UPGRADE is the European Journal for the Informatics Professional, published bimonthly at <http://www.upgrade-cepis.org/>

Publisher
UPGRADE is published on behalf of CEPIS (Council of European Professional Informatics Societies, <http://www.cepis.org/>) by **Novática** <http://www.ati.es/novatica/>, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática,* <http://www.ati.es/>)

UPGRADE monographs are also published in Spanish (full version printed; summary, abstracts and some articles online) by **Novática**

UPGRADE was created in October 2000 by CEPIS and was first published by **Novática** and **INFORMATIK/INFORMATIQUE**, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <http://www.svifsi.ch/>)

UPGRADE is the anchor point for **UP**ENET (**UP**GRADE European NETwork), the network of CEPIS member societies' publications, that currently includes the following ones:
- **Informatica**, journal from the Slovenian CEPIS society SDI
- **Informatik-Spektrum**, journal published by Springer Verlag on behalf of the CEPIS societies GI, Germany, and SI, Switzerland
- **ITNOW**, magazine published by Oxford University Press on behalf of the British CEPIS society BCS
- **Mondo Digitale**, digital journal from the Italian CEPIS society AICA
- **Novática**, journal from the Spanish CEPIS society ATI
- **OCG Journal**, journal from the Austrian CEPIS society OCG
- **Pliroforiki**, journal from the Cyprus CEPIS society CCS
- **Pro Dialog**, journal from the Polish CEPIS society PTI-PIPS
- **Tölvumál**, journal from the Icelandic CEPIS society ISIP

Editorial Team
Chief Editor: Llorenç Pagés-Casas
Deputy Chief Editor: Francisco-Javier Cantais-Sánchez
Associate Editor: Rafael Fernández Calvo

Editorial Board
Prof. Wolffried Stucky, CEPIS Former President
Prof. Nello Scarabottolo, CEPIS Vice President
Fernando Piera Gómez and Llorenç Pagés-Casas, ATI (Spain)
François Louis Nicolet, SI (Switzerland)
Roberto Carniel, ALSI – Tecnoteca (Italy)

UPENET **Advisory Board**
Matjaz Gams (Informatica, Slovenia)
Hermann Engesser (Informatik-Spektrum, Germany and Switzerland)
Brian Runciman (ITNOW, United Kingdom)
Franco Filippazzi (Mondo Digitale, Italy)
Llorenç Pagés-Casas (Novática, Spain)
Veith Risak (OCG Journal, Austria)
Panicos Masouras (Pliroforiki, Cyprus)
Andrzej Marciniak (Pro Dialog, Poland)
Thorvardur Kári Ólafsson (Tölvumál, Iceland)
Rafael Fernández Calvo (Coordination)

English Language Editors: Mike Andersson, David Cash, Arthur Cook, Tracey Darch, Laura Davies, Nick Dunn, Rodney Fennemore, Hilary Green, Roger Harris, Jim Holder, Pat Moody, Brian Robson

Cover page designed by Concha Arias Pérez
"Golden Ratio" / © ATI 2008
Layout Design: François Louis Nicolet
Composition: Jorge Llácer-Gil de Ramales

Editorial correspondence: Llorenç Pagés-Casas <pages@ati.es>
Advertising correspondence: <novatica@ati.es>

UPGRADE **Newslist** available at
<http://www.upgrade-cepis.org/pages/editinfo.html#newslist>

Copyright
© Novática 2008 (for the monograph)
© CEPIS 2008 (for the sections **UP**ENET and CEPIS News)
All rights reserved under otherwise stated. Abstracting is permitted with credit to the source. For copying, reprint, or republication permission, contact the Editorial Team

The opinions expressed by the authors are their exclusive responsibility

ISSN 1684-5285

Monograph of next issue (June 2008)

"Next Generation Technology-Enhanced Learning"

(The full schedule of **UP**GRADE is available at our website)

The European Journal for the Informatics Professional
http://www.upgrade-cepis.org

Vol. IX, issue No. 2, April 2008

2 Editorial
New **UP**ENET Partners — *Niko Schlamberger* (President of CEPIS)

2 From the Chief Editor's Desk
Welcome to our Deputy Chief Editor — *Llorenç Pagés-Casas* (Chief Editor of **UP**GRADE)

Monograph: Model-Driven Software Development
(published jointly with Novática*)
Guest Editors: *Jean Bézivin, Antonio Vallecillo-Moreno, Jesús García-Molina, and Gustavo Rossi*

4 Presentation. MDA® at the Age of Seven: Past, Present and Future — *Jean Bézivin, Antonio Vallecillo-Moreno, Jesús García-Molina, and Gustavo Rossi*

7 A Brief History of MDA — *Andrew Watson*

12 MDA Manifestations — *Bran Selic*

17 The Domain-Specific IDE — *Steve Cook and Stuart Kent*

22 Model Intelligence: an Approach to Modeling Guidance — *Jules White, Douglas C. Schmidt, Andrey Nechypurenko, and Egon Wuchner*

29 Model Differences in the Eclipse Modelling Framework — *Cédric Brun and Alfonso Pierantonio*

35 Model-Driven Architecture® at Eclipse — *Richard C. Gronback and Ed Merks*

40 Model-Driven Web Engineering — *Nora Koch, Santiago Meliá-Beigbeder, Nathalie Moreno-Vergara, Vicente Pelechano-Ferragud, Fernando Sánchez-Figueroa, and Juan-Manuel Vara-Mesa*

UPENET (UPGRADE European NETwork)

46 From **Informatik Spektrum** (GI, Germany, and SI, Switzerland)
High Performance Computing
The TOP500 Project: Looking Back over 15 Years of Supercomputing — *Hans Werner Meuer*

62 From **Mondo Digitale** (AICA, Italy)
Project Management
Critical Factors in IT Projects — *Marco Sampietro*

CEPIS NEWS

68 CEPIS Projects
Selected CEPIS News — *Fiona Fanning*

* This monograph will be also published in Spanish (full version printed; summary, abstracts, and some articles online) by **Novática**, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*) at <http://www.ati.es/novatica/>.

Model Intelligence: an Approach to Modeling Guidance

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

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: Constraint Checking, Constraint Reasoning, Domain-Specific Modeling, Model-Driven Engineering, 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].

Although MDE can improve solution correctness and catch previously hard to identify errors, in many domains

Authors

Jules White is a Ph.D. student in the Department of Electrical Engineering and Computer Science (EECS) at Vanderbilt University. His research focuses on using constraint optimization techniques for modeling guidance and optimization, constraint-based automated assembly of component applications, model-driven development, and distributed Java systems. He is currently the lead developer of the Generic Eclipse Modeling System (GEMS) <http://www.eclipse.org/gmt/gems>, a part of the Eclipse GMT project. Before joining the DOC group, he worked for IBM's Cambridge Innovation Center and was involved with constraint modeling and rule-based systems. <jules.white@ gmail.com>.

Douglas C. Schmidt is a Full Professor in the Electrical Engineering and Computer Science (EECS) Department, Associate Chair of the Computer Science and Engineering program, and a Senior Research Scientist at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, Nashville, TN. For the past two decades, he has led pioneering research on patterns, optimization techniques, and empirical analyses of object-oriented and component-based frameworks and model-driven development tools that facilitate the development of distributed middleware and applications. Dr. Schmidt is an expert on distributed computing patterns and middleware frameworks and has published over 350 technical papers and 9 books that cover a range of topics including high-performance communication software systems, parallel processing for high-speed networking protocols, real-time distributed object computing, object-oriented patterns for concurrent and distributed systems, and model-driven development tools. <d.schmidt@vanderbilt.edu>.

Egon Wuchner works as a researcher and consultant in the Coporate Technology SE2 department of Siemens AG in Munich, Germany. He is an expert in software architecture and distributed systems. His research focuses on concepts, technologies and tools to improve the development of large distributed systems, e.g. their handling of operational requirements, their comprehensibility and maintanability. His recent research has been in Aspect-oriented Software Development and Model-Driven Development. <egon.wuchner@siemens.com>.

Andrey Nechypurenko is a senior software engineer in Siemens AG Corporate Technology (CT SE2). Mr. Nechypurenko provides consulting services for Siemens business units focusing on distributed real-time and embedded systems. Mr. Nechypurenko also participates in research activities on model driven development and parallel computing. Before joining Siemens AG, he worked in the Ukraine on high performance distributed systems in the telecommunications domain. <andrey.nechypurenko@ siemens.com>.

 © Novática

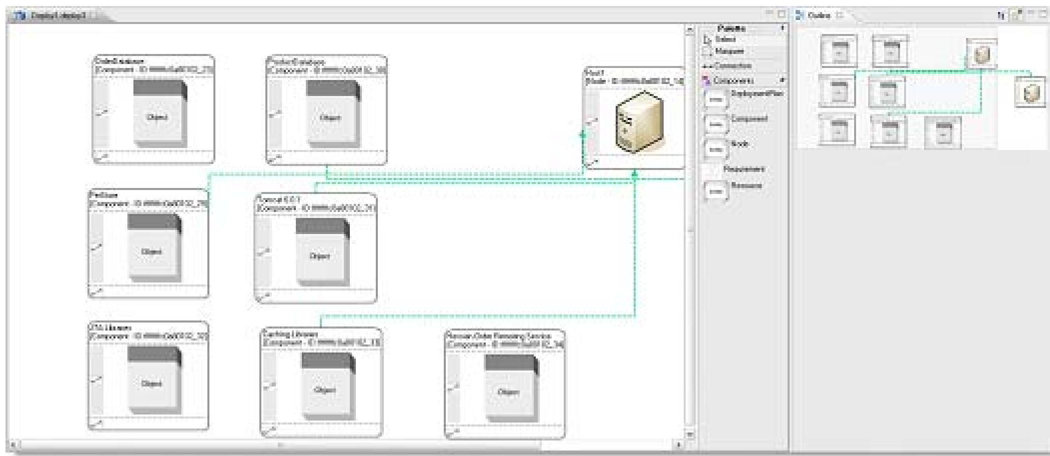

Figure 1: Deployment Model for a Datacenter.

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:

- 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.
- Constraints in enterprise systems often involve func-

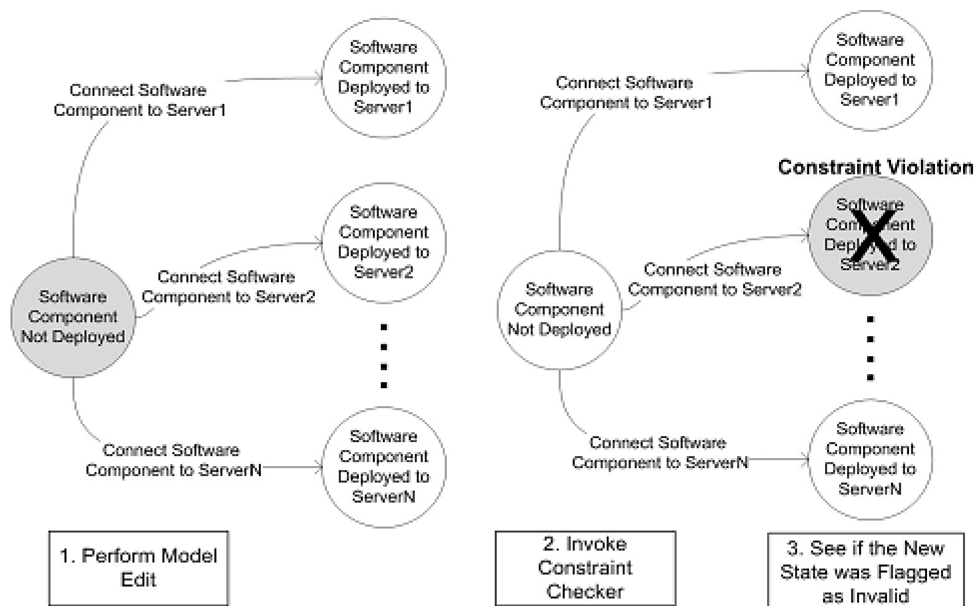

Figure 2: Model Editing and Constraint Checking.

© Novática

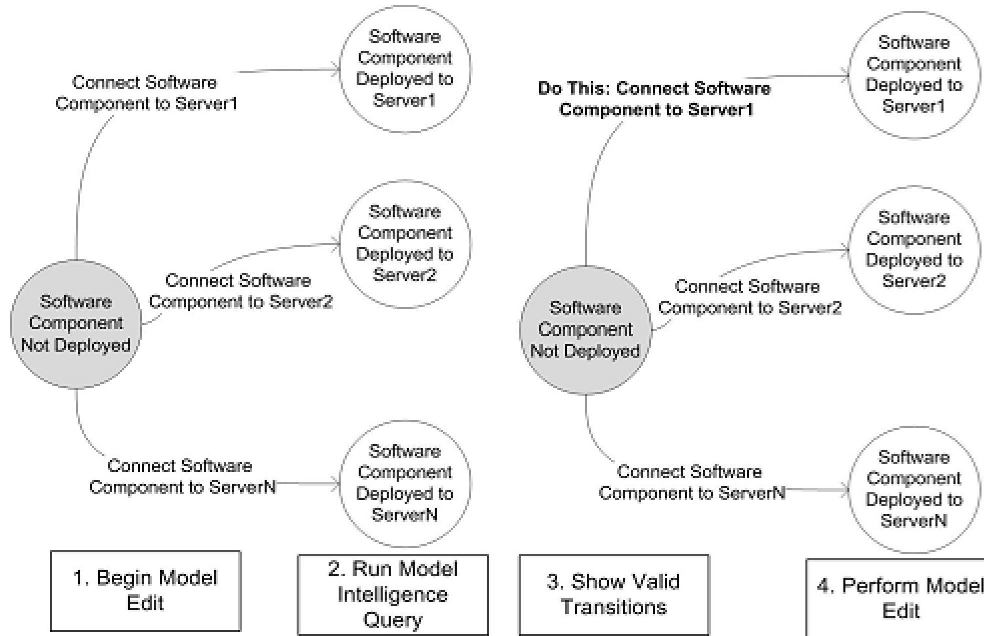

Figure 3: Model Editing Sequence for Model Intelligence.

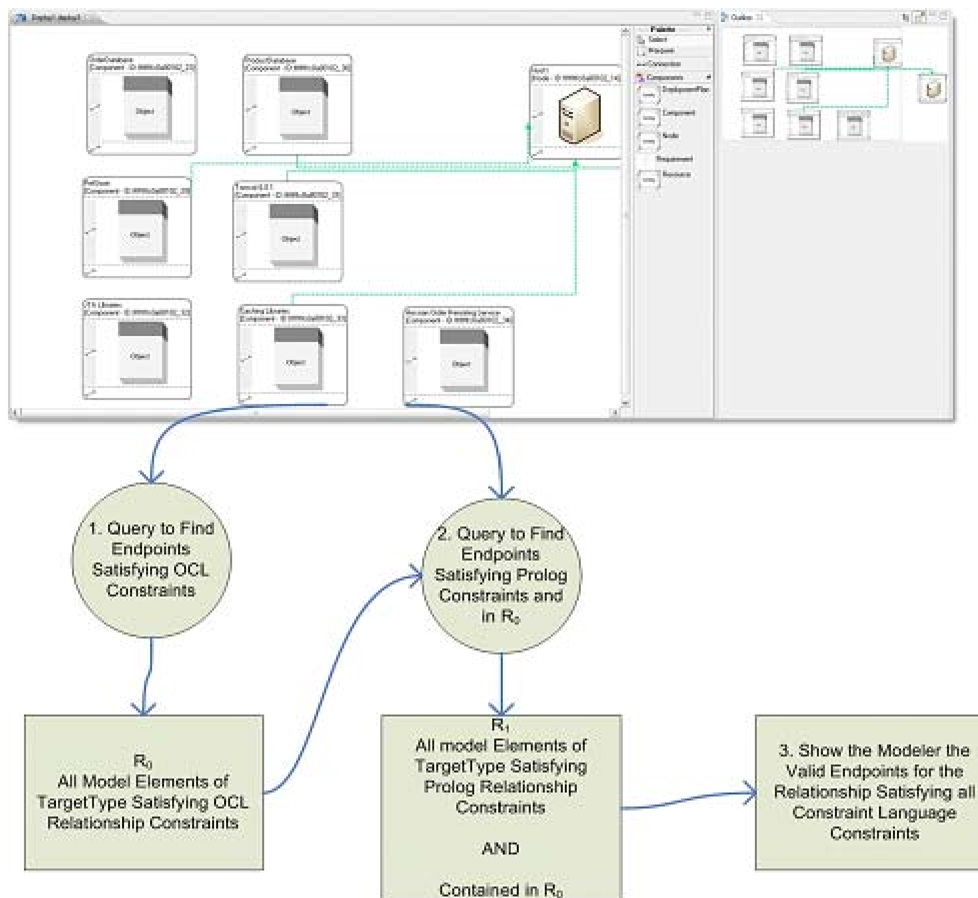

Figure 4: Model Intelligence Queries Across Multiple Constraint Languages.

 © Novática

tional and non-functional concerns that are scattered across multiple views or aspects of a model and are hard to solve manually, and

- 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;
```

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 2 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`):

© Novática

```
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.

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 com-

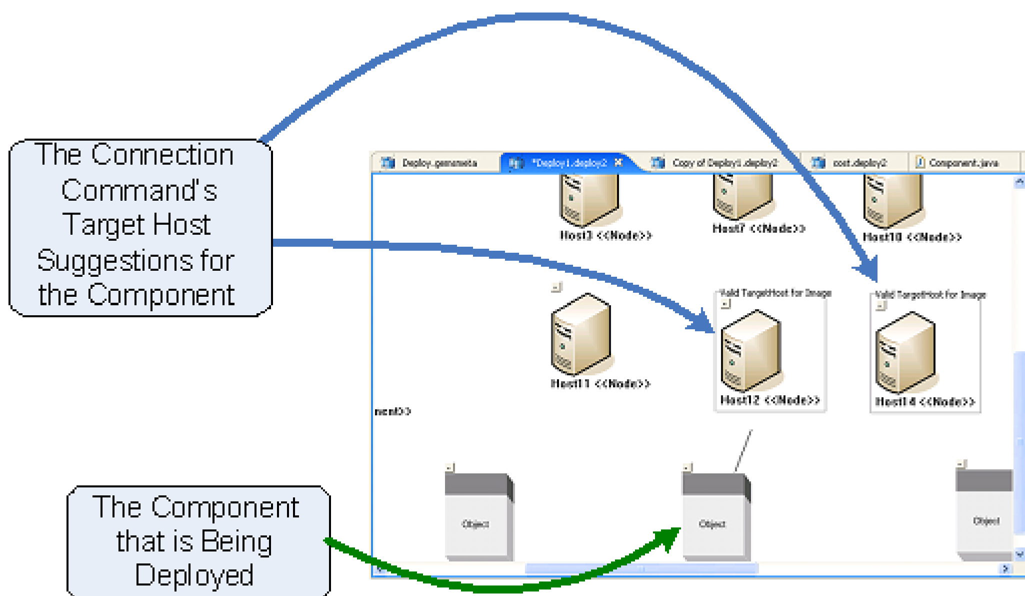

Figure 5: The Deployment Command Showing Valid Endpoints Derived via Model Intelligence.

 © Novática

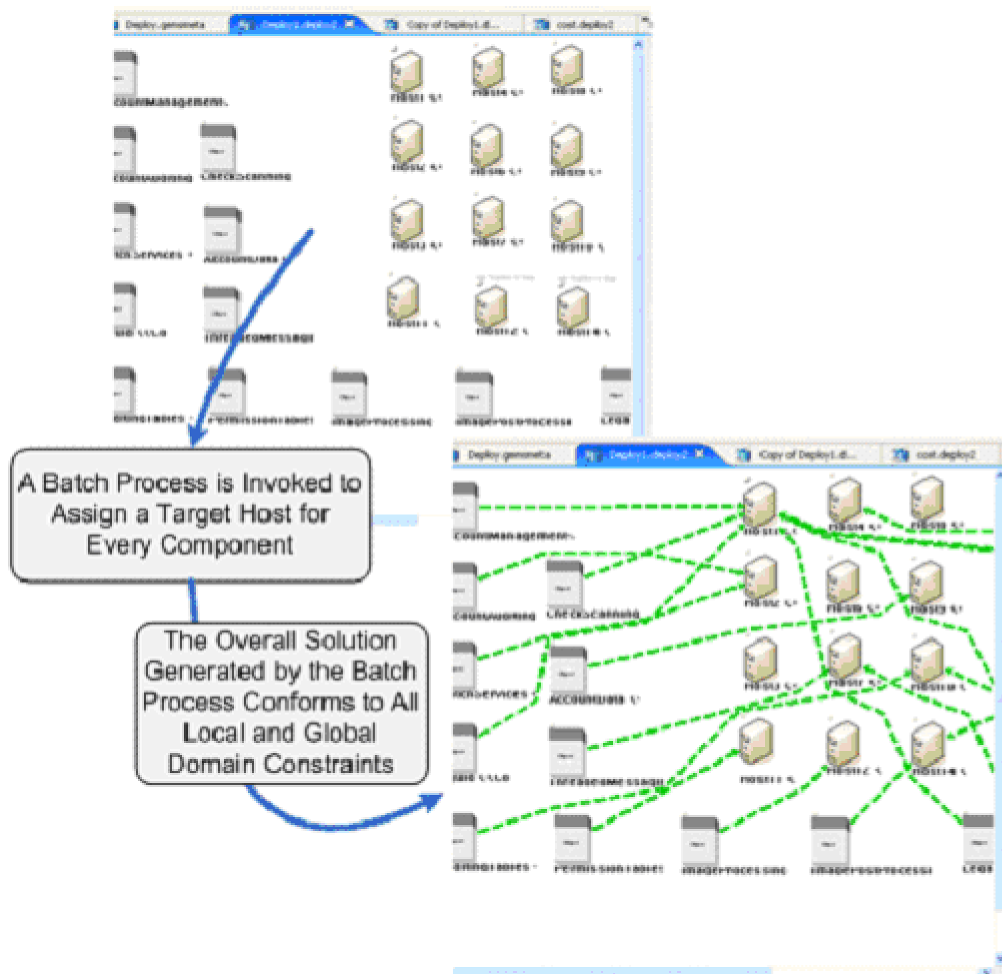

Figure 6: A Model Intelligence Batch Process to Assign a Host for Every Component.

posed 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.

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

© Novática

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.

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.

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. ISBN: 0321179366.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns: Elements of Reusable Object-oriented Software", Addison-Wesley, Boston, MA, USA, 1995. ISBN: 0201633612.

[4] J. White, A. Nechypurenko, E. Wuchner, D.C. Schmidt. "Reducing the Complexity of Designing and Optimizing Large-scale Systems by Integrating Constraint Solvers with Graphical Modeling Tools", in "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.

 © Novática