tools can approach the limits of the effective capability of humans. The process of modeling a large DRE system with a domain-specific modeling language (DSML), or a tool like MatLab, is different than traditional UML modeling. In DRE systems modeling, the models consist of instances of all objects in the system, which can number into several thousand instances from a set of types defined in a meta-model (e.g., thousands of individual instantiations of a sensor type in a large sensor network model). The traditional class-based modeling of UML, and supporting tools, are typically not concerned with the same type of instance level focus.

The issue of scalability affects the performance of the modeling process, as well as the correctness of the model representation. 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 clicking and typing within the associated modeling tool. 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).

Often, large-scale system models leverage architectures that are already well suited toward scalability. Likewise, the modeling languages that specify such systems may embody similar patterns of scalability, and may lend themselves favorably toward a generative replication process. The contribution of this paper is automatic generation of large-scale system models from smaller, baseline specification models by applying basic transformation rules that govern the scaling [2] and replication behavior.

The rest of the paper is organized as follows: Section 2 provides an overview of the tools used in the paper, followed by an outline of the technical challenges of model replication in Section 3. Two case studies of model scalability using replicators are provided in Section 4. The conclusion offers summary remarks and a brief description of future work.

[1] The term “replicator” 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 duplication and proper connection of modeling elements to address scalability concerns.

2. Background: Supporting Technologies and Related Work

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 the work are overviewed in this section. The purpose of the paper is not to describe these tools in detail, but an introduction may be needed to understand the subsequent sections of the paper.

2.1 Model-Integrated Computing

A specific form of model-driven development, called Model-Integrated Computing (MIC) [17], has been refined at Vanderbilt University over the past decade to assist 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[2]) [12] is a meta-modeling tool based on MIC that can be configured and adapted from meta-level specifications (called the modeling paradigm) that describe the domain. An effort to make the GME MOF-compliant is detailed in [6]. Each meta-model describes a domain-specific modeling language (DSML). 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 in order to synthesize a model into some other form (e.g., translation to code or simulation scripts). All of the modeling languages presented in the paper are developed and hosted within the GME.

2.2 C-SAW: A Model Transformation Engine

The paper advocates automated model transformation to address scalability concerns. The Constraint-Specification Aspect Weaver (C-SAW[3]) is the model transformation engine used in the case studies in Section 4. Originally, C-SAW was designed to address crosscutting modeling concerns [9], but has evolved into a general model transformation engine. C-SAW is a GME plug-in and is compatible with any meta-model; 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 and briefly explained in Figures 3 and 5.

---

[2] The GME is an open-source meta-programmable tool that is available from the following website: http://escher.isis.vanderbilt.edu/tools/get_tool?GME

[3] The C-SAW plug-in, publications, and video demonstrations are available at the following website: http://www.cis.uab.edu/gray/Research/C-SAW/

2.3 Related Work

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 (example surveys can be found in [4, 15]). 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. Other combinations of toolsuites are likely to offer similar capabilities.

There are several approaches to model transformation, such as graphical languages typified by graph grammars (e.g., GReAT [1] and Fujaba [7]), or a hybrid language (e.g., the ATLAS Transformation Language [3] and Yet Another Transformation Language [14]). 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 typically 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 renders complicated syntax and semantics for a hybrid language.

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 in intent 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.

3. Alternative Approaches to Model Replication

This section provides a discussion of key characteristics of a model replication technique. 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 a model transformation engine to support scalability of models through replicating transformations.

3.1 Key Characteristics for a Replication Approach

An approach that supports model scalability through replication should have the following desirable characteristics: 1) retains the benefits of modeling, 2) general

across multiple modeling languages, and 3) 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 ability to perform analysis (e.g., model checking and verification of system properties) in a way that would otherwise be difficult at the implementation level. A second advantage is the opportunity to explore various design alternatives. A model replication technique should not remove these benefits. That is, the replication mechanism and tool support should not perform scalability in such a way that analysis and design exploration is not possible. This seems to be an obvious characteristic to desire, but we have observed replication approaches that void 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. A side benefit of such generality is that a class of users can become familiar with a common replicator technique that can be applied to many modeling languages they use.

C3. Flexible to support user extensions: 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 notation or language that can be manipulated by an end-user. In contrast, replicator techniques that are hard-coded and unable to be extended restrict the impact for reuse, thus limiting the value of the time spent on creating the replicator.

The next subsection will compare existing replicator approaches to these characteristics.

3.2 Existing Approaches to Support Model Replication

From our past experience in applying MIC to DRE modeling, the following categories of techniques represent alternative approaches to support replicators: 1) an intermediate phase of replication within a model compiler, 2) domain-specific model compiler for a particular modeling language, and 3) specification of a replicator using a model transformation engine. Each of these approaches is discussed in this subsection and compared to the desiderata mentioned in Section 3.1.

A1. Intermediate stage of model compilation: As a model compiler performs its translation, it typically traverses a parse tree (containing an internal representation of the model) through data structures and APIs provided by the host modeling tool. Several model compilers can be con-

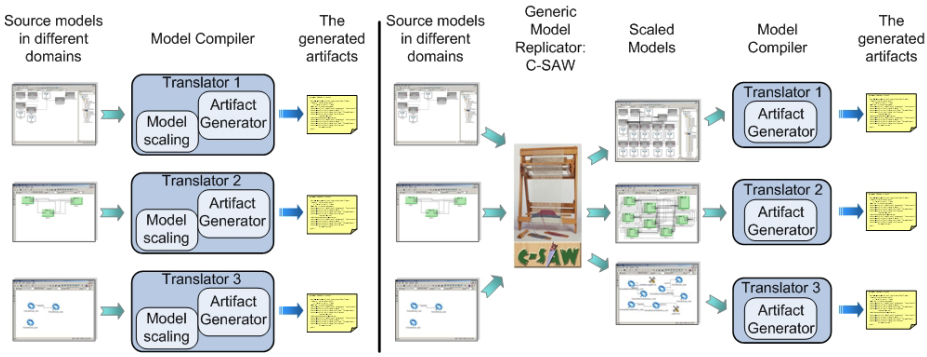

Fig. 1. Alternative Approaches for Scaling Models

constructed that generate different artifacts from the same model. 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 parse tree can be converted to an intermediate representation that can be expanded to address the desired scalability. This idea is represented in the left-hand side of Figure 1.

This is the least satisfying solution to replication and violates all three of the desired characteristics enumerated in Section 3.1. 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. Thus, analysis and design alternatives are not made available to the end-user for further consideration. Furthermore, the pre-processing rules are hard-coded into the model compiler and offer little opportunity for reuse across other modeling languages. In general, this is the least flexible of all approaches that we considered.

A2. Domain-specific model compiler to support replication: A model compiler is not only capable of synthesizing to an external artifact, but is also able to alter the current model structure through API calls. Another approach to model scalability is to construct a model compiler that is capable of replicating the models as they appear in the host modeling tool. 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.

This approach has a few drawbacks as well. Because the replication rules are domain-specific and hard-coded into the model compiler, the developed replicator has limited use outside of the intended modeling language. Although generality across modeling languages is lost, some replicators

based on this approach may have means to parameterize certain parts of the replication process (e.g., the replicator may request the size to scale, or the name of specific elements that are to be scaled).

A3. Replication with a model transformation specification: A special type of model compiler within the GME is a plug-in that can be applied to any meta-model (i.e., it is domain-independent). The C-SAW model transformation engine (see Section 2.2) is an example of a plug-in that can be applied to any modeling language. C-SAW executes as an interpreter and renders all transformations (as specified in the ECL) back into the host modeling tool. The ECL can be altered very rapidly to analyze the affect of different degrees of scalability (e.g., the affect on performance when the model is scaled from 256 to 512 nodes).

This third approach to replication advocates the use of a model transformation engine like C-SAW to perform the replication (please see the right-hand side of Figure 1 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 specified in a way that can be easily modified, as opposed to a hard-coded rule in the approaches described in A1 and A2. With a model transformation engine, a code generator is still required for each domain (see "Artifact Generator" in the right-hand side of Figure 1), but the scalability issue is addressed independently of the modeling language. Our most recent efforts have explored technique A3 on several existing modeling languages as described in the next section.

4. Case Studies in Scalability with Model Replicators

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

- System Integration Modeling Language, 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.
- Event QoS Aspect Language, which has been used to configure a large collection of federated event channels for mission computing avionics applications.

In addition to the above cases studies, our initial exploration into scalability of models was performed for a different modeling language representing unmanned air vehicles to address various quality of service concerns related to transmitted video (e.g., bandwidth and frame size adjustment). Space limitations prohibit further discussion of this third example.

4.1 Scaling the System Integration Modeling Language

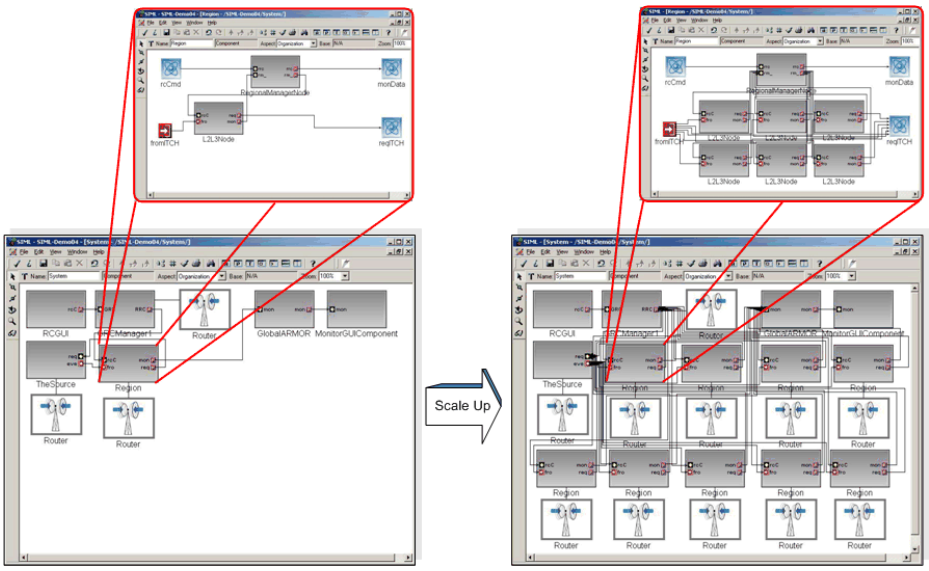

Fig. 2. Visual Example of SIML Scalability

The System Integration Modeling Language (SIML) is a language developed to specify configurations of large-scale fault tolerant data processing systems [16]. Features of SIML include hierarchical component decomposition and dataflow modeling with point-to-point and publish-subscribe communication between components. There are several rules defined by the SIML meta-model:

- 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

The *local process group* is the set of processes that run the set of critical applications to perform the system's overall function. In a data processing network, the local process group would include the algorithmic tasks to perform 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. As the SIML language itself is used to describe configurations of highly scalable architectures, it embodies some patterns of scalability as a by-product of the domain for which it was created. These patterns include the one-to-many relationship between system and regional managers, and also a one-to-many

relationship between regional and local process group managers. These relationships are well defined. Because this relationship can be captured, it should be feasible to perform automatic generation of additional local process groups and/or regions to create larger and more elaborate system models.

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. The left-hand side of Figure 2 shows a simple SIML base model that captures a system composed of one region and one local node 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). Consider this example when the system is increased to 9 regions with 6 local process groups per region. Such replication involves the following:

- Replication of the local process group models
- Replication of entire region models and their contents
- Generation of communication connections between 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-hand side of Figure 2. This example scales to just 9 regions and 6 nodes per region simply because of the printed space to visualize the figure. 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 2500 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 of design alternatives impractical.

ECL Transformation to Scale SIML: The scalability illustrated in Figure 2 can be performed with a model transformation, as illustrated by the ECL specification shown in Figure 3. 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 meta-model for less than a few hours.

The ECL transformation specification is composed of an aspect and several strategies. An aspect serves as the starting point of a transformation, and a strategy is used to specify the computation entities to perform a particular transformations task. In Figure 3, the aspect "`Start`" (Line 1) invokes two strategies, "`scaleUpNode`" and "`scaleUpRegion`" in order to replicate the local process group node ("`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 nodes based on the given name "L2L3Node." 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 the similar mechanism as above.

```
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 }
```

Fig. 3. ECL Model Transformation to Perform Replication Shown in Figure 2

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 specified can be modified directly. This is not the case in approaches A1 and A2 from Section 3.2 because the replication semantics are hard-coded into the model compiler.

4.2 Scaling the Event QoS Aspect Language

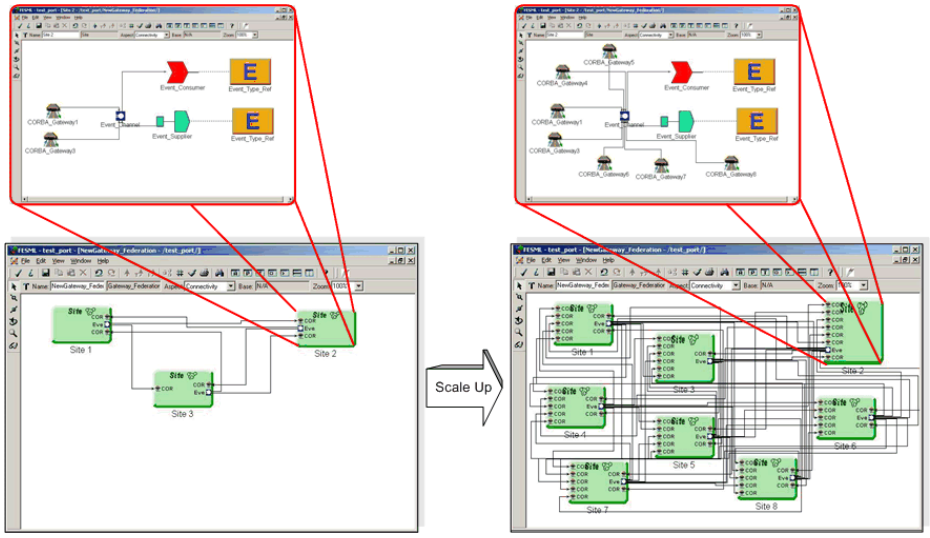

Fig. 4. Illustration of Replication in EQAL

The Event QoS Aspect Language (EQAL) [5] is a DSML for graphically specifying publisher-subscriber service configurations for large-scale DRE systems. Publisher-subscriber mechanisms, such as event-based communication models, are particularly relevant for large-scale DRE systems (e.g., avionics mission computing, distributed audio/video processing, and distributed interactive simulations) because they help reduce software dependencies and enhance system composability and evolution. In particular, the publisher-subscriber architecture of event-based communication allows application components to communicate anonymously and asynchronously. The publisher-subscriber communication model defines three software roles:

- *Publishers* generate events to be transmitted
- *Subscribers* receive events via hook operations
- *Event channels* accept events from publishers and deliver events to subscribers

The EQAL modeling environment consists of a GME meta-model that defines the concepts of publisher-subscriber systems, in addition to several model compilers that synthesize middleware configuration files from models. The EQAL model compilers automatically generate publisher-subscriber service configuration files and component property description files needed by the underlying middleware.

The EQAL meta-model defines a modeling paradigm for publisher-subscriber service configuration models, which specify quality of service (QoS) configurations, parameters, and constraints. For example, the EQAL meta-model contains a distinct set of modeling constructs for building a federation of real-time event services supported by the Component-Integrated ACE ORB (CIAO) [8], which is a component middleware platform targeted by EQAL. A federated event service allows sharing of filtering information to minimize or eliminate the transmission of unwanted events to a remote entity. Moreover, a federated event service allows events that are being communicated in one channel to be made available on another channel. The channels typically communicate through CORBA Gateways, UDP, or IP Multicast. Figure 4 illustrates the modeling concepts provided by EQAL including CORBA Gateways and other entities of the publish-subscribe paradigm (e.g., event consumers, event suppliers, and event channels) to model a federation of event channels in different sites.

```
1  //traverse the original sites to add CORBA_Gateways
2  //n is the number of the original sites
3  //m is the total number of sites after scaling
4  strategy traverseSites(n, i, m, j : integer)
5  {
6    declare id_str : string;
7    if (i <= n) then
8      id_str := intToString(i);
9      rootFolder().findModel("NewGateway_Federation").findModel("Site " + id_str)
10                                                    .addGateWay_r(m, j);
11     traverseSites(n, i+1, m, j);
12   endif;
13 }
14
15 //recursively add CORBA_Gateways to each existing site
16 strategy addGateWay_r(m, j: integer)
17 {
18   if (j<=m) then
19     addGateWay(j);
20     addGateWay_r(m, j+1);
21   endif;
22 }
23
24 //add one CORBA_Gateway and connect it to Event_Channel
25 strategy addGateWay(j: integer)
26 {
27   declare id_str : string;     declare ec, site_gw : object;
28   id_str := intToString(j);
29   addAtom("CORBA_Gateway", "CORBA_Gateway" + id_str); //create one CORBA_Gateway
30   ec := findModel("Event_Channel");  site_gw := findAtom("CORBA_Gateway" + id_str);
31   addConnection("LocalGateway_EC", site_gw, ec);
32 }
```

Fig. 5. ECL Fragment to Perform the First Step of Replication in EQAL

The scalability issues in EQAL arise when a small federation of event services must be scaled to a very large system, which usually accommodates a large number of publishers and subscribers. It is conceivable that EQAL modeling features, such as the event channel, the associated QoS attributes, connections and event correlations

must be applied repeatedly to build a large scale federation of event services. Figure 4 shows a federated event service with 3 sites, which is then scaled up to federated event services with 8 sites. This scaling process includes three steps:

- Add 5 CORBA_Gateways to each original site
- Repeatedly replicate one site instance to add 5 more extra sites, each with 5 CORBA_Gateways
- Create the connections between all of the 8 sites

The above process can be automated with an ECL transformation that is applied to a base model with C-SAW. Figure 5 shows a fragment of the ECL specification for the first step, which adds more Gateways to the original sites. The other steps would follow similarly using ECL. The size of the replication in this example was kept to 5 sites so that the visualization could be rendered appropriately in Figure 4. The approach could be extended to scale to hundreds or thousands of sites and gateways.

5. Conclusion

This paper has demonstrated the effectiveness of using a general model transformation engine to specify replicators that assist in scaling models. 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 plug-ins that are specific to a domain and scalability issue). The case studies presented in this paper highlight the ease of specification and the general flexibility provided across domains.

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 ECL focuses on the construction of testing and debugging utilities within C-SAW to ensure the correctness of the ECL transformation specifications [13].

6. Acknowledgments

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

References

1. Aditya Agrawal, Gábor Karsai, and Ákos Lédeczi, "An End-to-End Domain-Driven Software Development Framework," Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) – Domain-driven Track, Anaheim, CA, October 2003, pp. 8-15.
2. Don Batory, Jacob Neal Sarvela, and Axel Rauschmeyer, "Scaling Step-Wise Refinement," *IEEE Transactions on Software Engineering*, June 2004, pp. 355-371.
3. Jean Bézivin, F. Jouault, and P. Valduriez, "On the Need for MegaModels," *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, Vancouver, BC, October 2004.
4. Krzysztof Czarnecki, and Simon Helsen, "Classification of Model Transformation Approaches," *OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, CA, October 2003.
5. George Edwards, Gan Deng, Douglas Schmidt, Aniruddha S. Gokhale, Bala Natarajan, "Model-Driven Configuration and Deployment of Component Middleware Publish/Subscribe Services," *Generative Programming and Component Engineering (GPCE)*, Vancouver, BC, October 2004, pp. 337-360.
6. Matthew Emerson, Janos Sztipanovits, and Ted Bapty, "A MOF-Based Meta-modeling Environment," *Journal of Universal Computer Science*, October 2004, pp. 1357--1382.
7. *The FUJABA Toolsuite*, http://www.fujaba.com
8. Aniruddha Gokhale, Douglas Schmidt, Balachandran Natarajan, Jeff Gray, and Nanbor Wang, "Model-Driven Middleware," in *Middleware for Communications*, (Qusay Mahmoud, editor), John Wiley and Sons, 2004.
9. Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, Oct. 2001, pp. 87-93.
10. John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, Venkatesh Prasad Ranganath, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," *International Conference on Software Engineering*, Portland, OR, May 2003, pp. 160-173.
11. Sven Johann and Alexander Egyed, "Instant and Incremental Transformation of Models," *Automated Software Engineering*, Linz, Austria, September 2004, pp. 362-365.
12. Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, November 2001, pp. 44-51.
13. Yuehua Lin, Jing Zhang, and Jeff Gray, "A Framework for Testing Model Transformations," *Model-Driven Software Development*, Springer, 2005.
14. Octavian Patrascoiu, "Mapping EDOC to Web Services using YATL," *8th International IEEE EDOC Conference*, Monterey, CA, September 2004, pp. 286-297.
15. Shane Sendall and Wojtek Kozaczynski, "Model Transformation – the Heart and Soul of Model-Driven Software Development," *IEEE Software, Special Issue on Model Driven Software Development*, September/October 2003 (Vol. 20, No. 5). pp. 42-45.
16. Shweta Shetty, Steve Nordstrom, Shikha Ahuja, Di Yao, Ted Bapty, and Sandeep Neema, "Systems Integration of Large Scale Autonomic Systems using Multiple Domain Specific Modeling Languages," *Engineering of Autonomic Systems*, Greenbelt, MD, April 2005.
17. Janos Sztipanovits and Gábor Karsai, "Model-Integrated Computing," *IEEE Computer*, April 1997, pp. 10-12.