REDUCING THE COMPLEXITY OF MODELING AND OPTIMIZING LARGE SOFTWARE SYSTEMS

Jules White
Vanderbilt University
Department of Electrical Engineering and Computer Science
Box 1679 Station B
Nashville, TN, 37235, USA
jules@dre.vanderbilt.edu
+1 251 533 9432

Douglas C. Schmidt
Vanderbilt University
Department of Electrical Engineering and Computer Science
Box 1679 Station B
Nashville, TN, 37235, USA
schmidt@dre.vanderbilt.edu
+1 615 343 7440

Andrey Nechypurenko
Siemens AG
Corporate Technology (SE 2)
Otto-Hahn-Ring 6
81739 Munich, Germany
andrey.nechypurenko@siemens.com
+49 89 636 45848

Egon Wuchner
Siemens AG
Corporate Technology (SE 2)
Otto-Hahn-Ring 6
81739 Munich, Germany
egon.wuchner@siemens.com
+49 89 636 45848

REDUCING THE COMPLEXITY OF MODELING AND OPTIMIZING LARGE SOFTWARE SYSTEMS

Model-driven development is one approach to combating the complexity of designing software intensive systems. A model-driven approach allows designers to use domain notations to specify solutions and domain constraints to ensure that the proposed solutions meet the required objectives. Many domains, however, require models that are either so large or intricately constrained that it is extremely difficult to manually specify a correct solution. This chapter presents an approach to provide that leverages a constraint solver to provide modeling guidance to a domain expert. The chapter presents both a practical framework for transforming models into constraint satisfaction problems and shows how the Command Pattern can be used to integrated a constraint solver into a modeling tool.

Keywords: Modeling Languages, Meta Model, Logic Programming, Special Purpose Languages

INTRODUCTION

Model-driven development (MDD) (Ledeczi, 2001a; Kent, 2002; Kleppe, 2003; Selic, 2003) is a promising paradigm for software development that combines high-level visual abstractions—specific to a domain—with constraint checking and code-generation to simplify the development of a large class of systems (Sztipanovits, 1997). MDD tools and techniques help improve software quality by automating constraint checking (Sztipanovits, 1997). For example, in developing a software system for an automobile, automated constraint checking can be performed by the MDD tool to ensure that components connected by the developer, such as the anti-lock braking system and wheel RPM sensors, send messages to each other using the correct periodicity. An advantage of model-based constraint checking is that it expands the range of development errors that can be caught at design time rather than during testing.

Compilers for third-generation languages (e.g., Java, C++, or C#) can be viewed as a form of model-driven development (Atkinson, 2003). A compiler takes the third-generation programming language instructions (model), checks the code for errors (e.g., syntactic or semantic mistakes), and then produces implementation artifacts (e.g., assembly, byte, or other executable codes). A compiler helps catch mistakes during the development phrase and automates the translation of the code into an executable form.

Domain-specific Modeling Languages (DSML) (Ledeczi, 2001a) are one approach to MDD that use a language custom designed for the domain to model solutions. A metamodel is developed that describes the semantic type system of the DSML. Model interpreters traverse instances of models that conform to the metamodel and perform simulation, analysis, or code generation. Modelers can use a DSML to more precisely describe a domain solution, since the modeling language is custom designed for the domain.

MDD tools for DSMLs accrue the same advantages as compilers for third-generation languages. Rather than specifying the solution in terms of third-generation programming languages or other implementation-focused terminology, however, MDD allows developers to use notations specific to the domain. With an third-generation programming language approach (such as specifying the solution in C++), high-level information (such as messaging periodicity or memory consumption) is lost. Since a C++ compiler does not understand messaging periodicity (i.e., it is not part of the "domain" of C++ programs) it cannot check that two objects communicate at the correct rate.

With an MDD-based approach, in contrast, DSML developers determine the granularity of the information captured in the model. High-level information like messaging periodicity can be maintained in the solution model and used for error checking. By raising the level of abstraction for expressing design intent, more complex requirements can be checked automatically by the MDD tool and assured at design time rather than testing time (Sztipanovits, 1997), as seen in Figure 1. In general, errors caught during the design cycle are much less time consuming to identify and correct than those found during testing (Fagan, 1999).

As model-based tools and methodologies have developed, however, it has become clear that there are domains where the models are so large and the domain constraints so intricate that it is extremely hard for modelers to handcraft correct or high quality models. In these domains, MDD tools that provide only solution-correctness checking via constraints provide few real benefits over third-generation programming language approach. Even though higher-level requirements can be captured and enforced, developers must still find ways of manually constructing a model that adheres to these requirements.

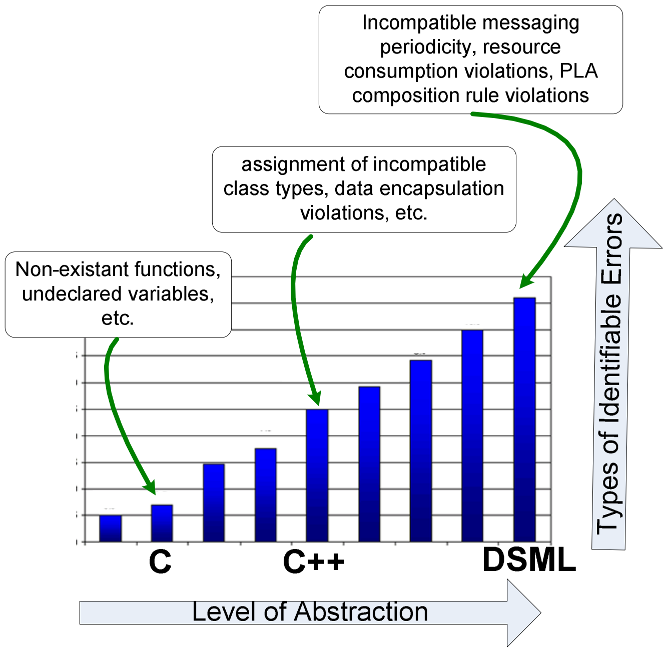


Figure 1. Complexity of Identifiable Errors Increases with Level of Abstraction

Distributed real-time and embedded (DRE) systems are software intensive systems that require guaranteed execution properties (*e.g.* deadlines), communication across a network, or must operate with extremely limited resources. Examples of DRE systems include automobile safety and aircraft autopilot systems. Inherent complexities in DRE systems is their large model sizes and the combinatorial nature of their constraints—not code construction per se. Specifying the deployment of software components to

Electronic Control Units (ECUs, which are the automotive equivalent of a CPU) in a car, while observing configuration and resource constraints, can easily generate solution spaces with millions or more possible deployments. For these large modeling problems, it is impractical (if not impossible) to create a complete and valid model manually.

To illustrate the complexities of scale, consider a group of 10 components that must be deployed to one of 10 ECUs within a car. There are $9^{\wedge 9}$ = *387,420,489 unique deployments* that could be tried. Part of the complexity of a DRE system models is how quickly the solution space grows as the number of model elements increases. Figure 2 depicts the speed at which the solution space grows for our automotive example.

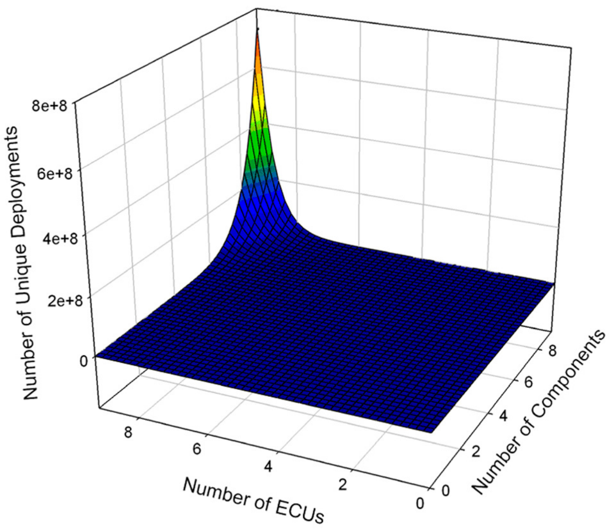


Figure 2. Number of Unique Deployments vs. Model Size

Clearly, any approach to finding a deployment that observes the constraints must be efficient and employ a form of pruning to reduce the time taken to search the solution space. A manual approach may work for a model with 5 or so elements. As shown in Figure 2, however, the solution space can increase rapidly as the number of elements grows, which render manual solutions infeasible for non-trivial systems.

Each component in an automobile typically has multiple constraints governing its placement. For example, an Anti-lock Braking System (ABS) must be hosted by a controller at least a certain distance from the perimeter of the car to enhance survivability in a crash. Moreover, the ABS will have requirements governing the CPU, memory, and bus bandwidth available on its host. When these constraints are considered for all the components, it becomes hard for modelers to handcraft correct solutions. The example in Figure 2 only has 9 components and 9 control units. Real automotive models typically contain 80 or more control units and hundreds of components. In models of this scale, manual approaches simply cannot handle the large numbers of possibilities and the complexity of the constraints.

The remainder of this chapter is organized as follows. The *Background* section illustrates the specific challenges of using MDD tools for these types of complex domains; The *Modeling Guidance* section presents techniques based on integrating constraint solvers into modeling environments that can be used to address these challenges; The *Future Research Directions* section describes future trends in modeling software intensive systems; and the final section presents concluding remarks.

BACKGROUND

Current Modeling Languages and Tool Infrastructure

There are a plethora of technologies and standards available for building MDD tools. This section explores some of the main frameworks, tools, and specifications that are available to develop model-driven processes for software systems.

Domain-independent modeling languages. On one end of the MDD tool spectrum are Unified Modeling Language (UML) (Fowler, 2000) based tools, such as IBM's Rational Rose (Quatrani, 2003), that focus on building UML and UML-profile (Fowler, 2000) based models. When using UML, all models and languages must be specializations of the UML language. UML provides a single generic language to describe all domains. The advantage of the domain-independent approach of UML-based tools is the increased interoperability between modeling platforms that can be obtained by describing models using a single modeling language and the wide acceptance of the language. New languages can be constructed on top of UML by defining profiles, which are language extensions. UML is based on the MOF metamodel specified by the OMG.

UML is well established in software development. More recently, numerous extensions to the language based on profiles have been developed. *SecureUML* (Lodderstedt, 2002) provides security related modeling capabilities to UML. *Embedded UML* (Martin, 2001) is another profile available for UML that provides DRE-specific extensions, such as timing properties of components. The UML extension approach allows developers to customize the language to meet the application domain, while still maintaining (some degree of) compatibility between tools.

Domain-specific modeling languages. On the other end of the MDD tool spectrum are domain-specific modeling language (DSML) (Ledeczi, 2001a) tools. In contrast to UML, DSML tools do not necessarily share a common metamodel or language format. This freedom allows DSMLs to have greater expressivity and handle domains (such as warehouse management, automotive design, and product line configuration), that contain concepts (such as spatial attributes) that are not easily expressed and visualized using UML-based tools. The drawback of DSMLs, however, is that choosing a language generally ties a development process not only to a specific way of representing the model but also generally to a specific tool. Although the loss of interoperability can be problematic, transformations can be written to convert between model formats and still achieve tool interoperability. In many cases, the greater expressivity gained by using a DSML can greatly improve the usability of the MDD tool.

Tools for building DSMLs. To build a DSML, a metamodeling language must be used to define the syntax of the language. A metamodel describes the rules that determine the correctness of a model instance and specifies the types that can be created in the language. The OMG's current standard is the Meta-Object Facility (MOF) (Object Management Group, 2007) language. MOF provides a metamodel language, similar to UML, that can be used to describe other new languages. MOF itself is recursively defined using MOF. MOF is a specification and therefore is not wedded to a particular tool infrastructure or language technology. Many DSMLs can be described using MOF.

Another popular metamodeling language is the Eclipse Modeling Framework's (EMF) (Moore, 2004) Ecore language. Ecore has nearly identical language constructs to

MOF but is a concrete implementation rather than a standard specification. Developers can describe DSMLs using Ecore (Moore, 2004) and then leverage EMF to automatically generate Java data structures to implement the DSML. EMF also possesses the capability to generate basic tree-based graphical editing facilities for Eclipse that operate on the Java data structures produced by EMF.

Complex diagram-like visualizations of EMF-based modeling languages can be developed using the Graphical Editor Framework (GEF) for Eclipse (Moore, 2004). GEF provides the fundamental patterns and abstractions for visualizing and interacting with a model. Editors can be developed using GEF that allow modelers to draw connections to create associations, nest elements to develop containment relationships, and edit element attributes. GEF editors are based on the Model, View, Controller (MVC) pattern (Gamma, 1995). GEF, however, requires complex graphical coding.

The Graphical Modeling Framework (GMF) (Graphical Modeling Framework, 2007), is higher level framework, built on top of GEF, that simplifies the development of graphical editors. GMF automates the construction of the controller portion of GEF editors and provides a set of reusable view classes. MVC controllers are developed using GMF by creating complex XML files that map elements and their attributes to views in the model. GMF takes the XML mappings of elements to views and generates controllers that developers can use to synchronize the model and view of the MDD tool automatically.

Even with the powerful development frameworks presented thus far, developing a visual MDD tool requires significant effort. *Meta-programmable* modeling environments (Ledeczi, 2001a) help alleviate this effort by allowing developers to specify the metamodel for a DSML visually. After the visual specification for the language is complete, the meta-programmable modeling environment can automatically generate the appropriate code and configure itself to provide graphical editing capabilities for the modeling language.

Meta-programmable modeling environments also provide complex remoting, model traversal, library, and other capabilities that are hard to develop from scratch. Two examples of these environments are the Generic Modeling Environment (GME) (Ledeczi, 2001b), which a windows-based meta-programmable MDD tool, and the Generic Eclipse Modeling System (GEMS) (Generic Eclipse Modeling System, 2007), a part of the Eclipse Generative Modeling Technologies (GMT) project. The main tradeoff in using meta-programmable modeling environments is that they tend to provide less flexibility in the visualization of the model.

Constraint Checking with OCL

Many modeling techniques rely on a constraint specification language to provide correctness checking rules that are hard to concisely describe using a graphical language. Certain types of constraints that specify conditions over multiple types of modeling elements, not necessarily related through an interface or inheritance, are more naturally expressed using a textual constraint specification language. The constraint language rules are run against instances of the UML, EMF, or other models to ensure that domain constraints are met. Constraint failures are returned to the modeler through the use of popup windows or other visual mechanisms.

The OMG Object Constraint Language (OCL) (Warmer, 1998) is a standard constraint specification for modeling technologies. OCL allows developers to specify invariants, pre-conditions, and post-conditions on types in the modeling language. For example, the OCL constraint:

```
context ECU
inv: self.hostedComponents->collect(x
                              | x.requiredRAM)->sum() < self.RAM
```

can be used to check that the sum of the RAM demands of the components hosted by an ECU do not exceed the available RAM on the ECU. The first line of the OCL rule defines the context or the type to which the OCL rule should be applied. The second part of the rule, beginning with "inv," defines the invariant condition for the rule. When there is a change to a property of a modeling element of the context type, the invariant conditions for the rules applicable to the element must be checked. Invariants that do not hold after the modification are flagged as errors in the MDD tool.

OCL works well for localized constraints that check the correctness of the properties of a single modeling element. As described earlier, however, the rule can only be used to check the correctness of the state of a modeling element and not to *derive valid states for a modeling element*, which is a process called ***backward chaining*** (Ginsberg, 1989). In a modeling context, backward chaining is a process whereby the MDD tool deduces correct modeling actions based on the domain constraints. For example, if it were possible to use the above OCL rule to backward chain, a MDD tool could not only determine whether or not an ECU was in a correct state but also, given the current state of an ECU, produce a list of components that could be hosted by the ECU without violating the rule.

For software systems with global constraints and large models, the inability of traditional modeling and constraint checking approaches, such as OCL, to not only flag errors but deduce solutions limits the utility of model-based development approaches. Backward chaining (providing modeling guidance) becomes more important as domains become more complex, and where it is thus harder to handcraft solutions.

Emerging Modeling Challenges

Deriving Solutions that meet a global constraint. The increasing proliferation of DRE systems is leading to the discovery of further hard modeling problems. These domains all tend to exhibit problems, such as scheduling with resource constraints (Yuan, 2003), that are exponential in complexity since they are different types of NP problems. A key challenge in developing effective and scalable DSMLs and models for these domains is deriving the overall organization and architecture of MDD tools and software platforms that can simultaneously meet stringent resource, timing, or cost constraints.

Mobile devices are a domain that have become widely popular and typically exhibit tight resource constraints that must be considered when designing software (Forman, 1994). Software design decisions, such as the CPU demand of the application, often have physical impacts on the device as well. For example, the scheduling of and workload placed on the CPU can affect the power consumed by the device. Poor scheduling or resource allocation decisions can therefore limit battery life (Yuan, 2003).

Determining the appropriate scheduling policies and application design decisions to handle the resource constraints of mobile devices is critical. Without the proper decisions, devices can have limited battery life and usability. Scheduling with resource constraints, however, is an NP problem (Cormen, 1990) and thus cannot be solved manually for non-trivial problems.

Adhering to non-functional requirements. Another challenge of DRE systems is that they often exhibit numerous types of non-functional QoS requirements that are hard to handle manually. For example, in automotive development, an application may have communication timing constraints on the real-time components (e.g., anti-lock braking control), resource constraints on components (e.g., infotainment systems), and feature requirements (e.g., parking assistance) (Weber, 2002). In environments with this range of QoS requirements, a correct design must solve numerous complex problems and solve them in a layered manner so the solutions are compatible.

For example, the placement of two components on particular ECUs may satisfy a timing constraint but cause a resource constraint failure for another component, such as the infotainment system. Not only must modelers be able to solve numerous types of individually challenging problems, therefore, but they must be able to find solutions that meet all of the requirements.

Another area where complex constraints are common is in configuration management, which is key in emerging software development paradigms, such as product-lines (Jaaksi, 2002) and feature modeling (Antkiewicz, 2006). In these domains, applications are built from reusable software components that interact through a common set of interfaces or framework. Applications are assembled using existing software assets for specific requirement sets. For example, in mission critical avionics product lines, such as Boeing Bold Stroke (Schmidt, 2002), the correct software component to update the HUD display is selected based on the timing, memory, and other requirements of the particular airframe that the software is being deployed to. Configuration-driven domains exhibit the same characteristics of computationally complex constraints that drive overall system organization as other complex domains.

The remainder of this chapter presents an approach to using a constraint solver integrated into a modeling environment to address these challenges. First, the chapter introduces the types of modeling assistance that can be provided to help alleviate these challenges. Second, the chapter illustrates how a constraint solver can be used to provide these types of modeling assistance. Finally, an architecture for integrating Prolog into a modeling environment as a constraint solver is described.

MODELING GUIDANCE

This section illuminates the challenges of modeling software intensive systems and then presents an approach to providing modelers with modeling guidance from a constraint solver. Specific emphasis is placed on how modeling guidance can be used to reduce the complexity of modeling software intensive systems. Finally, the chapter illustrates how a constraint solver can be integrated into a graphical modeling tool.

Measuring Domain Complexity

The complexity of modeling an arbitrary domain can be measured along the following three axes:

- **Typical Model Size in Elements:** Large Models are harder to work with using a manual approach. Clearly, modeler are more apt to make mistakes managing—and much more likely to have trouble visualizing—a domain with hundreds of model elements than one with dozens of model elements.
- **Degree of Global Constraint:** Global constraints, such as resource constraints, that are dependent on multiple modeling steps or the order of modeling steps make a domain much harder to work with. For example, a constraint requiring the deployment of an ABS component to a single ECU at a certain distance from the perimeter of the car is relatively easy to solve. It is much harder to solve constraints of an ABS component requiring its deployment to two ECUs, both a minimum distance from the outside of the car and a minimum distance from each other (for fault tolerance guarantees).
- **Degree of Optimality Required:** Optimality is hard to achieve with a manual modeling approach. In many domains, such as manufacturing, a small increase in the cost of a solution can lead to a dramatic increase in the overall cost of manufacturing when the millions of units affected by the change are considered. Many solutions must therefore be tried to find the best one. Domains that require optimal or good answers are much more challenging to model.

The three axes described above can be used to categorize and evaluate different modeling domains. The difficulty of modeling a domain can be viewed as the distance of the domain from the origin when plotted according to its degree of global constraint, degree of required model optimality, and typical model size, as shown in Figure 3.

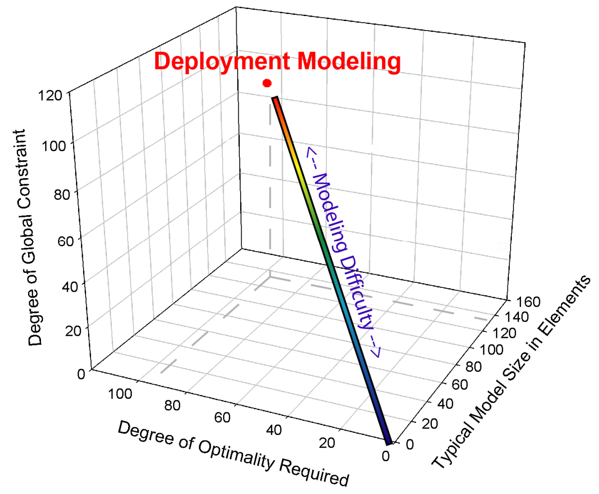


Figure 3. Axes of Measuring Modeling Complexity

Key Challenges of Complex Domains

The key reasons that manual modeling approaches do not scale as a modeling domain moves further and further along the axes, shown in Figure 3, away from the origin is:

1. When there are thousands, millions, billions, or more possible ways that a model can be constructed and few correct ones, finding a valid solution is hard.

2. A valid solution may not be a good solution in these domains. Often, a modeler may find a solution that is valid but is far from the optimal solution. Automation and numerical methods, such as the Simplex method (Nelder, 1965), are needed to efficiently search the solution space and find good candidates. A human modeler cannot effectively search a solution space manually once it grows past a certain magnitude.
3. For large models, manual construction methods, such as pointing and clicking to intricately connect hundreds or more components, are tedious and error prone.
4. Often, global constraints rely on so much information that not all of the relevant bits of information can be seen at once. When not all of the information can be seen, modelers cannot make an informed decision.

Another difficulty of highly combinatorial domains is that although modelers may create a model that satisfies the domain constraints, the model may be considered poor in quality. For example, a modeler creating a deployment of components to ECUs could easily select a scheme that utilized far more ECUs than the true minimum number required to host the set of components. For domains, such as automotive manufacturing, each modeling decision can have significant cost consequences for the final solution. For example, if a model can be constructed that uses three fewer control units to host the car's components and consequently saves $100 in manufacturing costs, millions of dollars in overall cost reduction for all cars of this make that are manufactured can be achieved. In these cases, it is crucial to not only find a correct solution but to find a cost effective one.

The difficulty of finding a good solution is that with large models and complex global constraints, modelers are lucky to find *any* valid solution. Since finding a single solution is incredibly challenging, it becomes infeasible or cost prohibitive to produce scores of valid solutions and search for an optimal one. Even if the set of valid solutions is large, there are numerous numerical methods to search for a solution with a given percentage of optimality. These methods, however, all rely on the ability to generate large numbers of valid solutions and are not possible without automation.

In domains with large models and intricate constraints, modelers must be able to see hundreds of modeling moves into the future to satisfy a global constraint or optimize a cost. The more localized a modelers decisions are and the less distant they peer into the future, the less chance there is that a correct or good solution will be found. Good local decisions, also known as "greedy decisions," do not necessarily produce a globally good decision.

For example, consider a simple model that determines the minimum number of ECUs needed to host a set of components. Assume that there are two types of ECUs, one that costs $10 and can host 2 components and another that costs $100 and can host 42 components. If modelers are deploying using a myopic view and not peering into the future, they will select many $10 ECUs and create a solution that costs $210, rather than looking ahead and choosing two $100 controllers for a final cost of $200. Making a series of locally good decisions may not produce the overall best decision (Cormen, 1990).

Solution: Integrating Constraint Solvers and MDD Tools

An MDD tool provides a visual language for a developer to build a solution specification. An instance of a visual model contains modeling entities or elements, similar to OO classes, and different visual queues (e.g. connections, containment) specifying relationships between the elements. For example, a connection between a component and an ECU specifies deployment in the automotive modeling example from the Introduction section.

The key objective of a modeler is to add the right model entities and relationships between the entities so that they create a solution that meets the application requirements. Modelers express relationships between entities by drawing connections between them, placing entities within each other for containment, or other visual means. For each relationship that a modeler creates between entities, such as deployment, the modeler must find the right source and target for the relationship so that the relationship satisfies any constraints placed on it. In the example of deploying components to ECUs, the modeler must only draw a connection from a component to an ECU that has the OS and resource capabilities to support the component.

As has been shown in the *Introduction*, *Background*, and *Measuring Domain Complexity* sections, the large size of DRE models and their complex constraints can make manually finding the right endpoints for these relationships, such as deployment, infeasible. To address the scalability challenges of manual modeling approaches presented in the aforementioned sections, this section outlines how a constraint solver can be integrated with an MDD tool to help automate the selection of endpoints for relationships between model entities.

In the context of modeling, a constraint solver is a tool that takes as input one or more model elements, a goal that the user is attempting to achieve, and a set of constraints that must be adhered to while modifying the elements to reach the goal. As output, the constraint solver produces a new set of states for the model elements that achieves the desired goal while adhering to the specified constraints. For example, a set of components can be provided to a constraint solver along with the deployment requirements (constraints) of the components. The goal can then be set to "all components connected to an ECU." The constraint solver will in turn produce a mapping of components to ECUs that satisfies the deployment constraints.

The remainder of this section first outlines the different type of modeling assistance that an MDD tool and integrated constraint solver can provide to a user . Next, the section discusses how a user's actions in an MDD tool can be translated into constraint satisfaction problems (CSPs) so that a constraint solver can be used to automatically derive the correct endpoints for the relationships the user wishes to create. Finally, the section illustrates an architecture for integrating Prolog as a constraint solver into an MDD tool.

Modeling Assistance

There are two types of constraint solver guidance that can be used to help modelers produce solutions in challenging domains: *local guidance* and *batch processes*. Local guidance is a mechanism whereby the constraint solver is given a relationship and one endpoint of the relationship and provides a list of valid model entities that could serve as the other endpoint for the relationship. One example is that a constraint solver could be provided a deployment relationship and a component and return the valid ECUs that could be attached to the other end of the connection. This type of local guidance for deploying components is shown in Figure 4.

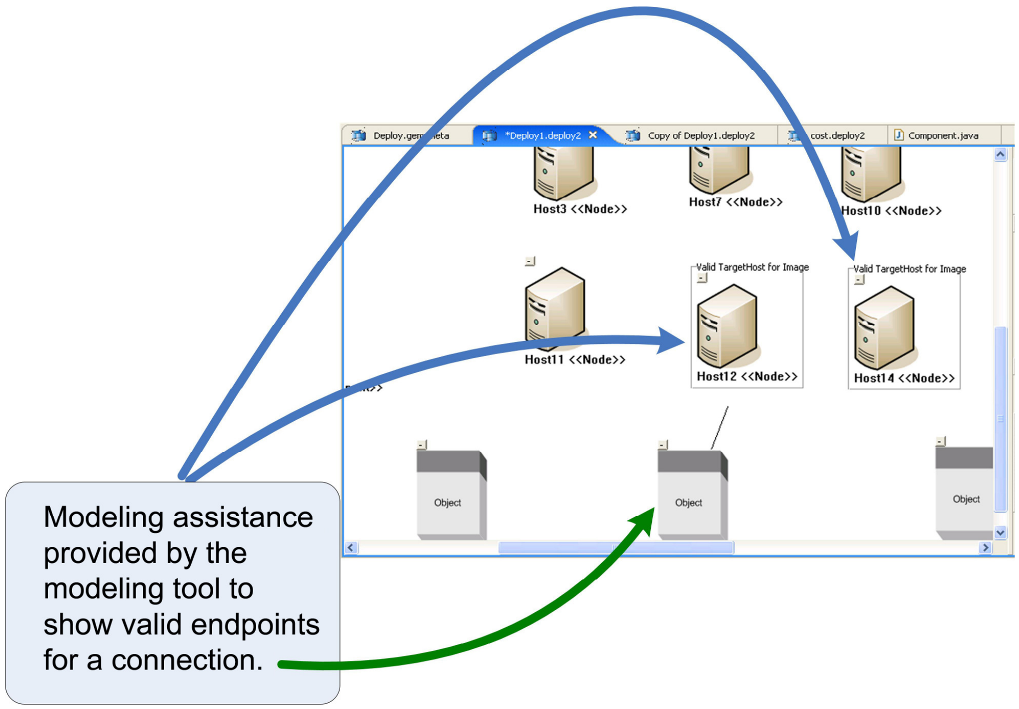


Figure 4. Local Modeling Guidance

The second type of modeling guidance is for deriving endpoints for a group of relationships so that the group as a whole satisfies a global constraint. An example of a batch process would be to connect each component to an ECU in a manner such that the no ECU hosts more components than its resources can support. A batch process takes an overall goal that the modeler is trying to achieve, such as all components connected to an ECU, and creates a series of relationships on behalf of the user to accomplish that goal. By offering both local guidance and batch processes, a MDD tool can help users to accomplish both small incremental refinements to a model and large goals covering multiple modeling steps.

Local Guidance

Local guidance helps modelers correctly complete a single modeling step. A single modeling step is defined as the creation of one relationship between two modeling elements. Local guidance can be implemented as a visual queue that shows the modeler the valid endpoints for a relationship that he or she is creating. For example, when a modeler creates a connection from a component to an ECU to specify where a component is deployed, the modeler must first click on the component modeling element to initiate the connection. When the connection is initiated, the constraint solver can be used to solve for the valid deployment locations for the component and the model elements corresponding to these deployment locations can be highlighted in the model.

Challenges 3 & 4 from the section *Key Challenges of Complex Domains* can be addressed with local guidance. By identifying the model elements that are valid target endpoints of the modeling action a user is performing, a modeling tool can use visual queues (*e.g.* highlighting, filtering, etc.) to show the user only the information relevant to the action. Furthermore, the modeling tool can use the list of valid targets to both help the modeler identify valid solutions (helping address challenge 1 of *Key Challenges of Complex Domains*) and to prevent the user from applying an action to an invalid target endpoint (addressing challenge 3 of *Key Challenges of Complex Domains*). With a traditional MDD approach, the correctness of a user's action is checked after completion and thus the user may have to do and undo an action multiple times before the correct target endpoint is found. By finding valid solutions before a modeler completes a modeling action, the tool can preemptively constrain (*e.g.* veto modeling actions) what modeling elements the action can be applied to and prevent tedious and error-prone manual solution searching.

Local guidance can not only provide suggestions of correct endpoints of a relationship but can provide rankings of the local optimality of each of the endpoints. For example, deployment locations could be ranked by the resource slack available on them so that modelers are led to choose deployment targets with sufficient free resources. This manner of local guidance provides a greedy strategy to modeling guidance. At each step, modelers are led towards a solution that provides the greatest immediate benefit to the model's correctness.

Correct solutions to modeling transactions of a single modeling step can be found using local guidance. In some cases, only considering single step transactions will not produce a solution that satisfies global constraints. For example, if modelers can add ECUs as needed to deploy components to, local guidance can produce a solution that is correct with respect to the constraints, although not necessarily optimal. If, however, ECUs cannot be added to the model and the local strategy guides the modeler to a solution where no ECU has free resources and several components are undeployed, the global constraints cannot be met.

Although a greedy strategy may not produce optimal results for certain types of CSPs, such as bin-packing, in many cases these localized strategies can provide a lower bound on the optimality of the final solution. With bin-packing, a First Fit Decreasing (FFD) (Coffman, 1998) packing strategy that sorts items to be placed into bins by their size and non-deterministically selects the first bin that can hold the item will guarantee that the solution never uses more than ~1.87 times as many bins as the optimal solution. Providing a lower bound on the quality of the solution that a modeler can produce can be extremely important in some domains, such as automotive manufacturing, where you want to minimize risk or cost. Although not guaranteed, a localized strategy may in fact arrive at an optimal or nearly optimal solution. Moreover, local guidance is substantially less computationally complex than providing a global maximum and can be implemented easily with a number of the approaches discussed later in this section.

Batch Processes

Global constraints require the correct completion of numerous modeling steps and are typically not amenable to user intervention. For global strategies, therefore, ***batch***

processes guided by constraint solvers can be used to create multiple relationships to bring the model into a correct state. The key differentiator between local guidance and a batch process is that local guidance deals with modeling transactions involving a single relationship while batch processes operate on modeling transactions containing two or more relationships. The larger the number of relationships in the transaction, generally the more complicated it is to complete.

One possible batch process for the component-to-ECU deployment tool could take each component in the model and create a connection to an ECU in the model to specify a deployment location. Local guidance would produce a single deployment connection for a single component. By increasing the size of the modeling transaction to consider the deployment locations of multiple components, the batch process can use the constraint solver to guarantee that if a possible solution is found, it utilizes only the ECUs currently in the model. By expanding the transaction size that the solver operates on, the batch process allows it to make model modifications that are not locally optimal, but lead to a globally optimal or globally correct solution.

Batch processes help address challenges 1, 2, & 3 of the section *Key Challenges of Complex Domains*. First, a batch process can correctly complete large numbers of modeling actions on behalf of the user, eliminating tedious and error-prone manual modeling (addressing challenge 3). Second, a constraint solver can create both a correct and an optimal solution that can be enacted by a batch process on behalf of the modeler (addressing challenge 1). By tuning the parameters used by the constraint solver, as is discussed in the section *Transforming Non-functional Requirements into Constraint Satisfaction Problems*, the modeler can guarantee both optimality and correctness (addressing challenge 2).

Transforming Non-functional Requirements into Constraint Satisfaction Problems

To integrate local and batch process guidance from a constraint solver, a model and the actions that modelers can perform on the model must be transformed into a series of Constraint Satisfaction Problems (CSPs). This transformation allows the MDD tool to translate the actions of users into queries for a constraint solver. Valid satisfactions of the CSPs correspond to correct ways of completing a modeling action, such as creating a connection.

A CSP is a set of variables and constraints over the values assigned to the variables. For example, $X < Y < 6$ is a CSP with integer variables X and Y. Solving a CSP is finding a set of values (a labeling) for the variables such that the constraints hold true. The labeling X = 3, Y=4, is a correct labeling of X < Y < 6. A constraint solver takes a CSP as input and produces a labeling (if one exists) of the variables. Solvers may also produce labelings that attempt to maximize or minimize variables. For example, X = 4, Y =5, is a labeling that maximizes the value of X.

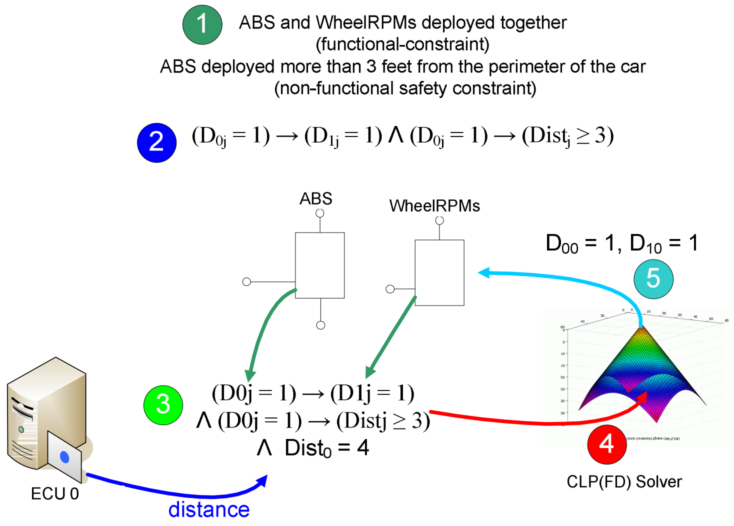
For the deployment example, a deployment of a set of components to a set of ECUs can be viewed as a binary matrix where the cell at row i and column j is 1 if and only if the ith component is deployed to the jth ECU (and 0 otherwise). Each cell can be represented as an independent variable in a CSP. Thus, each variable Dij determines if the ith component is deployed to the jth ECU. Finding a correct labeling of the values for the D variables creates a deployment matrix that can be used to determine where components should be placed.

Assume that the *ABS* (anti-lock braking system) component and the *WheelRPMs* components must be deployed to the same ECU. Also assume that the ABS component must be placed on an ECU at least 3 feet from the perimeter of the car. This series of deployment constraints can be translated in a CSP model. Let the ABS component be the 0th component and the WheelRPMs component be the 1st component. First, the constraint that the ABS component be deployed to the same ECU as the WheelRPMs component is encoded as $(D_{0j} = 1) \rightarrow (D_{1j} = 1)$. Next, for each ECU, a constant $Dist_j$ can be created to store the distance of the jth ECU from the perimeter of the car. Using these constants, the constraint on the placement of the ABS component relative to the perimeter of the car can be encoded as $(D_{0j} = 1) \rightarrow (\mathrm{Dist}_j \geq 3)$. If this CSP is input into a constraint solver, the solver will label the variables and produce a deployment matrix that is guaranteed to be correct with respect to the deployment constraints.

A constraint solver can also be used to derive a solution with a certain degree of optimality. Assume that *N* components need to be deployed to one or more of *M* ECUs using as few ECUs as possible. A new variable *UsedECUs* can be introduced to store the total number of ECUs used by a solution. The constraint UsedECUs = $\sum D_{ij}$ for all i from 0..N and all j from 0..M. The solver can then be asked to produce a labeling of the variables D_{ij} that minimizes the variable UsedECUs. The solver will in turn produce a valid deployment of the components to ECUs that also minimizes the total number of ECUs used.

Constraint solvers typically offer a number of solution optimization options. The options range from maximizing or minimizing a function to using a fast approximation algorithm that guarantees a specific worst-case percentage of optimality. Depending on the constraint solver settings used, a modeler can guarantee the optimality of a model or trade a certain percentage of model optimality for significantly reduced solving time. In contrast, a manual modeling approach provides no way to guarantee correctness, optimality, a percentage of optimality, or a tradeoff between optimality and solution time. For software intensive systems where optimality is important, allowing modelers to tune these parameters is a key advantage of using a constraint solver-integrated modeling approach.

One goal of using a constraint solver is to produce better solutions than a human modeler can create manually and to produce good solutions more reliably. When a solver uses either optimal or approximation algorithms, the solver's solution has a known and guaranteed worst case solution quality. In contrast, there is no guarantee on the solution quality with a manual approach.



1
ABS and WheelRPMs deployed together
(functional-constraint)
ABS deployed more than 3 feet from the perimeter of the car
(non-functional safety constraint)
2
$(D_{0j} = 1) \rightarrow (D_{1j} = 1) \wedge (D_{0j} = 1) \rightarrow (Dist_j \geq 3)$
ABS
WheelRPMs
$D_{00} = 1, D_{10} = 1$
5
3
(D0j = 1) → (D1j = 1)
∧ (D0j = 1) → (Distj ≥ 3)
∧ $Dist_0 = 4$
4
CLP(FD) Solver
ECU 0
distance

Figure 5. Transforming a Model into a Constraint Satisfaction Problem

As shown in Figure 5, the non-functional requirements for the software system must first be collected and documented (step 1). Each non-functional requirement must then be translated into a CSP, such as a system of linear equations (step 2). At this point, the data from the model, such as ECU distances to the car perimeter, are collected and bound to variables in the CSP produced in the previous step (step 3). Next, the CSP with some bound variables (such as resource demands) and some unbound variables (such as the D_{ij} variables in Figure 5) are input into the constraint solver (step 4). The constraint solver then produces bindings for the unbound variables and maps them back to changes in the model (step 5).

A crucial element for creating the right translation from non-functional requirements to a set of CSPs is the abstraction used to decompose the model into the variables and facts (i.e. bound variables) that the CSPs operate on. For example, should ECU and component be present in the formulation of the CSP to represent the bin-packing of the model's resources? The metamodel of a language, as described in *Background* section, provides the terminology and syntactic rules for a modeling language. Since the metamodel contains a precise definition of the relevant types in a modeling language it is ideal for identifying the key concepts that the CSPs should use. The metamodel of a modeling language can be viewed as a set of model entities and the role-based relationships between them. By using this abstraction based on entities and role-based relationships, a model can be conveniently decomposed for processing by a constraint solver. The idea of relationships between elements is the same as the widely used Resource Description Framework's predicate / argument format.

The role-based relationships of an entity represent both its properties (such as available CPU) and its associations (such as hosted components). Each entity can be decomposed into a unique ID and a set of role-based relationships associated with the ID. A requirement, such as "a component is only deployed to an ECU with the correct OS" can be translated into a CSP involving the *Deployment,* and *OS* relationships of a component and ECU. The variables of the CSP for this requirement would be the component and ECU that are being associated through the *Deployment* relationship. The constraint would be that the *OS* relationship of the component and the ECU had the same value (i.e. the same OS).

Associating Modeling Actions with the Constraint Solver

An important integration question is how/when to invoke the constraint solver and what CSPs and variable bindings should be passed to it. The goal is to use the constraint solver to provide local guidance and batch processes to bind the endpoints of relationships in the model. A constraint solver requires a CSP, a set of unbound variables (e.g. unbound endpoints), and a set of bound variables to produce a list of endpoints for relationships. Thus, users' actions and model state must be interpreted to find the correct CSPs, model entities, and unbound endpoints to pass to the solver. By defining the right formal model of the process by which users' actions are interpreted and translated into input data for the constraint solver, the integration process can be more cleanly defined. This section presents a formal abstraction for a user's interaction with a modeling tool and shows the point in the formal specification at which the constraint solver can be integrated and used to automate relationship endpoint binding decisions.

Modeling actions are transactions that take one or more elements of the model and modify the endpoints of the selected elements' role-based relationships. Creating a deployment connection takes a component (the source of the connection) and sets the endpoint of its *TargetECU* relationship. In the *Local Guidance* and *Batch Processes* sections, a modeling action was defined as a transaction by the user that takes a relationship and sets its source and target entities. More formally, a modeling action is a function, *action(X, R, E)*, that takes a model element *X*, a relationship of the element, *R*, and produces an endpoint for that relationship *E,* as shown in Figure 6.

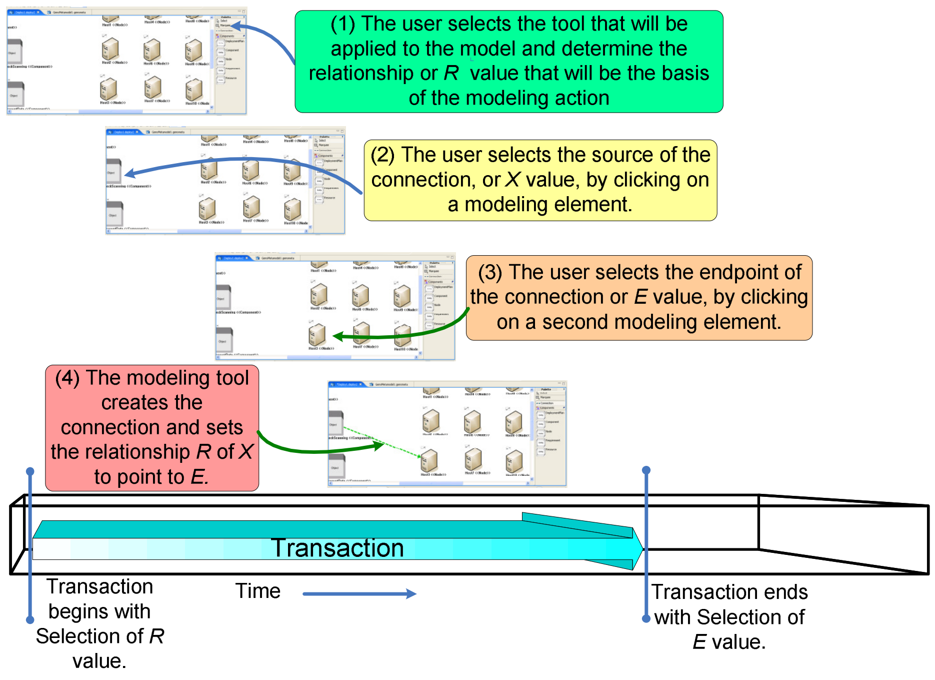

(1) The user selects the tool that will be applied to the model and determine the relationship or R value that will be the basis of the modeling action
(2) The user selects the source of the connection, or X value, by clicking on a modeling element.
(3) The user selects the endpoint of the connection or E value, by clicking on a second modeling element.
(4) The modeling tool creates the connection and sets the relationship R of X to point to E.
Transaction
Transaction begins with Selection of R value.
Time
Transaction ends with Selection of E value.

Figure 6. Diagram of a Modeling Transaction

The goal of a traditional MDD tool is to take the input produced by the user, such as mouse clicks, and translate them into the values for *X, R,* and *E* to update the model. With a traditional MDD tool, the values for *E* are explicitly bound by modelers. A MDD tool integrated with a constraint solver not only provides this traditional explicit binding capability but also provides a constraint solver binding process, in which the constraint solver deduces the proper endpoints for relationships on behalf of the modeler.

The GEF and EMF frameworks can be used to illustrate how X, R, and E are actually implemented in a modeling framework. GEF provides an MVC framework for displaying and editing EMF models. In GEF, each possible user action, such as connecting two elements with a line in the graphical model, is represented with a *Command* object. The command object is a part of the *Command Pattern* (Gamma, 1995), which encapsulates actions that can affect a model in an object. When the user clicks on an element and then presses the delete key, GEF constructs a *DeleteCommand*, sets the command's argument to be the element that was click on, and then calls the command's *execute()* method, which deletes the element from the EMF model. When the user wishes to create a connection, the user selects the connection tool from a tool palette. Selecting the connection tool causes GEF to construct a *ConnectionCommand*. When the user clicks on the first element for the connection, GEF passes the element to the ConnectionCommand as the source argument. When the user clicks on the endpoint for the connection, GEF passes the command the endpoint as the target argument and calls the command's execute() method, which creates the connection between the two elements. Tool implementers create Command objects to specify how each possible user action is translated into changes of the underlying EMF model.

With GEF's command pattern, R is determined by the type of Command object that GEF instantiates. In the deployment example, when the user selects the *DeploymentConnection* tool, GEF creates a corresponding *DeploymentConnectionCommand* object. The Command knows (because it is coded into the command object's execute method) that it is modifying the *TargetECU* relationship of its source argument. The command also knows that its source argument is the X variable in the action(X,R,E) function. Finally, the command knows that its target endpoint represents the E variable. Each Command object is used to translate a graphical user action (*e.g.* adding a connection) into values for X, R, and E. The Command is also responsible for modifying the R relationship between X and E in its execute method. The execute() method of a DeploymentConnectionCommand is shown in the Java code below:

```
public class DeploymentConnectionCommand extends Command{
  ....
  //apply action(X,R,E)
  public void execute() {
   Component source = (Component)this.getSource(); //the X
   ECU target = (ECU)this.getTarget(); //the E

   //the R relationship (targetECU) between X and E is set here
   source.setTargetECU(target);
 }
}
```

In the modified binding process for *E,* each relationship *R* is associated with a CSP specifying what is considered a correct value for *E*. For example, a component could specify that a correct value for its *TargetECU's E* value requires that the chosen *E* value and the component both have the same OS type. When a user input is translated into values for *X* and *R*, a constraint solver integrated MDD tool uses the CSP associated with *R* to automatically derive values for *E* on behalf of the user. In Figure 5, the CSP was found in step 2, the values for *X* and *R* were produced in step 4 and the bindings for *E* were delivered by the constraint solver in step 5. The modified modeling transaction process can be seen in Figure 7.

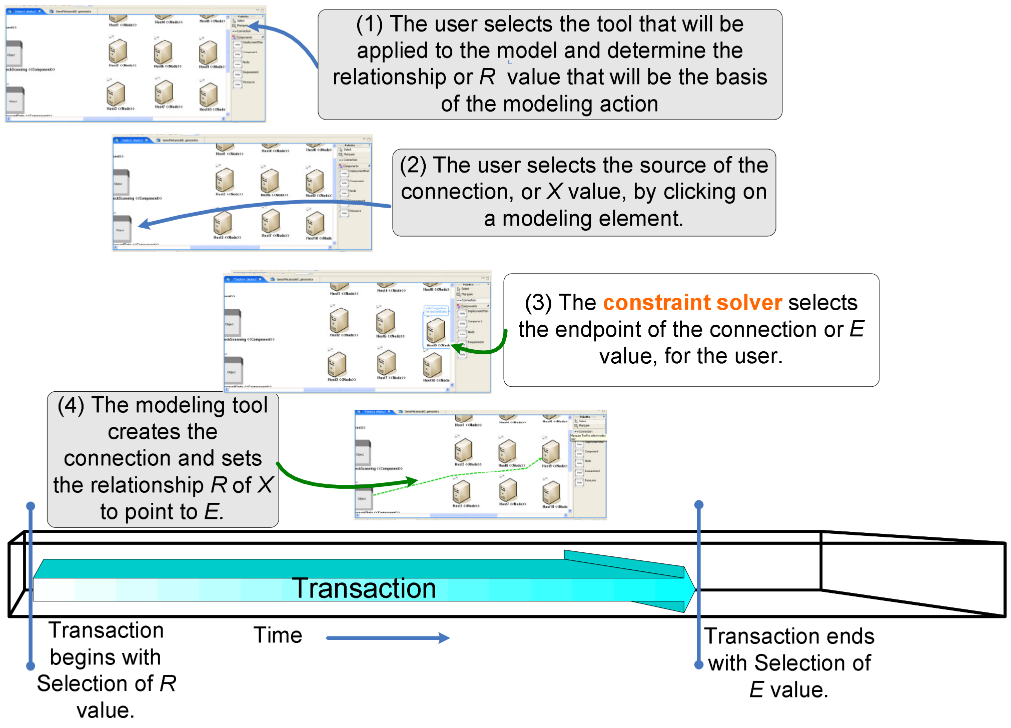

(1) The user selects the tool that will be applied to the model and determine the relationship or *R* value that will be the basis of the modeling action
(2) The user selects the source of the connection, or *X* value, by clicking on a modeling element.
(3) The constraint solver selects the endpoint of the connection or *E* value, for the user.
(4) The modeling tool creates the connection and sets the relationship *R* of *X* to point to *E*.
Transaction
Transaction begins with Selection of *R* value.
Time
Transaction ends with Selection of *E* value.

Figure 7. A Diagram of a Modeling Transaction with a Constraint Solver

In the first step, the user selects a tool or action that will be applied to the model. The tool determines the *R* value or relationship that will be modified by the user's actions. In the second step, the user clicks on a modeling element to initiate a connection and hence modify a relationship in the underlying model. The element that the user clicks on becomes the *X* value that will be passed to the constraint solver. In the third step, the modeling environment looks up the correct CSP that must be satisfied by the endpoints of the relationship specified by the *R* value. The modeling environment then passes this CSP, the *X,* and *R* values to the solver. The solver finds the endpoints that satisfy the CSP and returns these endpoints as possible *E* values. Finally, the *E* values are presented graphically to the user.

The GEF *DeploymentConnectionCommand* can be modified to incorporate this new process by which the constraint solver chooses the value for *E.* The Command creation and initial argument setting remains unchanged. However, after the source of the connection has been set, the constraint solver can be invoked to solve for a value for E. If a value is returned, the execute() method can be called immediately. The new *DeploymentConnectionCommand* is:

```
public class DeploymentConnectionCommand extends Command{
  ....
  public void setSource(Object obj) {
     this.source = obj;

     //the X
     Component source = (Component)obj;

     //call the solver to find valid values for E
     List endpoints = this.solver.findEndpoints(source.getId(),
                                                 “targetECU”);

     //if there is only one possible value, go ahead and execute
     if(endpoints.size() == 1){
        setTarget(endpoints.get(0));
        execute();
     }
     else if(endpoints.size() > 0) {
        //otherwise, show the user valid E values by
        //modifying their background color
        for(Object obj : endpoints)
           ((ECU)obj).setBackgroundColor(Color.yellow);
     }
     else {
        //notify the user that there are no
        //possible deployment locations for the Component
        source.setBackgroundColor(Color.red);
     }
  }

  //apply action(X,R,E)
  public void execute() {
   Component source = (Component)this.getSource(); //the X
   ECU target = (ECU)this.getTarget(); //the E

   //the R relationship (targetECU) between X and E is set here
   source.setTargetECU(target);
 }
}
```

In the modified DeploymentConnectionCommand, immediately after GEF sets the source of the connection, the command invokes the constraint solver to find valid endpoints. If exactly one endpoint is found, the setTarget method is called with that endpoint and the Command is executed. If more than one valid endpoint is found, each valid target has its background color changed to yellow (a visual queue). If there is no possible deployment location for the Component, its background color is changed to red.

In a traditional process, the user would be required to click first on the source element, decide on a valid deployment location for the source, and then click on the deployment location. With the modified Command object, the object itself attempts to determine the valid targets (E) using the constraint solver. The Command can then either automatically complete the action on the user's behalf, if there is exactly one possible endpoint. If there is more than one possible endpoint, the Command can highlight those endpoints for the user. If no endpoints are found, the Command can notify the user by changing the Component's background color to red.

In many situations, the user will wish to find a valid endpoint for a specified R relationship for every member of a set of modeling elements. For example, the user may wish to select some or all of the Components and have the solver find a valid target ECU for every Component such that no global deployment constraint, such as resource consumption, is violated. Using the GEF framework, a new *BatchDeploymentCommand* can be created.

Just as with other GEF commands, the BatchDeploymentCommand can have a tool palette entry associated with it that the user can select. When the user selects the corresponding tool entry, the BatchDeploymentCommand is created. The batch command takes a group of modeling elements, which the user specifies through a group selection, and creates a connection for each member of the group to a valid ECU. The Java code for the BatchDeploymentCommand is:

```
public class BatchDeploymentCommand extends Command{
  ....

  public void execute() {
   //the set of Xs
   Component[] sources = (Component[])this.getSources();

   //the solver deduces an E for each X
   Object[] targets = this.solver.findValidTargets(sources,
                                                   "targetECU");

   if(targets != null){
     for(int i = 0; i < targets.length; i++) {
       sources[i].setTargetECU((ECU)targets[i]);
     }
   }
 }
}
```

Constraint Solver and MDD Tool Integration Frameworks

There are a large number of optimization, constraint solver, and inference engines available for use with local guidance and batch processes. As noted in (Van Hentenryck, 1996), however, automating the formulation of real problems in a suitable form for efficient algorithmic processing is hard. Transforming an arbitrary graphical model, a modeling action, and a set of modeling constraints into a CSP for a constraint solver is tedious and error-prone. Integrating the results of the solver back into a MDD tool and providing interactive capabilities is also hard. Each of the five steps from the section *Transforming Non-functional Requirements into Constraint Satisfaction Problems* may require substantial effort. By choosing the right approach and architecture, however, the difficulty of leveraging a constraint solver in a modeling environment can be reduced substantially.

The following are five important properties of an architecture for integrating a constraint solver with a MDD tool:

1. **Solver frameworks must respect domain-specific concepts from the MDD tool** and provide a flexible mechanism for translating non-functional requirements into CSPs using domain notations. MDD tool users should be able to specify constraints in a language or notation that mirrors the domain rather than a system of linear equations and makes mapping requirements to a CSP easier.
2. **The local guidance and batch processes should lead modelers towards solutions that are considered optimal** or good based on quality metrics from the domain. Whenever possible, solvers should be used to iterate through multiple valid solutions and suggest only those considered most optimal. Modelers should be able to plug-in custom formulas for measuring optimality in the target domain and the tool should be able to present multiple suggestions based on various types of optimization.
3. **The constraint solver integration should automate tedious and complex modeling tasks**, such as solving for and assigning values for global constraints, performing repetitive localized decisions, or providing feedback to modelers to suggest valid modeling decisions.
4. **The solver framework must accommodate long-running analyses** for problem instances that cannot be solved on-line. For large optimization problems, such as finding the lowest cost assignment of components to ECUs, the constraint solver may need several hours or days to find a solution. In cost-critical situations, such as manufacturing, allowing the solver the extra time to find the best solution can be critical.

With a constraint-solver integrated modeling environment, a user goes through an iterative process of specifying portions of a model, adding or refining non-functional requirements as constraints, and using the constraint solver to automate model construction and optimization. Figure 9 illustrates the modeling processing with an integrated constraints solver.

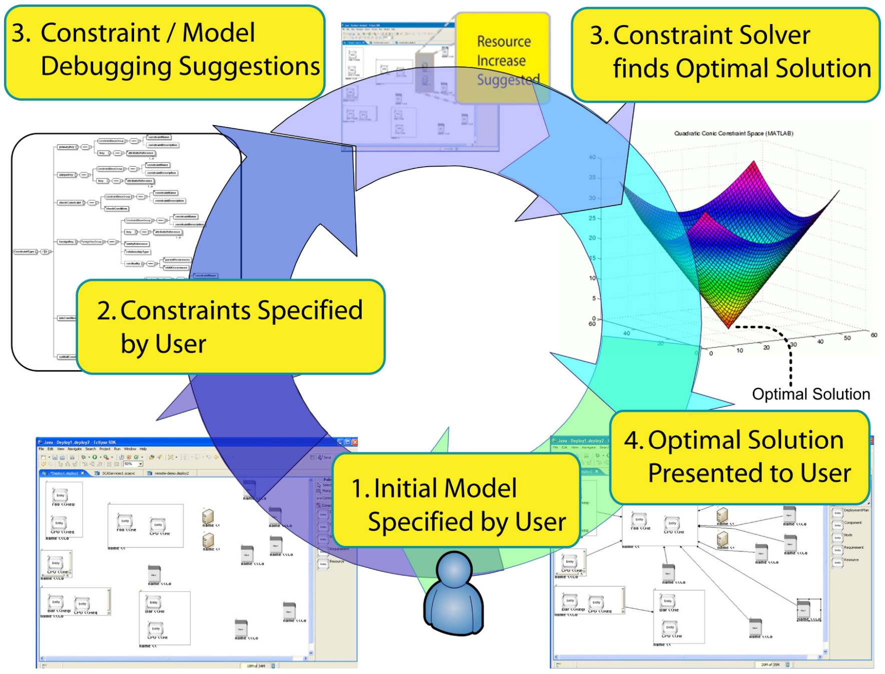

3. Constraint / Model Debugging Suggestions
Resource Increase Suggested
3. Constraint Solver finds Optimal Solution
Quadratic Conic Constraint Space (MATLAB)
2. Constraints Specified by User
Optimal Solution
4. Optimal Solution Presented to User
1. Initial Model Specified by User

Figure 9. A Modeling Cycle with Constraint Solver Integration

Sidebar 1: Prolog

Prolog is a logic programming language that allows developers to specify a set of facts or Knowledge Base and then create rules specifying logical assertions or constraints on the facts. Prolog rules take one or more input variables, denoted by variable names with capital letters, and specify a series of logical assertions on these variables, other facts, or rules in the KB. When a Prolog rule is invoked with only bound variables, meaning all variables have values assigned to them, Prolog returns whether or not the logical assertions contained within the rule hold true. An important capability of Prolog is that if a rule is invoked with some unbound variables, Prolog will attempt to find bindings of those variables from the facts in the KB that satisfy the logical assertions in the rule. When constraints are implemented as Prolog rules, Prolog can deduce valid bindings for the variables that the constraints restrict.

In the first step, a user specifies the initial model entities in the solution. In the second step, the user adds constraints for the requirements of the solution into the MDD tool. During the third phase, the user invokes the constraint solver, using local-guidance or a batch processes, to find valid endpoints for various relationships in the model. Finally, in the fourth step, the valid endpoints found by the constraint solver are shown to the modeler using visual queues, such as highlighting valid entities.

A Prolog-based Approach to Constraint Solver Integration

Choosing a constraint solver is one of the driving forces in the process of transforming a set of non-functional requirements into a CSP. Each solver will generally have a unique representation of the problem in its native format. The choice of solver therefore affects how the transformation from non-functional requirements to a concrete representation of a CSP is performed. Many types of solvers are available and implemented in a number of languages. The remainder of this section presents an approach we have developed, called Role-based Object Constraints (White, 2006), to providing local guidance and batch processes based on Prolog (Bratko, 1986).

Using ROCs, we have implemented constraint-solver integrated modeling tools for automated product line variant selection (White, 2007), component to ECU deployment in automobiles (White, 2006), and aspect weaving (Nechypurenko, 2007). Our implementation of ROCs is integrated with the Generic Eclipse Modeling System (GEMS) (White, 2005), a part of the Eclipse Generative Modeling Tools (GMT) project. A screenshot of a batch process executing in our GEMS- based deployment modeling tool is shown in Figure 10.

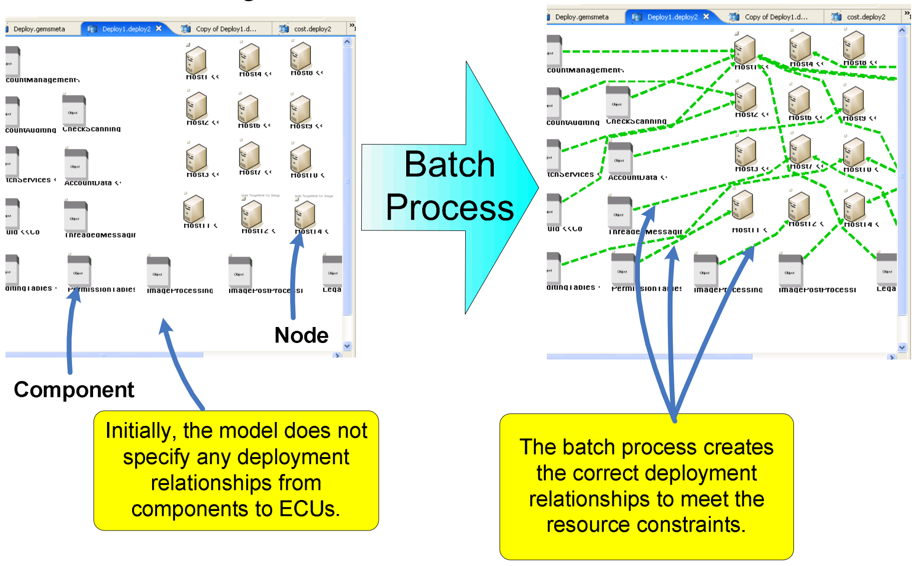

Batch Process
Node
Component
Initially, the model does not specify any deployment relationships from components to ECUs.
The batch process creates the correct deployment relationships to meet the resource constraints.

Figure 10. Execution of a ROCs Batch Process Shown in GEMS

Prolog is a declarative programming language that allows programmers to define a Knowledge Base (KB) (also known as a fact set) and a group of rules that implement a set of CSPs (see Sidebar 1). Prolog can then evaluate these rules and determine if they can be satisfied by the known facts. Prolog uses a predicate syntax, where rules can be defined as predicates that resolve to the satisfaction of a conjunction of other predicates. Rules are akin to methods that check if a constraint over a set of variables holds true.

Predicate rules can be used to check constraints, by invoking the rule with all variables bound, in which case Prolog replies with whether or not the rule or CSP evaluates to true. If variables are left unbound when the rule is invoked, however, Prolog uses backward chaining to produce bindings from the KB of the unbound variables that will satisfy the CSP. Prolog therefore provides a key degree of flexibility since it can be used both to check constraints (similar to OCL described in the section *Constraint Checking with OCL*) or to derive solutions to the CSPs.

The remainder of this Section presents an approach to integrating Prolog into a modeling tool. Prolog was chosen since it has a readable textual syntax as opposed to the linear-equation based syntax of other possible solvers. Using Prolog, however, does trade some speed for readability and ease of use. Prolog also is a widely used and supported programming language for constraint solving and numerous existing solvers and libraries are available in Prolog.

Transforming Models into Prolog Knowledge Bases

Integrating Prolog as a constraint solving engine involves capturing the state of the model and translating it into a Prolog KB, as seen in Figure 11. For the deployment of components to ECUs example from the *Introduction* section, the components, ECUs, and their resources must be translated into predicate facts in Prolog. Generally, predicates are created that relate a unique key or ID of each model element to various properties that the model element possesses. This concept is similar to the use of pointers and allows the flattening of an object-oriented model into a predicate KB.

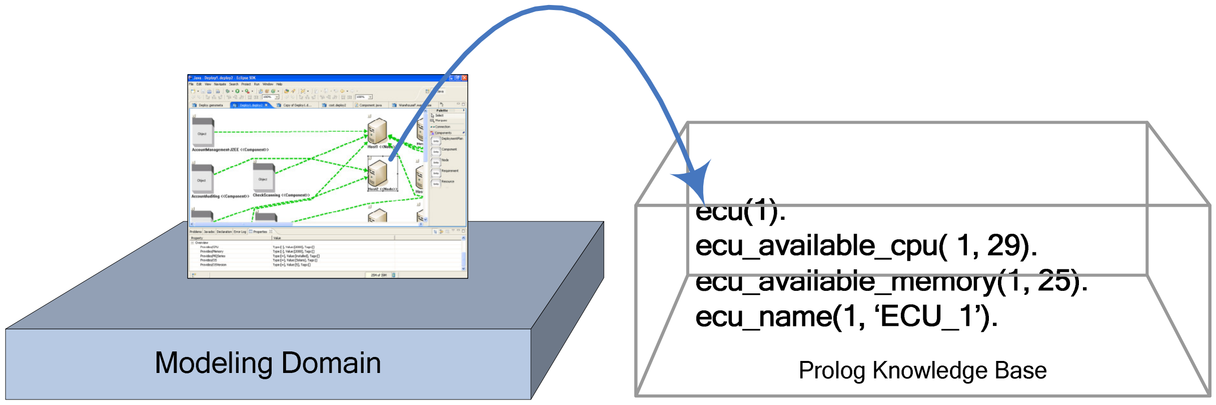


Figure 11. Transforming Model Elements into Prolog Facts

Developers must select the format of the predicates used to translate the model into a Prolog KB. One approach is to use a consistent set of Prolog predicates across modeling languages and customize them by adding domain-specific information into the variables the predicates operate on. For example:

```
self_type(1, ecu).
self_attribute(1, available_cpu, 29).
self_attribute(1, available_memory, 25).
self_attribute(1, name, 'ECU_1').
```

describes a set of Prolog facts that provide a general predicate format applicable to a range of model types. This set of facts asserts that the element with ID "1" is of type "ecu." The facts also assert that the element has three attributes: *available_cpu*, *available_memory*, and *name*, with values: "29", "25", and "ECU_1," respectively. Different modeling languages can be accommodated by changing the second argument of the predicates, the attribute name, that is being defined. The tradeoff of using a general format, however, is it violates the first design criterion described in the section *Constraint Solver and MDD Tool Integration Frameworks*, i.e., offering a domain-specific interface. The predicates do not vary across domains, which makes it harder for a domain expert to understand how they relate the concepts from his or her domain. Rather than using terminology specific to the deployment of components to ECUs (e.g. ecu, available_memory, etc.), the predicates are based on describing the attributes and types.

A more domain-specific approach is to create custom predicates for each modeling language to mirror the notation from the domain. For example, the same set of facts can be rewritten as:

```
ecu(1).
ecu_available_cpu( 1, 29).
ecu_available_memory(1, 25).
ecu_name(1, 'ECU_1').
```

which provides a more domain-specific interface. The main drawback of this format, however, is that introspection is not possible, i.e., rules cannot query for all of the properties of an arbitrary element. When translating non-functional requirements into Prolog rules, domain-specific predicates are generally more advantageous since they allow the production of more compact and readable rules. Introspection is also typically not needed for writing CSPs in Prolog.

To identify the domain-specific predicates to use for the KB, the metamodel for a modeling language can be viewed as a set of model entities and the role-based relationships between them. For each entity, a unique id and a predicate statement specifying the type associated with the entity. For example, each ECU in the model is transformed into the predicate statement ecu(id), where id is the unique id for the ECU. For each instance of a role-based relationship in the model, a predicate statement is generated that takes the id of the first participating entity and the id of the entity to which the first entity is being related.

For example, if a component, with id 23, has a *TargetECU* relationship with an ECU, with id 25, the predicate statement targetECU(23,25) is generated. This predicate statement specifies that the entity with id 25 is a *TargetECU* of the entity with id 23. Each KB provides a domain-specific set of predicate statements. As a model is manipulated in its graphical editor, the Prolog KB is updated using assert/1 and retract/1 statements, which add and remove facts from the Prolog KB, respectively.

Mapping Non-functional Requirements to Prolog Rules

Using a domain-specific knowledge base, modelers can specify non-functional requirements in the form of Prolog rules for each type of metamodel relationship. These constraints semantically enrich the model to indicate the non-functional requirements of a correct model. They are used by constraint solvers to deduce solutions to local guidance and batch process problems. For example, consider the following constraint to check whether an ECU is a valid ECU of a component:

```
is_a_valid_component_targetECU(Component, ECU) :-
        component_requiredOS(Component, OS),
        ecu_providedOS(ECU, OS).
```

This constraint, which checks to ensure that the OS required by the component matches the OS provided by the ECU, can be used to check a component-ECU combination, i.e.:

is_a_valid_component_targetECU(component_23, ecu_25).

by assigning the *Component* variable the value “component_23” and the *ECU* variable the value “ecu_25.” The rule can also be used to find valid ECUs that can play the *TargetECU* role for a particular component using Prolog's ability to deduce the correct solution:

```
is_a_valid_component_targetECU(component_23, ECU).
```

by leaving the *ECU* variable unbound (unbound variables are begin with capital letters). In this example, the *ECU* variable will be be bound to the ID’s of the ECUs in the KB that have the same OS as the component. This example shows how the non-functional requirement rules can be used both to check and to deduce solutions.

The role-based relationships present in the metamodel not only produce domain-specific predicates but also serve as the glue between graphical modeling actions, such as creating connections, and the constraint solver. The non-functional requirement rules that developers create can be associated with role-based relationships in the metamodel as seen in Figure 12.

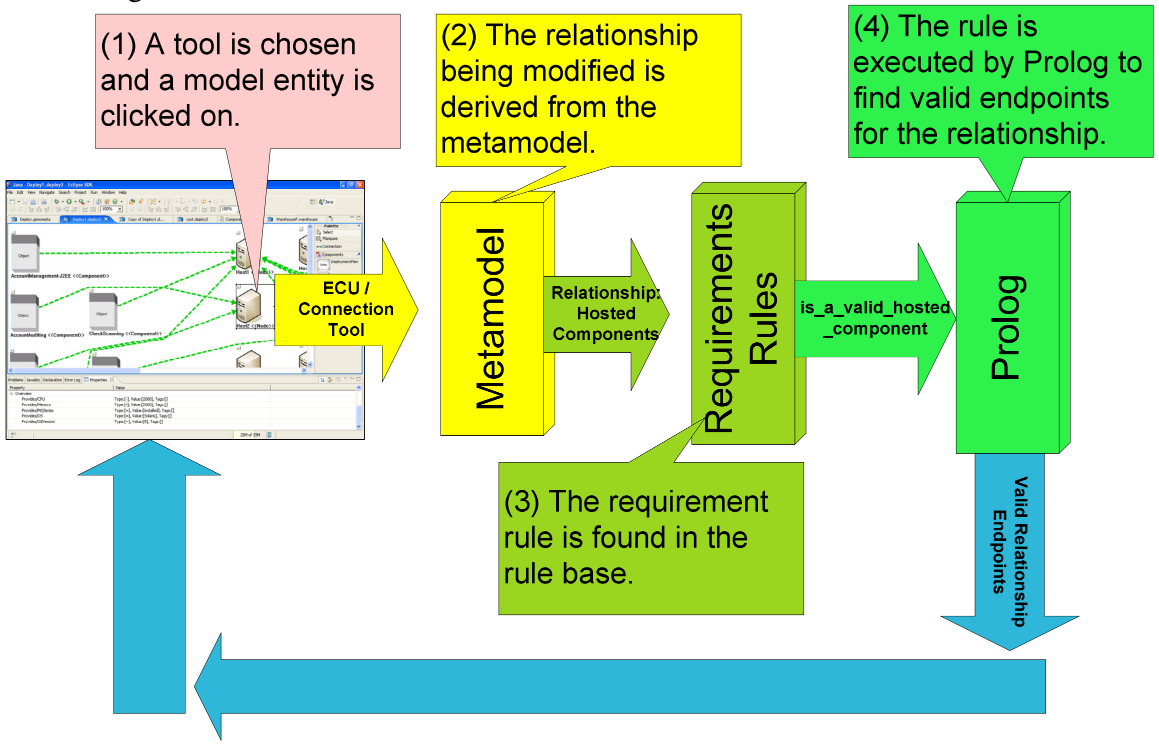

(1) A tool is chosen and a model entity is clicked on.
(2) The relationship being modified is derived from the metamodel.
(4) The rule is executed by Prolog to find valid endpoints for the relationship.
ECU / Connection Tool
Metamodel
Relationship: Hosted Components
Requirements Rules
is_a_valid_hosted_component
Prolog
Valid Relationship Endpoints
(3) The requirement rule is found in the rule base.

Figure 12, Invoking Requirement Rules to Find Relationship Endpoints

When a model element is clicked on to initiate a change (step 1), the metamodel is consulted to determine the role-based relationship (step 2) affected by the change. The corresponding non-functional requirement rules can then be obtained (step 3) and executed by Prolog (step 4) to check the validity of the change.

To provide local guidance, the non-functional requirement rules associated with metamodel relationships can be executed with unbound variables to deduce endpoints for the relationship. For example, if a user begins creating a deployment connection originating from a component, the MDD tool can deduce that the deployment connection will set the *TargetECU* relationship of the component and execute the is_a_valid_component_targetECU rule with only the originating component bound as an argument. Prolog will then return bindings for the ECUs variable of the rule that are valid deployment targets for the component.

In a graphical modeling environment that does not include a constraint solver, graphical actions, such as mouse clicking and movement, are translated into changes in the underlying object graph of the model. By adding a constraint solver, graphical actions are translated into proposed model modification transactions and then the proposed transactions are turned into CSPs and solved. Users can therefore modify the model directly as in a traditional approach and use graphical actions to initiate constraint solvers that modify the model on their behalf.

In the connection creation example from the *Local Guidance* section, a modeler's mouse-click on the source component is translated into the initiation of a connection from the component. This connection initiation proposal is then used to query the metamodel to determine the relationship that is being modified on the component. Next, the CSPs or non-functional requirement rules that are bound to the relationship are obtained and finally they are executed in the constraint solver to find valid bindings for the unbound variables. The bindings produced represent the valid completions of the modeling transaction. These valid bindings can be returned to the user as graphical proposals, such as highlighting model elements, or committed as changes to the underlying model.

CONCLUDING REMARKS

For large-scale DRE systems, traditional modeling approaches allow developers to raise the level of abstraction used for solution specification and illuminate design flaws earlier in the development cycle. Many DRE systems, however, have exponential design constraints, extremely large model sizes, or other complexities that make it hard to handcraft a model of a solution. For these types of challenging domains, automated design guidance based on the design constraints is needed.

A constraint solver can be integrated into a modeling environment to provide design guidance for complex domains. As shown in section *Modeling Guidance*, using the concepts of *local guidance* and *batch processes* a constraint solver can help modelers perform both single and multi-step modeling activities.

The lessons learned from our ROCs approach, described in the section *Modeling Guidance*, to integrating a Prolog constraint solver with the GEMs modeling environment are:

- **Constraints can be used for reasoning by a constraint solver**. A constraint solver improves solution quality not by checking manually produced solutions, but by actively guiding a user towards a correct solution. The solver helps ensure that users do not produce an incorrect solution, rather than just notifying them if their solutions are invalid.
- **User actions can be abstractly modeled as functions**. User interactions with MDD tools can be viewed as a function that takes a set of modeling elements and maps the endpoints of a specific relationship of the elements to an endpoint explicitly provided by users. A constraint solver can be integrated into a modeling environment by dynamically choosing the endpoints for the relationships with the constraint solver rather than requiring endpoints to be explicitly enumerated by the modeler. Local

guidance and batch processes can be used to produce the endpoints for relationships, as described in the section *Local Guidance*.
- **Constraint solvers should be reused**. Writing a constraint solver is hard. Developers of model-driven processes should therefore focus on integrating existing constraint solvers or constraint solving languages. Prolog is a good choice for a general purpose constraint solving language that can be integrated, as shown in the section *A Prolog-based Approach to Constraint Solver Integration*.
- **Debugging constraint conflicts or over constrained systems is hard.** When no valid solution to a CSP can be found, deriving why a solution can't be found can be complex. With global constraints, the cause of the failure can be the overall organization of the solution and thus it is difficult to provide a meaningful explanation to the modeler.
- **Constraint solvers typically perform well in practice.** Although many optimization and constraint satisfaction problems are combinatorialy complex, constraint solvers typically can solve them in a reasonable time frame. Constraint solvers can use approximation algorithms to quickly produce solutions that are good but not optimal.
- **The complexity of a constraint satisfaction problem is dependent on each problem instance.** Certain instances of a type of constraint satisfaction problem will be easier to solve than others. Predicting which instances are challenging is hard. Although it is hard to predict which instances are challenging, constraint solvers often work well on all instances in domains that humans manually produce solutions for.

As MDD tools continue to develop and capture more useful design decisions for larger and more complex applications, constraint solving and other design automation techniques will become more important. Design automation should not only improve design quality but should also help to allow model-driven processes to scale to handle next-generation models with significantly more complexity.

The tools and code presented in this chapter are a part of the Generic Eclipse Modeling System (GEMS). GEMS is an opensource project available from http://www.sf.net/projects/gems.

FUTURE RESEARCH DIRECTIONS

This section describes the emerging trends in the development of software intensive systems, how these trends will affect software development methodologies, and what future research will be needed to address future development problems. The future trends are presented in the context of MDD. Particular emphasis is placed on how these trends will impact the use of constraint solvers in software development.

Capturing Design Rules

Model-driven technologies are raising the level of abstraction for software development by enabling developers to express higher-level, more domain-specific intentions in the solution specifications they produce. These intentions have traditionally been captured through documentation, such as text files or MS Word documents. With MDD technologies, developers can formally specify the design goals and rules that traditionally could only be expressed in documentation, in the solution specification.

In the past, conventional tools could not document the rules that led a developer to make design decisions in a rigorous manner that could be used for automated design assistance. For example, implementation-based software development methodologies, such as coding a solution in C++, could not capture communication rate information, memory consumption, minimum distance from the car's perimeter, or other constraint information in a form that could drive application organization. With an MDD approach, however, a designer can specify that a connection needs to provide a guaranteed messaging rate rather than the type of connection that should be used, i.e., designers can specify *why* one connection type should be preferred over another connection type.

Various MDD tools are developing that can utilize this design information. The Component Utilization and Test Suite (CUTS) (Hill, 2007) is an MDD tool that allows developers to empirically evaluate system designs before they are implemented. Other tools, such as J2EEML (White, 2007), provide the ability to perform analysis on adaptive applications and anticipate conflicting design goals. Finally, feature modeling tools (Antkiewicz, 2007) provide the capability to capture software component commonality and variability requirements and enforce them during system design.

Utilizing Design Information to Provide Automated Design Assistance

Automation can be applied to help guide designers to better solutions by formally capturing the goals of the application and/or *why* designers chose particular design decisions. MDD tools, and specifically domain-specific modeling languages, have allowed developers to tailor the solution specification to include information pertinent to their domain. As MDD technologies increase the breadth of information that can be distilled from designers into a solution specification, a wealth of new design guidelines or constraints will become available for constraint solving.

An emerging area of research therefore involves the integration of automated reasoning systems with MDD tools. In particular, the reuse of existing constraint solver, decision assistance, and other guidance mechanisms across will be an important software development goal. These solver and decision assistance mechanisms are costly to produce and thus will benefit from greater portability between MDD tool infrastructures, such as the Eclipse Modeling Framework, Microsoft DSL tools (Microsoft, 2007), and the Generic Modeling Environment.

One challenge of leveraging constraint solvers in a modeling environment is mapping the domain requirements to CSPs that can be used for automated solving. These mappings are often complex and tightly coupled to individual solver and metamodel formats, despite the fact that requirement types, such as resource requirements, occur across multiple domains. Current solutions for transforming requirements into CSPs tightly couple the translation to a specific solver or metamodel and require costly reinvention and rediscovery of existing mappings. Additional work is needed ensure that the complex mappings from requirements to CSPs can be templatized and reused across applications.

Model transformation techniques are developing rapidly and may provide a mechanism for future decoupling of CSPs from solvers and provide portability through translation. The Atlas Transformation Language (ATL) (Jouault, 2005) provides powerful transformation capabilities as well as compilation to bytecode. Other emerging approaches include Open Architecture Ware (oAW) (Open Architecture Ware, 2007).

Finally, templatization approaches are also viable, such as those proposed by Willans (Willans, 2002).

Constraint Solver Guided Software Reuse

Significant advances in the area of software reuse will drive the need for integrating constraint solvers and MDD tools. These advances are evident in the increased use of commercial-off-the-shelf (COTS) components (Schmidt, 2002) rather than customized proprietary solutions. In particular, using COTS components in the DRE systems domain requires constraint solvers since applications in this domain often have exponential constraints that must be met by selecting and assembling COTS components together with proprietary components into a final composite application.

The selection of components is an example of a CSP. Developers must take the requirements of an application in a DRE system, the capabilities of the COTS and proprietary components, and find a set of compatible components, possibly from numerous vendors, that will satisfy both the functional and complex non-functional constraints. Moreover, components may have complex configuration needs, such as setting messaging policies, to enable them to function properly. Solving these challenging CSPs will require the use of constraint solvers to select components based on high-level design criteria captured in models.

COTS components will require further development and standardization in how metadata, such as messaging periodicity, is captured and disseminated to tools. Standardization will allow for greater interoperability between tools. A standard metadata format will also provide component developers with a consistent methodology for documenting component requirements, dependencies, and capabilities.

Services have seen the most standardization in metadata descriptions. The Resource Description Framework (RDF) (Lassila, 1999) and the Web Services Description Language (WSDL) are emerging as promising standards for describing services. Other approaches, such as those presented by O'Sullivan et. al (O'Sullivan, 2002), focus on capturing the non-functional aspects of services. Formal methods for describing components are also emerging, such as those proposed by Poizat et. al (Poizat, 2004).

REFERENCES

Antkiewicz, M., & Czarnecki, K. (2006, October). Framework-Specific Modeling Languages with Round-Trip Engineering. In *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Genoa, Italy.

Atkinson, C. & Kuhne, T. (2003). Model-driven development: a Metamodeling Foundation. *IEEE Software*, 20(5), 36-41.

Bratko, I. (1986). *Prolog Programming for Artificial Intelligence*. Reading, Massachusetts: Addison-Wesley.

Coffman, E., Galambos, G., Martello, S., & Vigo, D. (1998). Bin Packing Approximation Algorithms: Combinatorial Analysis. *Handbook of Combinatorial Optimization*. Norwell, Massachusetts: Kluwer Academic Publishers.

Cormen, T., H. Rivest, R., L. Leiserson, C., E., & Stein, C. (1990). *Introduction to algorithms*. Cambridge, Massachusetts: MIT Press.

Fagan, M. (1999). Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, *38(2/3)*, 258-287

Forman, G., & Zahorjan, J. (1994). The Challenges of Mobile Computing. *IEEE Computer*, *27(4)*, 38-47

Fowler, M. & Scott, K. (2000). *UML Distilled*. Reading, Massachusetts: Addison Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Massachusetts: Addison-Wesley.

Georg, G., France, R., & Ray, I. (2003). Composing Aspect Models. In *4th AOSD Modeling With UML Workshop.* Boston, Massachusetts.

Ginsberg, M. (1989). A Circumscriptive Theorem Prover. *Artificial Intelligence*, *32(2)*, 209-230.

Graphical Modeling Framework. (2007). http://www.eclipse.org/gmf.

Hill, J. H. & Gokhale, A. (2007 to appear). Model-driven engineering for development-time QoS validation of component-based software systems. In *Proceeding of International Conference on Engineering of Component Based Systems*, Tuscon, AZ.

Jaaksi, A., (2002). Developing Mobile Browsers in a Product Line. *IEEE Software, 19(4)*, 73-80.

Jouault, F., Kurtev, I. (2005, October). In *Model Transformations in Practice Workshop at MoDELS,* Montego Bay, Jamaica

Kent, S. (2002, May). Model Driven Engineering. In *Integrated Formal Methods: Third International Conference*, Turku, Finland.

Kleppe, A., Bast, W., & Warmer. B. (2003). *The Model Driven Architecture: Practice and Promise*. New York, New York: Addison-Wesley Professional.

Lassila, O. & Swick, R. (1999). *Resource Description Framework (RDF) Model and Syntax*. World Wide Web Consortium.

Ledeczi, A., Bakay, A., Maroti M., Volgysei P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). Composing Domain-Specific Design Environments. *IEEE Computer*, *34(11)*, 44-51.

Ledeczi, A. (2001). The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing*. Budapest, Hungary.

Lodderstedt, T., Basin, D., & Doser, J. (2002). SecureUML: A UML-Based Modeling Language for Model-Driven Security. *UML*, *2460*, 426-441.

Martin, G., Lavagno, L., & Louis-Guerin, J. (2001). Embedded UML: a Merger of Real-time UML and Co-design. In *9th International Symposium on Hardware/Software Codesign*, Copenhagen, Denmark

Microsoft Domain-Specific Language Tools (2007). http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx

Moore, B. (2004). *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework.* Boca Raton, Florida: IBM, International Technical Support Organization.

Nechypurenko, A., Wuchner, E., White, J., & Schmidt, D.C. (2007). Application of Aspect-based Modeling and Weaving for Complexity Reduction in the Development of Automotive Distributed Realtime Embedded Systems, In

Proceedings of the Sixth International Conference on Aspect-Oriented Software Development. Vancouver, British Columbia.

Nelder, J., & Mead, R. (1965). A Simplex Method for Function Minimization. *Computer Journal, 7(4),* 308-313.

Object Management Group. (2007). *Meta Object Facility (MOF), Version 1.4*, Retreived January, 2007, from http://www.omg.org/docs/formal/02-04-03.pdf.

Open Architecture Ware (2007). www.openarchitectureware.org

O'Sullivan, J., Edmond D., & Ter Hofstede, A., (2002). What's in a Service? Towards Accurate Description of Non-functional Service Properties. *Distributed and Parallel Databases, 12(2)*, 117-133

Poizat, P., Royer, J., & Salaun, G. (2004, June). Formal Methods for Component Description, Coordination and Adaptation. In *1st International Workshop on Coordination and Adaptation Techniques for Software Entities*, Oslo, Norway

Quatrani, T. (2003). *Visual Modeling with Rational Rose and UML.* Reading, Massachusetts: Addison Wesley.

Schmidt, D., C. (2002). Middleware for Real-time and Embedded Systems. *Communications of the ACM, 45(6)*, 43-48.

Selic, B. (2003). The Pragmatics of Model-Driven Development. *IEEE Software, 20(5)*, 19-25.

Sztipanovits, J., & Karsai, G. (1997). Model-integrated Computing. *IEEE Computer, 30(4)*, 110-111.

Van Hentenryck, P., & Saraswat, V. (1996). Strategic Directions in Constraint Programming. *ACM Computing Surveys*, *28(4)*, 701-726.

Warmer, J., & Kleppe, A. (1998). *The Object Constraint Language: Precise Modeling with UML.* Boston, Massachusetts: Addison-Wesley Longman Publishing.

Weber, M., & Weisbrod, J., (2002). Requirements Engineering in Automotive Development-experiences and Challenges. In *IEEE Joint International Conference on Requirements Engineering,* Essen, Germany.

White, J. Schmidt, D.C., Wuchner, E., & Nechypurenko, A. (2007). Automating Product-Line Variant Selection for Mobile Devices. In *Proceedings of the 11th Annual Software Product Line Conference (SPLC)*, Kyoto, Japan.

White, J., Gokhale, A. & Schmidt, D.C. (2007). Simplifying autonomic enterprise Java Bean applications via model-driven development: A case study. *Journal of Software and System Modeling*.

White, J., Nechypurenko, A., Wuchner, E., & Schmidt, D.C. (2006). Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains. In *Proceedings of the Workshop on Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems*. Portland, Oregon.

White, J. (2005). The Generic Eclipse Modeling System. http://www.sf.net/projects/gems.

Willans, J.S., Sammut, P., Maskeri, G., & Evans, A. (2002). *The Precise UML Group*

Yuan, W., & Nahrstedt, K. (2003). Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems. In *19th ACM Symposium on Operating Systems Principles*, Bolton Landing, New York.

ADDITIONAL READING

Bast, W., Kleppe, A.G., & Warmer, J.B. (2003). MDA Explained: *The Model Driven Architecture: Practice and Promise*. Boston, Massachusetts: Addison-Wesley.

Beckert, B., Keller, U., & Schmitt, P.H. (2002). Translating the Object Constraint Language into First-order Predicate Logic. In *Proceedings of VERIFY*, Copenhagen, Denmark.

Bézivin, J. (2005). On the Unification Power of Models. In *Software and Systems Modeling*, 4(2), 171-188.

Bézivin, J. (2004). In Search of a Basic Principle for Model-driven Engineering. In *Novatica,* 5(2).

Bézivin, J., Farcet, N., Jezequel, J.M., Langolis, B., & Pollet, D. (2003). Reflective Model-driven Engineering. In *Proceedings of 6th International Conference on the Unified Modeling Languages and Applications*. San Francisco, California

Bézivin, J. (2001). From Object Composition to Model Transformation with the MDA. In *Proceedings of the 39th International Conference on the Technology of Object-Oriented Languages and Systems.* Santa Barbara, California.

Budinsky, F. (2003). *Eclipse Modeling Framework*. Boston, Massachusetts: Addison-Wesley.

Clocksin, W.F., & Mellish, C.S. (1984). *Programming in Prolog*. New York, New York: Springer-Verlag

Czarnecki, K. & Eisenecker, U.W. (2000). *Generative Programming: Methods, Tools, and Applications*. New York, New York: ACM Press/Addison-Wesley Publishing Co.

Coplien, J.O., & Schmidt, D.C. (1995). *Pattern Languages of Program Design*. New York, New York: ACM Press/Addison-Wesley Publishing Co.

Frankel, D. (2003). *Model-driven Architecture.* New York, New York: Wiley.

Gray, J., Bapty, T., Neema, S., Schmidt, D.C., Gokhale, A., & Natarajan, B. (2002). An Approach for Supporting Aspect-Oriented Domain Modeling. In *Proceedings of the Second International Conference on Generative Programming and Component Engineering*. Pittsburgh, Pennsylvania.

Gray, J., Bapty, T., Neema, S., & Tuck, J. (2001). Handling Crosscutting Constraints in Domain-specific Modeling. In *Communications of the ACM*, 44(10), 87-93.

Hillier, F.S. (2004). *Introduction to Operations Research.* New York, New York: McGraw-Hill.

Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., & Huh, M. (1998). FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architectures. In *Annals of Software Engineering,* 5(1), 143-168.

Mannion, M. (2002). Using First-order Logic for Product-line Model Validation. In *Proceedings of the Second International Conference on Software Product-lines.* San Diego, California**.**

Mellor, S.J. (2004). *MDA Distilled: Solving the Integration Problem with the Model Driven Architecture*. Boston, Massachusetts: Addison-Wesley.

Northrup, L., Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D.C., & Sullivan, K. (2006). *Ultra-Large-Scale*

Systems: The Software Challenge of the Future. Pittsburgh, Pennsylvania: Carnegie Mellon Software Engineering Institute**.**

Selic, B., & Rumbaugh, J. (1998). Using UML for Modeling Complex Real-time Systems. In *Lecture Notes In Computer Science*, 1474(1), 250-260.

Sterling, L.S., & Shapiro, E.Y. (1994). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Massachusetts: MIT Press.

Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*. Cambridge, Massachusetts: MIT Press.

Vaziri, M., & Jackson, D. (2000). Some Shortcomings of OCL, the Object Constraint Language of UML. In *Proceedings of the 34th International Conference on the Technology of Object-Oriented Languages and Systems*. Santa Barbara, California.

Warmer, J.B., & Kleppe, A.G. (2003). *Getting Your Models Ready for MDA*. Boston, Massachusetts: Addison-Wesley.

White, J. Czarnecki, K., Schmidt, D.C., Lenz, G., Wienands, C., Wuchner, E., & Fiege, L. (2007). Automated Model-based Configuration of Enterprise Java Applications. In *Proceedings of the Enterprise Computing Conference (EDOC) 2007*, Annapolis, Maryland.

White, J. Schmidt, D.C., Wuchner, E., & Nechypurenko, A. (2007). Automating Product-Line Variant Selection for Mobile Devices. In *Proceedings of the 11th Annual Software Product Line Conference (SPLC)*, Kyoto, Japan.

.