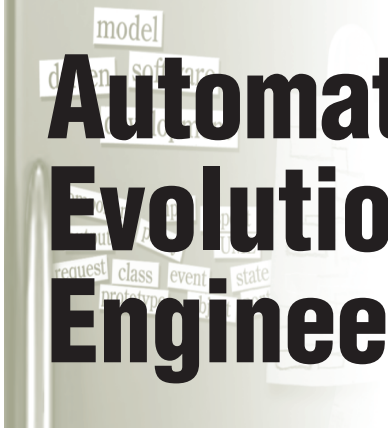

Automating Change Evolution in Model-Driven Engineering

Jeff Gray, Yuehua Lin, and Jing Zhang
University of Alabama at Birmingham

The escalating complexity of software and system models is making it difficult to rapidly explore the effects of a design decision. Automating such exploration with model transformation and aspect-oriented techniques can improve both productivity and model quality.

With the expanded focus of software and system models has come the urgent need to manage complex change evolution within the model representation.[1] Designers must be able to examine various design alternatives quickly and easily among myriad and diverse configuration possibilities. Ideally, a tool would simulate each new design configuration so that designers could rapidly determine how some configuration aspect, such as a communication protocol, affects an observed property, such as throughput.

To provide that degree of model-evolution support, a tool must accommodate two categories of changes that designers now do manually—typically with poor results. The first category comprises changes that crosscut the model representation's hierarchy.[2] An example is the effect of fluctuating bandwidth on the quality of service across avionics components that must display a real-time video stream. To evaluate such a change, the designer must manually traverse the model hierarchy by recursively clicking on each submodel. This process is tedious and error-prone, because system models often contain hierarchies several levels deep.

The second category of change evolution involves scaling up parts of the model—a particular concern in the design of large-scale distributed, real-time, embedded systems,[3] which can have thousands of coarse-grained components.[1] This type of change requires creating multiple modeling elements and connections among them. Scaling a base model of a few elements to thousands of elements requires a staggering amount of clicking and typing within the modeling tool. The ad hoc nature of this process causes errors, such as forgetting to make a connection between two replicated elements. Clearly, manual scaling affects not only modeling performance, but also the representation's correctness.

Both these change evolution categories would benefit greatly from automation. To this end, we have developed the Constraint-Specification Aspect Weaver, a generalized transformation engine for manipulating models. C-Saw is a plug-in for Vanderbilt University's Generic Modeling Environment (GME)—a configurable toolset that supports the creation of domain-specific modeling environments (www.isis.vanderbilt.edu/Projects/gme). In "Developing Applications Using Model-Driven Design Environments" on pp. 33-40, Krishnakumar Balasubramanian and coauthors describe a GME application. To address crosscutting changes, C-Saw incorporates several aspect-oriented principles.[4]

The combination of model transformation and aspect weaving provides a powerful technology for rapidly transforming legacy systems from the high-level properties that models describe. Further, by applying aspect-oriented techniques and program transformation, small changes at the modeling level can trigger very large transformations at the source code level. Thus, model engineers can explore alternative configurations using an aspect weaver targeted for modeling tools and then use the models to generate program transformation rules for adapting legacy source code on a wide scale.

0018-9162/06/$20.00 © 2006 IEEE

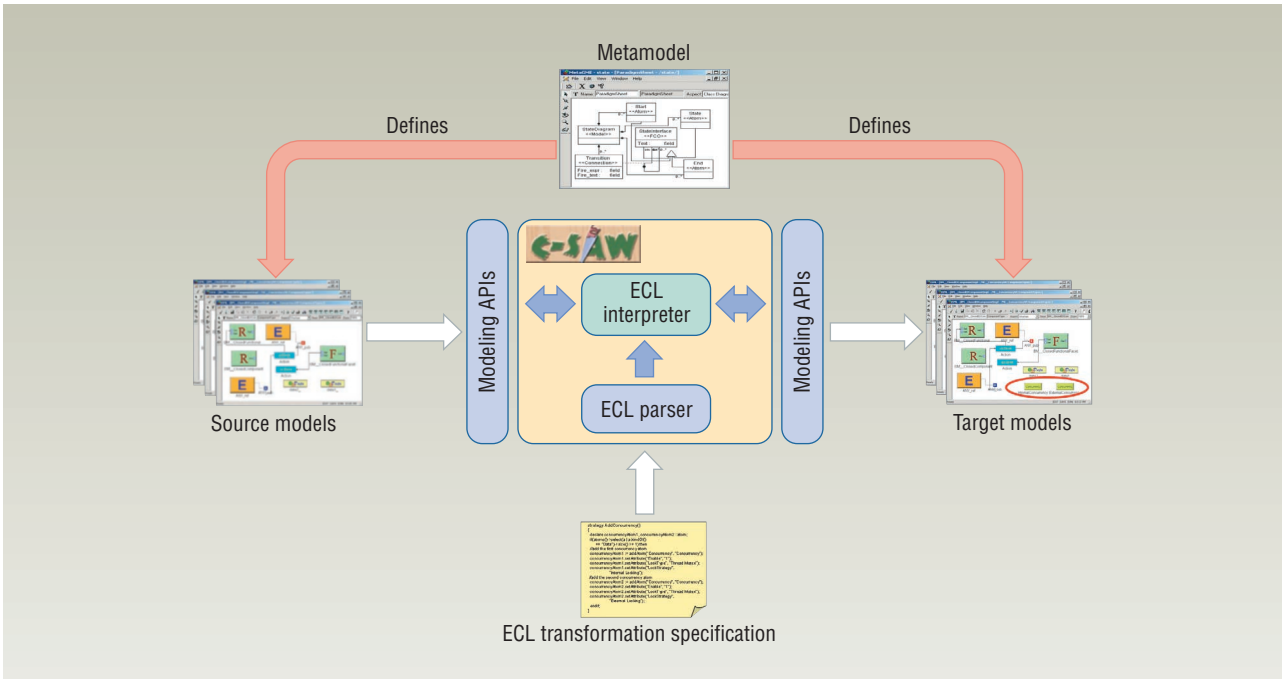

Figure 1. Model transformation in C-Saw. Source models and a transformation specification (written in the Embedded Constraint Language—ECL) serve as input to a transformation engine that derives new target models. The center box represents C-Saw and illustrates the connection between the GME API and the transformation specification language's parser and interpreter.

On the basis of our experience applying C-Saw to models from the mission-computing avionics domain, we believe that it has the potential to greatly improve productivity and decrease the errors characteristic of a manual process.

AUTOMATED MODEL TRANSFORMATION

The idea of model transformation is to enable automation that will remove accidental design complexities in the modeling process, just as software factories aim to remove accidental software development complexities.

Model transformation can take one of two approaches.[5] Model-to-code transformation generates source code (Java or C++, for example) from models, thus lowering the abstraction level of modeling artifacts and making them executable. Model-to-model transformation—this article's focus—applies a set of transformation rules that take one or more source models as input to produce one or more target models as output.[6] A typical application of model-to-model transformation is to refine an abstract model with additional details that emerge in multiple locations. From our experience, model transformation can automate many activities, which in turn will enhance engineering productivity and model quality.

Figure 1 shows how model transformation works in C-Saw. In GME, a metamodel represents the definition of the language that describes a particular domain; metamodel instances capture specific design configurations. When a model engineer invokes C-Saw from the GME toolbar, C-Saw asks the user to provide a set of files containing transformation rules that describe the location and behavior of the change to be performed on the source models.

Model transformation produces a new set of target models that contain adaptations spread across the model hierarchy according to the transformation specification. The user can undo these adaptations and perform new changes to the models by selecting different transformation rules. In essence, the user can distribute any system property endemic to a specific domain across the model hierarchy. C-Saw also aids in structural changes within the model, even when the required change involves issues that crosscut model components. Additional information about C-Saw is available at www.cis.uab.edu/gray/Research/C-SAW.

Model transformation language

To support the automation of model transformation, several modeling tools provide different techniques to assist an engineer in performing a restricted set of model changes. As the sidebar "Model Transformation Languages" describes, a specific language for model engineers and developers to use for specifying and executing the desired transformations is a critical requirement.[7]

Such a language must have two core characteristics. First, it must *be focused on a particular domain or user context*. Users must be able to describe a transformation using concepts from their own domain. To maximize conciseness and comprehension, the language must have core abstractions that are intuitive and cover the

Model Transformation Languages

In general, tools define model transformations using direct model manipulation[1] or intermediate representation or by adopting a *specialized transformation language*. Because the transformation language provides a set of constructs for explicitly specifying transformation behavior, it enables a more concise description than is possible by manipulating a model through a general programming language.

Both academic and industrial researchers have proposed many such languages that model engineers can use to define transformation rules and rule application strategies, either graphical or textual. Model transformation languages can also be either imperative or declarative.[2] In addition to our Embedded Constraint Language, two transformation languages are worth noting.

The *Graph Rewriting and Transformation Language*[3] is a graphical transformation language based on graph grammars, the approach that Atom[3] also adopts.[4] In GReAT, model engineers treat models as graphs and specify model transformations as graph transformations. Applying transformation rules, which are essentially rules for rewriting graphs, realizes the graph transformations. A transformation rule consists of two parts: the left-hand side (LHS) is a graph to match, and the right-hand side (RHS) is a replacement graph. If GReAT finds a match for the LHS graph, a rule is fired, which results in replacing the matched subgraph with the replacement RHS graph. GReAT provides graphical notations to specify graph patterns, model transformation rules, and the control flow of transformation execution.

ATLAS Transformation Language (ATL)[5] is a hybrid transformation language that combines declarative and imperative constructs. Model engineers use declarative constructs to specify source and target patterns as transformation rules (to filter model elements, for example), and imperative constructs to implement sequences of instructions (assignment, looping, and conditional constructs, for example).

Choosing the most suitable transformation language style depends on factors such as model size, designer experience, and transformation task. Each style has its own set of tradeoffs. For example, with a pattern-based graph transformation language, an engineer might be able to express a complex pattern more concisely than with an imperative textual language, but the pattern-matching engine could slow performance in a large model. Understanding these tradeoffs will help model engineers more effectively plan for model evolution.

References

1. S. Sendall and W. Kozaczynski, "Model Transformation—The Heart and Soul of Model-Driven Software Development," *IEEE Software*, Sept./Oct. 2003, pp. 42-45.
2. K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," *Workshop Generative Techniques in the Context of Model-Driven Architecture*, 2003; www.softmetaware.com/oopsla2003/czarnecki.pdf.
3. A. Agrawal, G. Karsai, and A. Lédeczi, "An End-to-End Domain-Driven Software Development Framework," *Proc. 18th Ann. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 03), ACM Press, 2003, pp. 8-15.
4. J. de Lara and H. Vangheluwe, "AToM^3: A Tool for Multi-Formalism and Meta-Modeling," *Proc. European Joint Conf. Theory and Practice of Software* (ETAPS 02), LNCS 2306, Springer-Verlag, 2002, pp. 174-188.
5. I. Kurtev, K. van den Berg, and F. Jouault, "Rule-Based Modularization in Model Transformation Languages Illustrated with ATL," *Proc. ACM Symp. Applied Computing* (SAC 06)—*Model Transformation Track*, ACM Press, 2006, to be published.

largest possible range of situations that reflect current modeling practice. A user-centered transformation language should have a small (ideally minimal) set of concepts and constructs, but be powerful enough to express a complete set of desired transformation activities.

Second, a model transformation language must also address *specific model evolution tasks*. It should have full power to specify all types of modeling objects and transformation behaviors, including model navigation, querying, and modification. This requires both a robust type system and a set of functionally rich operators.

In addition to full expressiveness, such a language should provide specific constructs and mechanisms for users to describe model transformations in an efficient way. In other words, a model transformation language is a domain-specific language that captures all the model evolution features, but also provides constructs that let users refer to concepts from their own domains.

As Figure 1 shows, C-Saw uses the Embedded Constraint Language (ECL), an extension to the Object Constraint Language (OCL), to meet these requirements. ECL reflects concepts from the user's modeling domain and lets model engineers refine the model in a stepwise manner.[8] Model engineers can write transformations concisely and intuitively because the transformations refer to domain elements that they recognize. The ECL constructs that support such transformation include a type system, an element selection mechanism, and a set of operators to manipulate the source models.

Type system. In GME terminology, a model is structurally a graph in which its elements are the nodes and the relationships between elements are the edges. An atom is a model element that cannot contain any other model elements. A model can be a container in which there is a combination of submodels and atoms. Both models and atoms have a set of properties.

ECL provides a basic type system to describe values and model objects in a transformation. The data types in ECL include standard primitive data types (for example, Boolean, integer, real, and string) and model object types (for example, atom, model, object, and connection) that can reflectively refer to domain concepts specified in the metamodel.

Element selection. A common activity during model transformation is to find model elements that might need modifying. To locate these elements, model engineers can use querying or pattern matching. Querying evaluates a predicate expression over a model, returning only the elements for which the expression holds. Pattern matching binds a term or a graph pattern containing free variables against the model.

ECL supports model queries by providing the `select` operator, a set of special operators to select a collection of model objects, and a set of operators to find a single model object. The `select` operator specifies a selection from an existing collection, which can be the result of previous operations and navigations. The result of the `select` operation is always a subset of the original collection.

ECL provides a basic type system to describe values and model objects in a transformation.

Numerous operators support model aggregation: `models(<expression>)` selects all the submodels that satisfy the constraint the expression specifies. Other query operators include `atoms(<expression>)`, `connections(<expression>)`, and `attributes (<expression>)`. Operators like `findAtom` and `findModel` find a single atom or model; `source` and `destination` are functions that return the source and the destination objects in a connection.

Transformation operations. The primary operations in this category support the engineer in adding and removing objects and changing object properties. Standard OCL can specify assertions about a model, but is not intended to describe model changes. As an extension of OCL, ECL provides a set of operators for changing the model's structure. For adding or removing elements (a model, atom, or connection, for example), ECL provides `addModel`, `addAtom`, `addConnection`, and `removeModel`, `removeAtom`, and `removeConnection`. The `setAttribute` function aids in changing the value of any element attribute.

CROSSCUTTING DESIGN PROPERTIES

When a concern spreads across an artifact, a model is difficult to comprehend and change. *Aspect-oriented software development*[4] offers techniques to modularize concerns that crosscut system components. Although the application of AOSD originally focused on programming languages, the community investigating aspect-oriented modeling is growing. Indeed, the eighth AOM workshop will take place in March 2006 (http://dawis.informatik.uni-essen.de/events/AOM_AOSD2006).

The most prominent work in aspect modeling concentrates on notational aspects for the Unified Modeling Language,[9] but tools could also provide automation using AOSD principles. In fact, one of our motivations for developing C-Saw was the need to specify constraints that crosscut the model of a distributed real-time embedded system,[2] such as the one in Figure 2. We have since evolved C-Saw into a general model transformation engine that addresses a broad range of transformations.

The top of Figure 2 shows the interaction among components in a mission-computing avionics application modeled in the Embedded Systems Modeling Language (http://escher.isis.vanderbilt.edu/tools). The model illustrates a set of avionics components (Global Positioning Satellite and navigational display components, for example) that collaborate to process a video stream providing a pilot with real-time navigational data. The middle of the figure shows the internal representation of two components, which reveals the data elements and other constituents intended to describe the infrastructure of component deployment and the distribution middleware. The infrastructure implements an event-driven model, in which components update and transfer data to each other through event notification and callback methods.

Among the components in Figure 2 are a concurrency atom and two data atoms (circled). Each of these atoms represents a system concern that is spread across the model hierarchy. The concurrency atom (red circle) identifies a system property that corresponds to the synchronization strategy distributed across the components. The collection of atoms circled in blue defines the recording policy of a black-box flight data recorder. Some data elements also have an attached precondition (green circle) to assert a set of valid values when a client invokes the component at runtime.

To analyze the effect of an alternative design decision manually, model engineers must change the synchronization or flight data recorder policies, which requires making the change manually at each component's location. The partial system model in Figure 2 is a subset of an application with more than 6,000 components. Manually changing a policy will strain the limits of human ability in a system that large.

With ECL, model engineers merely define a modeling aspect to specify the intention of a crosscutting concern. The `PreData2` aspect, for example, specifies that all `data2_` atoms must have a precondition defining the valid range of values:

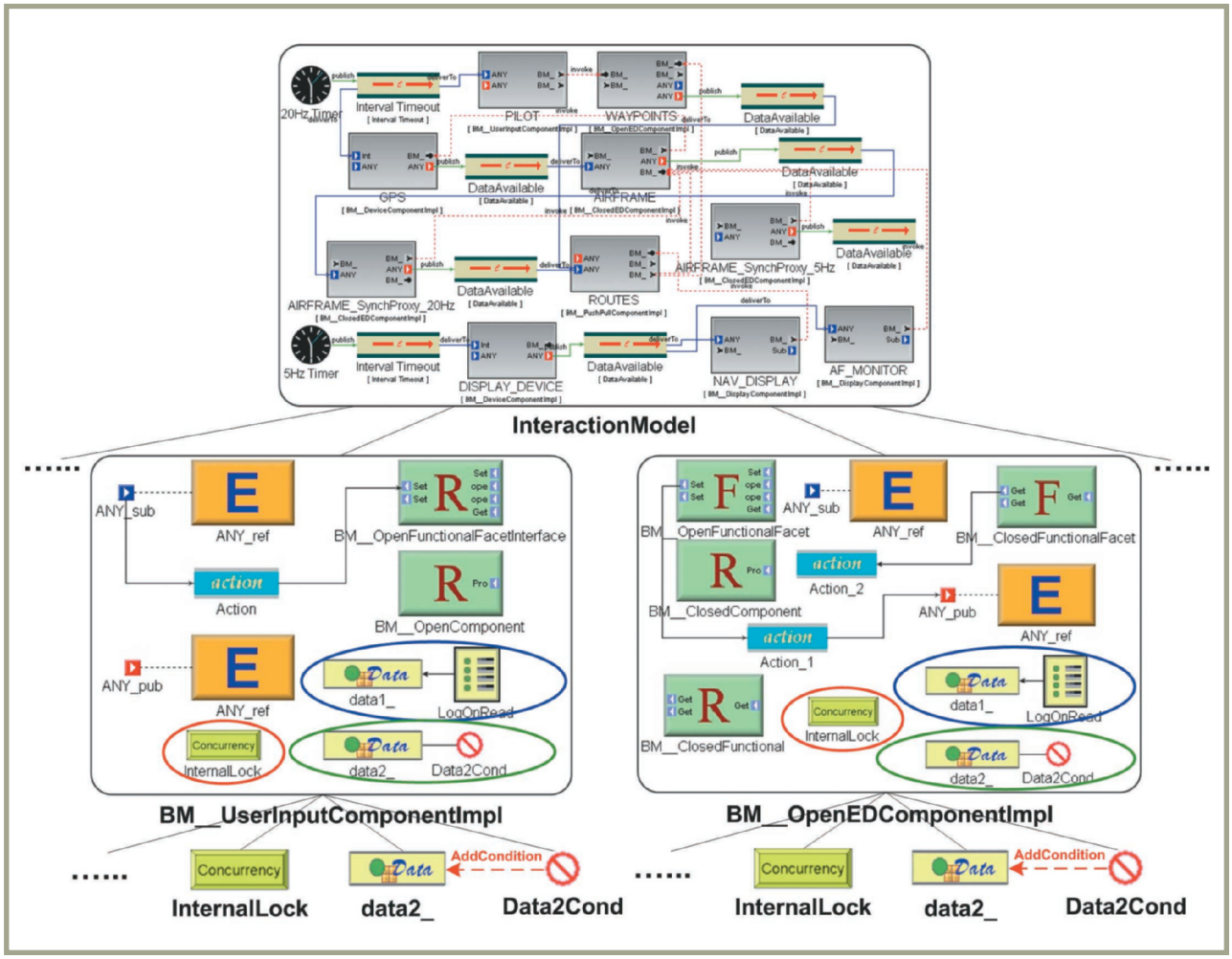

Figure 2. A subset of a model hierarchy with crosscutting model properties. Concerns related to synchronization (red circle), black-box data recording (blue circle), and preconditions (green circle) are scattered across many submodels.

```
aspect PreData2()
{
  rootFolder().findFolder
   ("ComponentTypes").models()->
  select(m|m.name().endWith("Impl"))
   ->AddPre("data2_", "Data2Cond",
   "value<200");
}
```

This aspect tells C-Saw to collect from the root folder all the models that are type `ComponentTypes`. From this collection, it then selects all the components whose name ends with `"Impl"` and applies the `AddPre` strategy.

C-Saw applies an aspect by traversing a model and matching model elements that satisfy a predicate (in this case, the selection predicate in the `PreData2` aspect). It then transforms the matched elements according to the rules in the associated *strategy*—a modular ECL construct that defines the transformation. C-Saw performs each strategy in a specific modeling context, which the aspect provides. The context can be an entire project; a specific model, atom, or connection; or a collection of assembled modeling elements. Using ECL, engineers can define aspects to quantify the modeling elements they want to change and apply the strategy construct to perform the required transformation.

Figure 3 shows how transformation strategies work. C-Saw begins by applying the `AddPre` strategy to each component it has selected from the `PreData2` aspect. `AddPre` finds all the `Data` component atoms and matches the name passed in as an argument from the aspect. The `AddCond` strategy attaches a new precondition to all `Data` atoms that match this predicate. In line 12, `AddCond` retrieves a placeholder for the actual `Data` to be transformed. It then reflectively obtains the parent component that contains the `Data` atom (line 13). The transformation sequence defined in lines 15 through 17 creates a new `Condition` atom (line 15) and sets attributes defining a precondition with an associated expression (lines 16 and 17). Finally,

```
 1 strategy AddPre(atomName, condName, condExpr :
                                          string)
 2 {
 3   atoms()->select(a | a.kind() == "Data" and
     a.name() == atomName)->
 4        AddCond(condName, condExpr);
 5 }
 6
 7 strategy AddCond(condName, condExpr : string)
 8 {
 9   declare p : model;
10   declare data, pre : atom;
11
12   data := self;
13   p := parent();
14
15   pre := p.addAtom("Condition", condName);
16   pre.setAttribute("Kind", "PreCondition");
17   pre.setAttribute("Expression", condExpr);
18   p.addConnection("AddCondition", pre, data);
19 }
```

Figure 3. ECL transformation to add a precondition expression to a Data atom.

`AddCond` adds a connection to the parent component that links the `Data` atom to the newly created `Condition` atom.

AUTOMATED MODEL SCALABILITY

A second form of design exploration involves scaling up different parts of a model. Scalability support within modeling tools is of utmost concern to designers of distributed real-time embedded systems. The issue of scalability affects the performance of the modeling activity, as well as the model representation's correctness.

Figure 4 is an example of applying scalability to the Event QoS Aspect Language (EQAL), which model engineers use to configure a large collection of federated event channels for mission-computing avionics applications (www.dre.vanderbilt.edu/cosmic).

The scalability issues in EQAL arise when the model engineer must scale a small federation of CORBA event services from a base model to accommodate more publishers and subscribers. As the figure shows, scaling an EQAL model from three sites and two gateways per site to a model with eight sites and

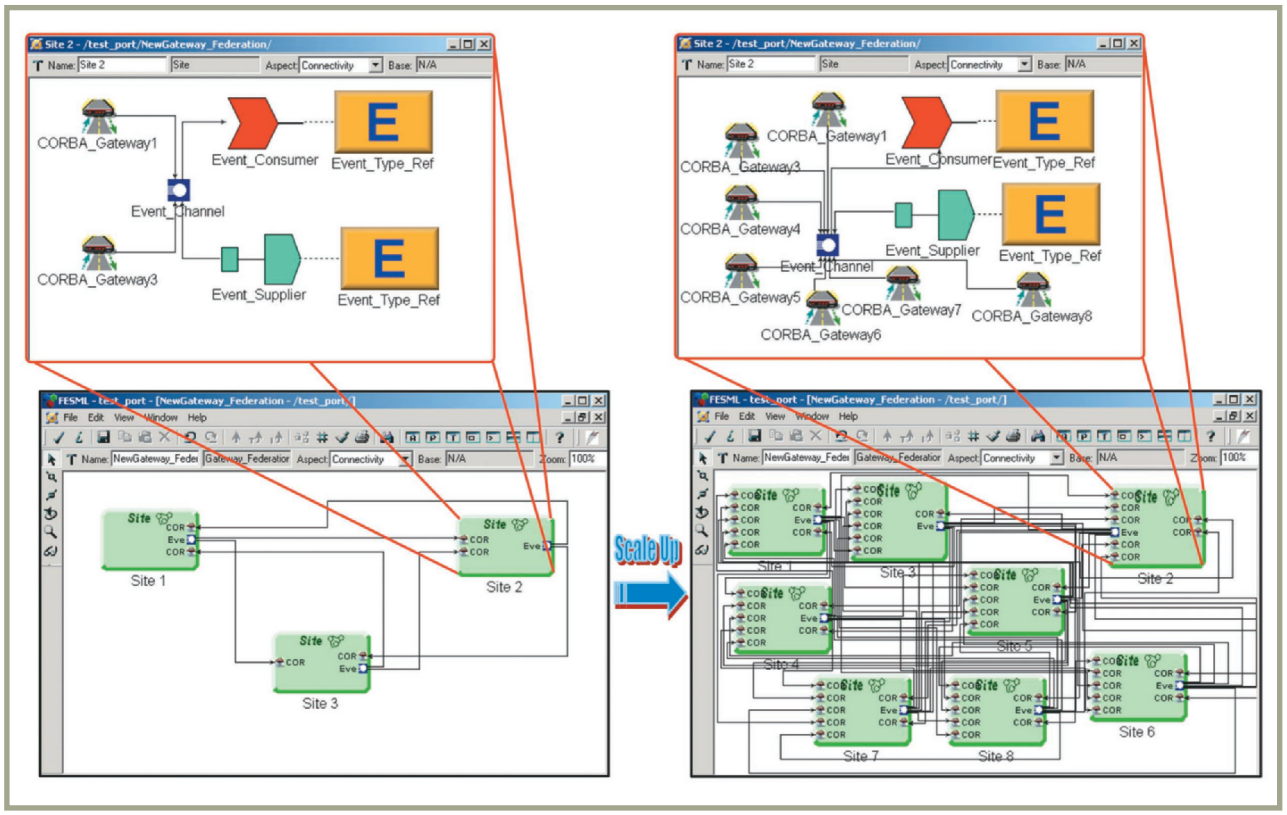

Figure 4. Replication among a federation of CORBA gateways. The model engineer is attempting to replicate three sites to eight (bottom), as well as replicate two internal sites to seven gateways (top). Such manual replication is tedious and error prone, requiring close to 300 mouse clicks.

seven gateways per site requires adding many new connections. Indeed, to perform the equivalent transformation manually, the model engineer would have to insert more than 120 new modeling elements and almost 150 connections among all the new elements. Model transformations that serve as replicators can significantly automate this manual task.[3]

Figure 5 shows the model transformation that replicates the internal CORBA gateways for a specific site. In line 1, C-Saw obtains the site and number of desired gateways from the user and passes them as parameters to the `expandSite` strategy, which then finds the specific site (line 3) and calls `addGateWay_r`, a strategy that ensures C-Saw adds the right number of gateways. `addGateWay` performs the actual transformation by creating a new gateway (line 19), locating the site's core event channel (line 20), and connecting the new gateway to the existing channel (line 21).

The collection of interacting strategies in Figure 5 accomplishes the transformation within each internal site (for example, the internal expansion of Site 2 in the top right of Figure 4). Because C-Saw strategies are reusable, the model engineer can flexibly explore a design by applying the transformation to several sites. We have also specified a set of transformations to scale the outermost number of sites (the transformation that expands sites in the bottom of Figure 4).

```
 1 strategy expandSite(site, numGW :
                                integer)
 2 {
 3  findModel("Site" + intToString
      (site)).addGateWay_r(1, numGW);
 4 }
 5
 6 strategy addGateWay_r(curr, numGW :
                                 integer)
 7 {
 8  if (curr <= numGW) then
 9    addGateWay(curr);
10    addGateWay_r(curr+1, numGW);
11  endif;
12 }
13
14 strategy addGateWay(num : integer)
15 {
16  declare site_gw : atom;
17  declare ec : model;
18
19  site_gw := addAtom("CORBA_Gateway",
    "CORBA_Gateway" + intToString(num));
20  ec := findModel("Event_Channel");
21  addConnection("LocalGateway_EC", site_gw,
                                      ec);
22 }
```

Figure 5. Model transformation to expand the number of CORBA gateways at a specific site.

Being able to pass parameters to `expandSite` is critical to defining alternative designs. In contrast, manual replication severely limits the exploration of design alternatives. Suppose the model engineer now needs to scale the base model to 20 sites with 12 gateways per site. Using a manual approach, the engineer must repeat the same task from the beginning. With C-Saw, the engineer merely inputs new parameters to the strategies.

Despite recent advances in modeling tools, many modeling tasks can still benefit from increased automation. In contrast, integrated development environments for programming languages provide many features for comprehending and evolving large code bases, including automated refactoring, code navigation, and visualization. These capabilities still need investigating within the context of modeling tools.

Model transformation can provide a common technology for implementing various kinds of model evolution. We have applied our C-Saw transformation engine to several modeling languages in the distributed real-time and embedded system domain to modularize cross-cutting properties and replicate elements of a core model. We are exploring several additions that will provide extra value to the transformation process, such as testing and debugging support for ECL to detect and find possible errors in a transformation specification.

C-Saw is just one of many model transformation approaches, which differ widely in application and results.[5] Standards might appear in the near future, however, since in 2002 the Object Management Group issued a request for proposals on query/views/transformations, which is near finalization (www.omg.org/docs/ad/05-07-01.pdf). Given the diversity of existing approaches, the effects of such standardization on general modeling practice remain to be seen.

Acknowledgments

The initial investigation of this research was made possible by previous support from the DARPA Program Composition for Embedded Systems program. Current support from the NSF CSR-SMA program is enabling the application of these ideas to the evolution of performance modeling for distributed design patterns.

References

1. S. Johann and A. Egyed, "Instant and Incremental Transformation of Models," *Proc. 19th IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 04), IEEE CS Press, 2004, pp. 362-365.
2. J. Gray et al., "Handling Crosscutting Constraints in Domain-Specific Modeling," *Comm. ACM*, Oct. 2001, pp. 87-93.
3. J. Gray et al., "Replicators: Transformations to Address Model Scalability," *Proc. Int'l Conf. Model-Driven Eng. Languages and Systems* (MoDELS 05) (formerly the UML series of conferences), LNCS 3713, Springer-Verlag, 2005, pp. 295-308.
4. R.E. Filman et al., *Aspect-Oriented Software Development*, Addison-Wesley, 2004.
5. T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," *Proc. Int'l Workshop Graph and Model Transformation*, 2005, http://tfs.cs.tu-berlin.de/gramot/FinalVersions/PDF/MensVanGorp.pdf.
6. S. Sendall and W. Kozaczynski, "Model Transformation—the Heart and Soul of Model-Driven Software Development," *IEEE Software*, Sept./Oct. 2003, pp. 42-45.
7. J. Bézivin, "On the Unification Power of Models," *J. Software and System Modeling*, May 2005, pp. 171-188.
8. D. Batory, J.N. Sarvela, and A. Rauschmeyer, "Scaling Step-Wise Refinement," *IEEE Trans. Software Eng.*, June 2004, pp. 355-371.
9. R. France et al., "An Aspect-Oriented Approach to Design Modeling," *IEE Proc. Software*, special issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, Aug. 2004, pp. 173-186.

Jeff Gray *is an assistant professor in the Department of Computer and Information Sciences at the University of Alabama at Birmingham, where he codirects research in the Software Composition and Modeling (SoftCom) Laboratory. His research interests include model-driven engineering, aspect orientation, and generative programming. Gray received a PhD in computer science from Vanderbilt University. He is a member of the IEEE and the ACM. Contact him at gray@cis.uab.edu.*

Yuehua ("Jane") Lin *is a PhD candidate in the Department of Computer and Information Sciences at the University of Alabama at Birmingham and a member of the SoftCom Laboratory. Her research interests include model transformation and supporting tools. Lin received an MS in computer science from Auburn University. She is a student member of the IEEE and the ACM. Contact her at liny@cis.uab.edu.*

Jing Zhang *is a PhD candidate in the Department of Computer and Information Sciences at the University of Alabama at Birmingham and member of the SoftCom Laboratory. Her research interests include techniques that combine model transformation and program transformation to assist in evolving large software systems. Zhang received an MS in computer science from UAB. She is a student member of the ACM. Contact her at zhangj@cis.uab.edu.*