Chapter 8.14
Reducing the Complexity of Modeling Large Software Systems

Jules White
Vanderbilt University, USA

Douglas C. Schmidt
Vanderbilt University, USA

Andrey Nechypurenko
Siemens AG, Germany

Egon Wuchner
Siemens AG, Germany

ABSTRACT

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 integrate a constraint solver into a modeling tool.

INTRODUCTION

Model-driven development (MDD) (Ledeczi, 2001a; Kent, 2002; Kleppe, Bast, & Warmer, 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 & Karsai, 1997). MDD tools and techniques help improve software quality by automating constraint checking (Sztipanovits & Karsai, 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 antilock 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 & Kuhne, 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, because 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 a third-generation programming language approach (such as specifying the solution in C++), high-level information (such as messaging periodicity or memory consumption) is lost. Because 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 & Karsai, 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 the 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.

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