Model Intelligence: an Approach to Modeling Guidance

Jules White, Douglas C. Schmidt, Andrey Nechypurenko, and Egon Wuchner

Abstract

Model-Driven Engineering (MDE) facilitates building solutions in many enterprise application domains through its use of domain-specific abstractions and constraints. An important attribute of MDE approaches is their ability to check a solution for domain-specific requirements, such as security constraints, that are hard to evaluate using traditional source-code focused development efforts. The challenge in many enterprise domains, however, is finding a legitimate solution, not merely checking solution correctness. For these domains, model intelligence that uses domain constraints to guide modelers is needed. This paper shows how existing constraint specification and checking practices, such as the Object Constraint Language, can be adapted and leveraged to guide users towards correct solutions using visual cues.

Keywords: Domain-specific Modeling, Model-Driven Engineering, Constraint Checking, Constraint Reasoning, Modeling Guidance

1. Introduction

Model-Driven Engineering (MDE) [1] has emerged as a powerful approach to building complex enterprise systems. MDE allows developers to build solutions using abstractions, such as custom diagramming languages, tailored to their solution domain. For example, in the domain of deploying software to servers in a datacenter, developers can manipulate visual diagrams showing how software components are mapped to individual hosts, as shown in Figure 1.

A major benefit that MDE approaches provide is that custom constraints for each domain can be captured and embedded into an MDE tool. These domain constraints are properties, such as the memory demands of a software component on a server, that cannot be easily checked by a compiler or other third-generation programming language tool. The domain constraints serve as a domain solution compiler that can significantly improve the confidence in the correctness of a solution. The most widely used constraint specification language is the Object Constraint Language (OCL) [2].

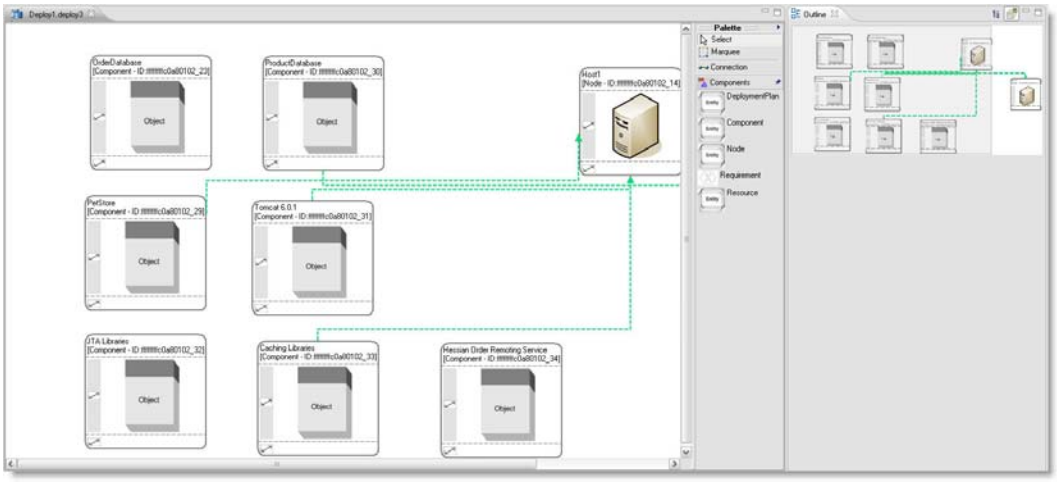

Figure 1. Deployment Model for a Datacenter

Although MDE can improve solution correctness and catch previously hard to identify errors, in many domains the major challenge is deriving the correct solution, not checking solution correctness. For example, when deploying software components to servers in a datacenter, each component can have numerous functional constraints, such as requiring co-hosting a specific set of other components with it, and non-functional constraints, such as requiring a firewalled host, that make developing a deployment model hard. When faced with large enterprise models with 10s, 100s, or 1,000s of model elements and multiple constraints per element, manual model building and validation approaches do not scale.

Enterprise models can also contain global constraints, such as stipulating that no host's allocation of components exceeds its available RAM, which further complicates modeling. Although languages like OCL can be used to validate a solution, they still do not make finding the correct solution any easier. Developers must still manually construct models and invoke constraint checking to see if a mistake has been made.

The following properties of enterprise models make building models challenging:

1. Enterprise models are often large and may contain multiple views, making it hard or infeasible for modelers to see all the information required to make a complex modeling decision
2. Constraints in enterprise systems often involve functional and non-functional concerns that are scattered across multiple views or aspects of a model and are hard to solve manually and
3. Enterprise modeling solutions may need to satisfy complex global constraints or provide optimality, both of which require finding and evaluating a large number of potential solution models.

Current model construction techniques are largely manual processes. The difficulty of understanding an entire large enterprise model—coupled with the need to find and evaluate a large number of potential solutions—makes enterprise modeling hard.

To motivate the need for tool support to help modelers deduce solutions to domain constraints, we use an application for modeling the deployment of software components to servers in a datacenter. Ideally, when creating a deployment, as a developer clicked on each individual software component to deploy it, the underlying tool infrastructure could use the domain constraints to derive the viable hosts for the component. We refer to these mechanisms for guiding modelers towards correct solutions as *model intelligence*.

2 Limitations of Current Constraint Checking Approaches

To motivate the challenges of using existing constraint infrastructure, such as OCL, as a guidance mechanism, we will evaluate a simple constraint for deploying a software component to a server. For each component, the host that it is deployed to should have the correct OS for which the component is compiled. This constraint can be captured in OCL as:

```
context:SoftwareComponent;
inv: self.hostingServer.OS = self.requiredOS;
```

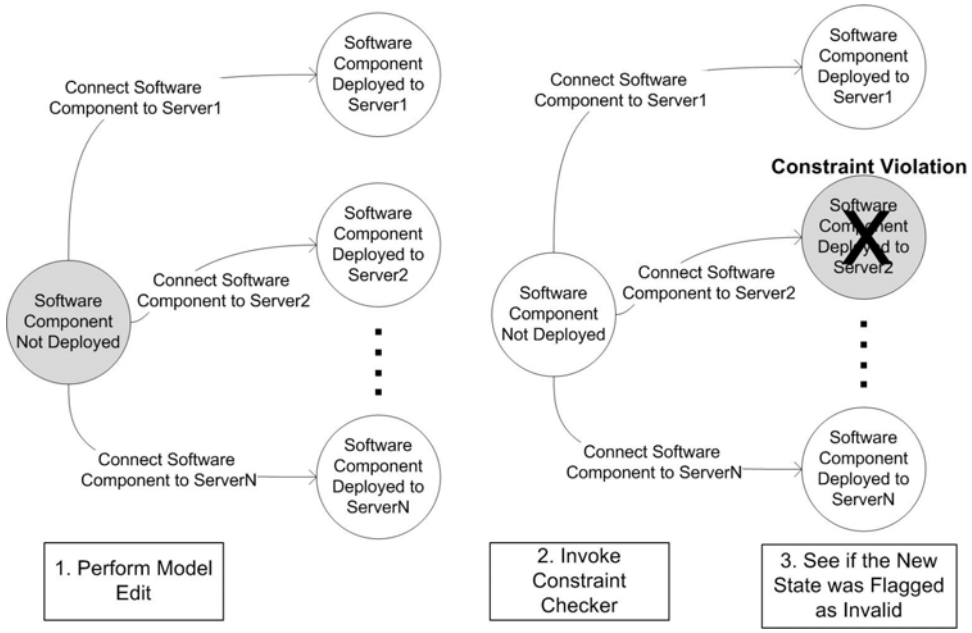

Figure 2. Model Editing and Constraint Checking

After a `SoftwareComponent` has been deployed to a server, the above constraint checks that the host (stored in the `hostingServer` variable) has the `OS` required by the component. As shown Figure 2, to utilize the constraint, the modeler first makes a change to the model (Step 1), invokes the constraint checker (Step 2), and then sees if an error state has been entered (Step 3). The challenge is that the modeler cannot predict ahead of time if the model is being transitioned to an invalid state. A state is only checked for errors after control has been transitioned to it.

One way around the inability to check the constraint before the host is committed to the SoftwareComponent is to use OCL preconditions as guards on transitions. An OCL precondition is an expression that must hold true before an operation is executed. The chief problem of using OCL preconditions as guards, however, is that they are designed to specify the correct behavior of an operation performed by the *implementation* of the model. Using an OCL precondition as a guard during modeling requires defining the constraint in terms of the operation performed by the *modeling tool* and not the model.

For example, the precondition that should be imposed to check for the correct OS is a constraint on an operation (*e.g.*, creating a connection) performed by the modeling tool, not by the model. To define the OCL precondition, therefore, developers must define the OCL constraint in terms of the modeling tool's definition of the operation, which may not use the same terminology as the model. Moreover, defining the constraint as a precondition on an operation performed by the modeling tool requires developers to create a duplicate constraint to check if an existing model state is correct.

Without two constraints—one to check the correctness of the modeling tool action and one to check the correctness of an already constructed model state—it is impossible to identify operation endpoints and ensure model consistency. The OCL precondition approach therefore adds complexity by requiring developers to maintain separate—and not necessarily identical—definitions of the constraint that can potentially drift out of sync. The precondition approach also couples the constraint to a single modeling platform since the precondition is defined in terms of the connection operation exposed by the tool, not the model.

3 Model Intelligence: an Approach to Modeling Guidance

A modeling tool can implement model intelligence, by using constraints to derive valid end states for a model edit *before* committing the change to the model. Traditional mechanisms of specifying constraints associate a constraint with objects (*e.g.*, SoftwareComponents) rather than the relationships between the objects (*e.g.*, the deployment relationship between a SoftwareComponent and a Server). To determine the validity of a relationship between two objects, therefore, the relationship must be created and committed to the model so that constraints on the two objects associated with the relationship can be checked.

The transitions in the state diagram from Figure 1 correspond to the creation of relationships between objects. To support model intelligence, a tool needs to use domain constraints to check the correctness of the modification of relationships between objects in a model before the modification is committed to the model. If constraints are associated with the relationships rather than the objects, a tool can use the constraints associated with the relationship to deduce valid end states and suggest transitions to a modeler.

3.1 Constraining Relationships

Relationships between objects are edges in the underlying object graph of a model. Each edge has a source and target object. Using this understanding of relationships, constraints can be created that specify the correctness of a relationship in terms of properties of the source and target elements. For example, the deployment of a SoftwareComponent to a Server is represented as a *deployment* relationship. A constraint can be applied to a deployment relationship and specified in terms of the properties of the source (*e.g.*, a SoftwareComponent) and the target (*e.g.*, a Server):

```
context:Deployment;
inv: source.requiredOS = target.OS;
```

A key property of associating constraints and specifying them in terms of the source and target of the relationship is that a constraint can be used to check the correctness of the creation of a relationship *before* the relationship is committed to a model. Prior to the creation of a relationship, the proposed source and target elements can be substituted into the constraint expression and the constraint expression checked for correctness. If the constraint expression holds true for the proposed source and target elements, the corresponding relationship can be created in the model.

Section 2 showed that using existing OCL approaches to model intelligence requires maintaining separate specifications of each constraint. If constraints are associated with relationships and expressed in terms of the source and targets of a relationship, they can be used to check the validity of a modeling action *before* it is committed to the model. Moreover, the same constraint can be used to check existing relationships between modeling elements, which can not be done with the standard OCL approach.

3.2 Relationship Endpoint Derivation

A model can be viewed as a knowledge base, *i.e.*, the model elements define facts about the solution. The goal of model intelligence is to run queries against the knowledge base to deduce the valid endpoints (*e.g.,*, valid hosts for a component) of a relationship that is being created by a modeler. In terms of the state diagram detailing a model editing scenario shown in Figure 3, the queries derive the valid states to which a model can transition.

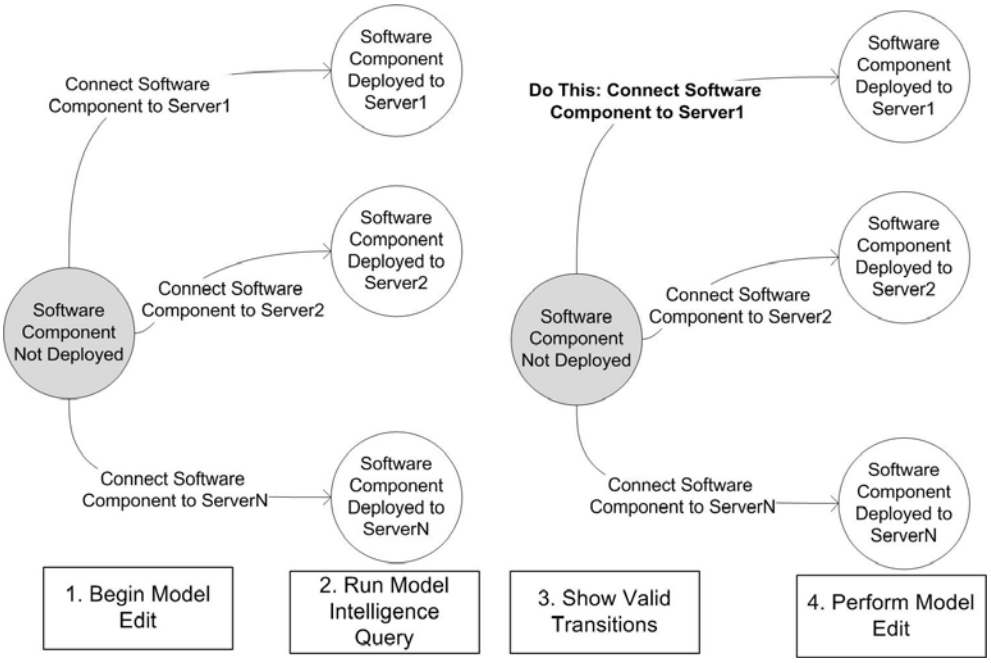

Figure 3. Model Editing Sequence for Model Intelligence

The creation of a relationship begins by modelers selecting a relationship type (*e.g.*, a deployment relationship) and one endpoint for the new relationship (*e.g.*, a `SoftwareComponent`). Model intelligence uses the relationship type to determine the constraints that must hold for the relationship and then uses the constraints to create queries to search the knowledge base for valid endpoints to create the

relationship, as shown in Step 2 of Figure 3. The valid endpoints determine the valid states to which the model can transition. As shown in Step 3 of Figure 3, the transitions that lead to these valid states can then be suggested to modelers as valid ways of completing an in-progress modeling edit.

The creation of a new relationship begins by the modeler selecting a source for the relationship and a type of relationship to create. Each relationship type has a set of constraints associated with it. Once model intelligence knows the source object and the OCL constraints on the relationship being modified, a query can be issued to find valid endpoints to complete the relationship. Using the OS deployment constraint from Section 2 the query to find endpoints for a deployment relationship would be:

```
Server.allInstances()->collect(target |
                               target.OS = source.OS);
```

In this example, model intelligence would specify to the OCL engine that the `source` variable mapped to the SoftwareComponent that had been set as the source of the deployment relationship. The query would then return the list of all Servers that had the correct OS for the component. For an arbitrary relationship, with constraint `Constraint`, between elements `Source` and `Target` of types `SourceType` and `TargetType`, a query can be composed to derive valid endpoints. Assuming that a relationship has endpoint `Source` set, a query can be issued to find potential values for `Target` as follow:

```
TargetType->allInstances()->collect(target | Constraint);
```

where `Constraint` is a boolean expression over the source and target variables. More generally, the query can be expressed as: *Find all elements of type TargetType where Constraint holds true if the source is Source*.

3.3 Endpoint Derivation Across Multiple Constraint Languages

Although we have only focused on OCL thus far, the generalized query definition from Section 3.2 can be mapped to other constraint or expression languages, as well. In prior work [4], we implemented model intelligence using OCL, Prolog, BeanShell, and Groovy. For example, Prolog naturally defines a knowledge base as a set of facts defined using predicate logic. Queries can be issued over a Prolog knowledge base by specifying constraints that must be adhered to by the facts returned. Model intelligence

can also be used to derive solutions that are restricted by a group of constraints defined in multiple heterogeneous languages. An iterative result filtering process can be used to derive endpoints that satisfy constraints specified in multiple languages, as shown in Figure 4.

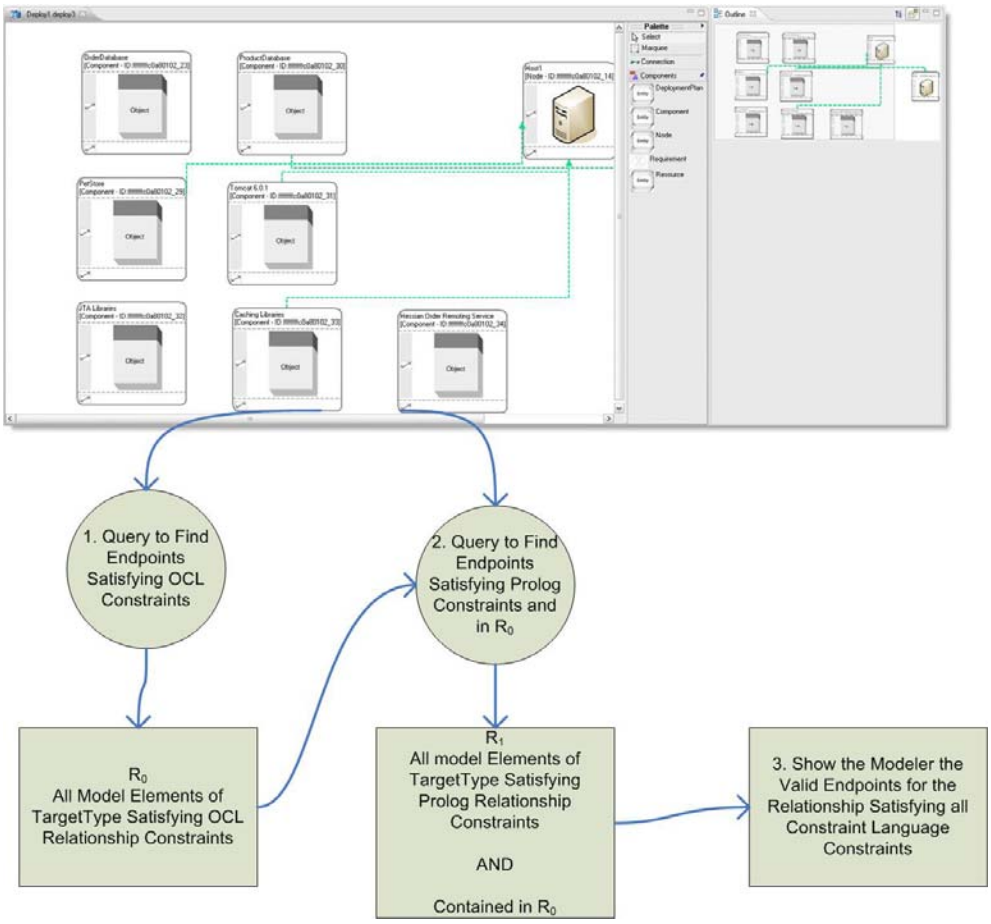

Figure 4. Model Intelligence Queries Across Multiple Constraint Languages

Initially, model intelligence issues a query to derive potential solutions that respect the constraint set of one constraint language. The results of the query are stored in the set R_0. For each subsequent query language C_i, the results of the query that satisfy the language's constraint set are stored in R_i. For each constraint language C_i, where $i > 0$, model intelligence issues a query using a modified version of the query format defined in Section 3.2: *Find all elements of type TargetType where Constraint holds true if the source is Source and the element is a member of the set* R_{i-1}.

The modified version of the query introduces a new constraint on the solution returned: all elements returned as a result were a member of the previous result set. A simple mechanism for specifying result sets is to associate a unique ID with each modeling element and to capture query results as lists of these IDs.

The modified queries can then be defined by checking to ensure that both the constraint set holds and the ID property of each returned modeling element is contained by the previous result set.

4 Integrating Model Intelligence with the Command Pattern

There are a large number of uses for model intelligence, including automatically performing an autonomous batch process of model edits and providing visual feedback to modelers. In this section, we show how model intelligence can be integrated with the *Command pattern* [3] to provide visual cues to aid modelers in correctly completing modeling actions. The Command pattern uses an object to encapsulate an action and its needed data and is used in many graphical modeling frameworks, such as the Eclipse Graphical Editor Framework [5]. As a modeler edits a model, commands are created and executed on the model to perform the actions of the modeler.

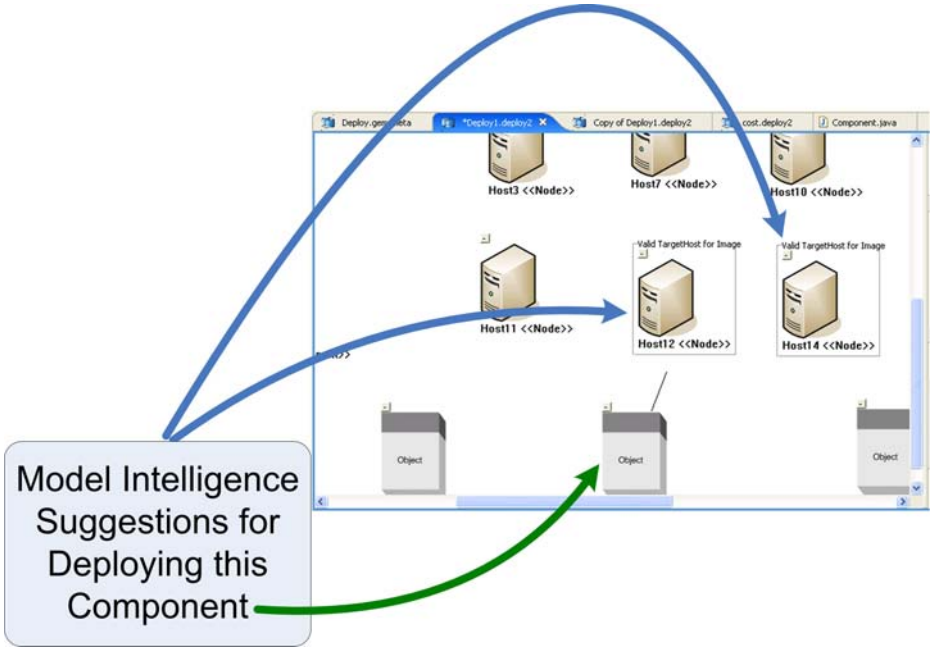

Figure 5. The Deployment Command Showing Valid Endpoints Derived via Model Intelligence

Modeling platforms provide tools, such as a connection tool, that a modeler uses to manipulate a model. Each tool is backed by an individual command object, such as a connection command. When a modeler chooses a tool, an instance of the corresponding command class is created. Subsequent pointing, clicking, and typing by the user, sets the arguments (*e.g.*, connection endpoints) operated on by the command. When

the arguments of the command are fully specified (*e.g.*, both endpoints of a connection command are set), the command executes.

Section 3 described the ability to highlight the valid deployment locations for a software component after a modeler clicked on it to initiate a deployment connection. This functionality can be achieved by combining model intelligence with a deployment connection command. After the initial argument to the deployment connection command is set, the command can use model intelligence to query for valid deployment locations. If there is a single server that can host the component, the command can autonomously choose it as the deployment location and execute. If there is more than one potential valid host, each host can be highlighted via a command to help the user select the command's final argument, as shown in Figure 5.

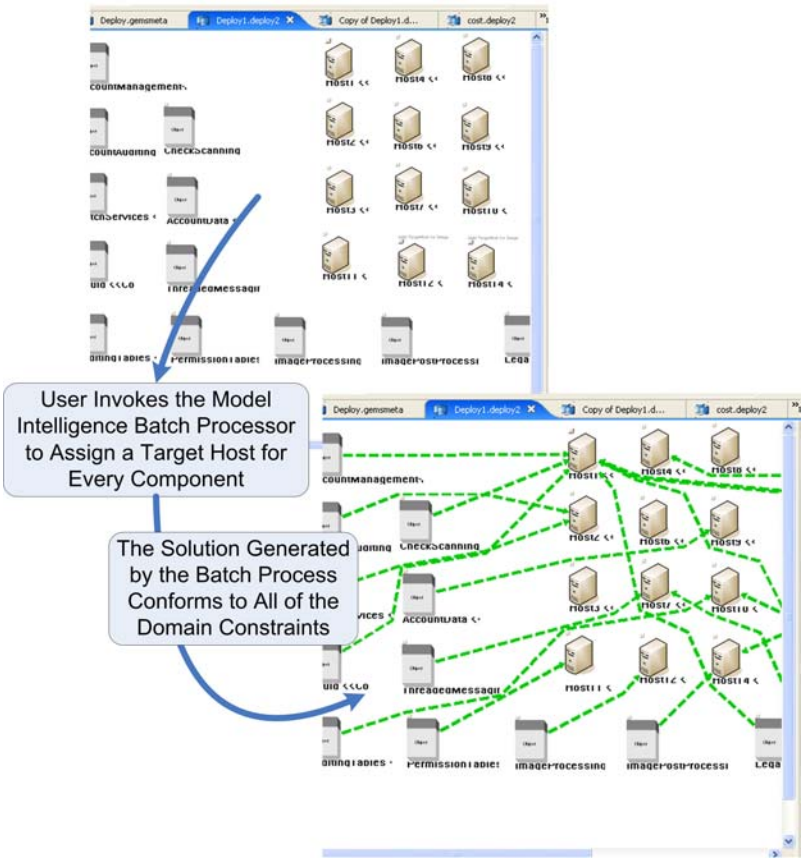

Figure 6. A Model Intelligence Batch Process to Assign a Host for Every Component

5 Concluding Remarks

Our experience developing models for enterprise application domains indicates that simply determining if a model is correct is not always helpful. We have learned that using constraints to verify the correctness of relationships between objects—rather than just individual object states—allows modeling tools to guide modelers towards correct solutions by suggesting ways of completing edits. Moreover, batch processes can be built atop of suggestion mechanisms to allow tools to autonomously complete sets of modeling actions. For example, a batch process can be created to deploy a large group of software components, by deriving sets of valid hosts for each component and intelligently selecting a host from each set, as shown in Figure 6. In other work [4], we have used model intelligence as the basis for creating batch modeling processes that use constraint solvers to automate large sets of modeling actions and optimally select endpoints for relationships to satisfy global constraints or optimization goals.

Our implementation of model intelligence for the Eclipse Modeling Framework [6], called GEMS EMF Intelligence, is an open-source project available from www.eclipse.org/gmt/gems.

References

[1] J. Bézivin. "In Search of a Basic Principle for Model Driven Engineering," Novatica/Upgrade, V(2):21—24, (2004).

[2] J.B. Warmer, A.G. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*, Addison-Wesley Professional, New York, NY, USA, (2003).

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley, Boston, MA, USA (1995).

[4] J. White, A. Nechypurenko, E. Wuchner, and D.C. Schmidt. "Reducing the Complexity of Designing and Optimizing Large-scale Systems by Integrating Constraint Solvers with Graphical Modeling Tools, "*Designing Software-Intensive Systems: Methods and Principles*, edited by Dr. Pierre F. Tiako, Langston University, Oklahoma, USA, (2008).

[5] Graphical Editor Framework, www.eclipse.org/gef.

[6] F. Budinsky, S.A. Brodsky, E. Merks. Eclipse Modeling Framework, Pearson Education, Upper Saddle River, NJ, USA, (2003).