Embedded Software Design Automation

Rance Cleaveland David Hansel Steven Sims Scott Smolka
Reactive Systems, Inc.
120-B East Broad Street
Fall Church, VA 22046

Abstract

We argue that *operational modeling and analysis* of software systems offers an attractive avenue for improving both the productivity of software engineers and the quality of the *control-intensive* software they produce. Operational modeling forms the basis for the tools offered by Reactive Systems, Inc. (RSI) in support of *Embedded Software Design Automation* (ESDA), an emerging discipline that provides modeling and analysis tools for developers of embedded software. RSI expects that ESDA will make embedded-software development much more like the processes followed in more established engineering disciplines such as electrical, mechanical, and aerospace.

1 Introduction

In traditional engineering disciplines such as mechanical or electrical, engineers routinely use mathematical modeling and analysis techniques during the design and development phases of products and processes. Such models provide precise feedback on design decisions much more quickly and at a much lower cost than would be otherwise possible. For example, aerospace engineers employ differential and integral equations to model air flow around aircraft and to analyze pressure loads. Such *virtual wind tunnel* methodologies allow experimentation with different designs without the use of expensive mockups in actual wind tunnels. The advent of computer-aided design and simulation tools has further reduced the costs associated with the use of mathematical modeling in engineering, and one may find several commercially successful packages on the market.

In contrast, the use of rigorous modeling techniques in the development of software remains virtually non-existent. Current software-development practice largely relies on informal approaches such as code review and testing to ensure that a system implementation adheres to its requirements. The general lack of formal modeling at design time means that reviews and testing may only be applied when code has been developed, at which point significant resources have already been expended.

The high costs of software development, the frequency with which software-development projects fail, and the relatively low quality of many software products, may be attributed in part to this lack of formalized design support for software. Traditional Computer-Aided

Software Engineering (CASE) tools provide some assistance, but only at the level of project management and design documentation. The facilities for modeling and analyzing system behavior, even of modern, commercially successful CASE environments, remain quite primitive in comparison with the analytical tools available to engineers in more-established branches of engineering.

Reactive Systems, Inc. (RSI) believes that a compelling basis for the rigorous modeling of *control-intensive* (as opposed to data-intensive) software can be found in the recent emergence of mathematically precise, yet operational and hence easy-to-grasp, approaches to modeling systems and efficient techniques for validating such models. Control-intensive software is found prominently in embedded applications such as automobiles, cardiac pacemakers, aircraft and cell phones; RSI is developing a collection of technologies we refer to as *embedded-software design automation* (ESDA) that provides operational yet rigorous modeling, analysis and implementation support for embedded-software engineers. The mathematical precision ESDA brings to the traditionally *ad hoc* arena of software design will greatly enhance engineer productivity while improving the quality of the software produced. As a result, RSI expects that ESDA users will be able to reduce their embedded-software costs, including those related to development, software-related product failures, and liability, by 50% or more.

RSI's vision is that within 10 years, every embedded-software developer will use rigorous ESDA technology to improve software quality and reduce costs. The Company foresees the ESDA industry becoming as important as the Electronic Design Automation industry, which produces digital hardware design tools and had revenues in 2000 of $3.8 billion.

2 Methodology

RSI's ESDA environment is built around executable models. Users formulate these models in notations that have a notion of execution step. These models might be given in existing notations such as Simulink/Stateflow or UML state machines, or they can be given in RSI-proprietary notations that provide support for *model interoperability* by providing rigorous yet easy-to-use constructs for "glueing together" submodels into asynchronously executing system models. Once an executable model is in place, a variety of different techniques may be used to study their behavior, including:

Simulation. The results of model execution may be studied for aberrant behavior.

Testing. Models may be subjected to tests.

Model checking: Models may be (semi-)exhaustively executed and the resultant executions checked for undesirable properties.

Once a stable model is produced, it may be used as a basis for further development activities: tests for source code may be generated from them, as can the eventual source itself in many cases. The models also serve as "active documentation" that can be studied and executed by maintainers and system enhancers.

Development Stage	**Automated Support Offered by ESDA**
Requirements	Capture requirements as scenarios, logical formulas, abstract models.
Design	Support model construction and validation against requirements.
Implement	Generate code from models, or provide models as *blueprints* for manual implementation.
Test	Generate test suites from models – for both unit and integration testing, as well as for model reconstruction from existing code.
Monitor	Generate deployable on-chip monitoring code from models.
Maintain	Use models as documentation, aid for debugging (together with diagnostic information from monitors), and basis for assessing new system features.

Figure 1: Model-based software development using ESDA.

Figure 1 summarizes how ESDA supports different stages of the software development life-cycle, with models serving as the link from one stage to the next. Ideally, use of ESDA tools begins during the requirements stage and continues throughout the life-cycle, as indicated in Figure 1. ESDA can also be effectively applied to *legacy systems*. To re-engineer an existing system, engineers would build models reflecting their understanding of the system. They would then deploy simulate the model's to check that their behavior accords with their expectations. Tests could then be generated and applied to both the model and the original system; the responses to these tests would help the engineer determine whether or not s/he has correctly captured the system's functionality in the model. Once this determination has been made, documentation templates could be generated from the model.

3 Reactis

RSI's Reactis tool suite is intended to reify the ESDA vision outlined in Section 2 by providing mathematically rigorous modeling, analysis and implementation support for embedded-software engineers. By automating coding tasks and providing precise design-time feedback to designers, Reactis is intended to speed software development, lower costs, and improve software quality.

Reactis will comprise six tools, each of which is described below.

Reactis Modeler allows users to develop structured models of embedded software. The models consist of hierarchically structured “networks” of submodels that interact with one another using “communication links”. Submodels may take different forms.

- They may be subnetworks.
- They may be individual component models defined in RSI’s proprietary notation.
- They may be models developed using other tools and “imported” into Modeler.

To ensure semantic compatibility with non-RSI modeling notations, the Company has given its modeling language a flexible, yet mathematically precise, “foreign language” interface. In particular, it is possible to use Modeler to design models containing submodels developed in the Simulink/Stateflow languages of The MathWorks; the STATEMATE notation of I-Logix; the SDL notation of Telelogic; and the UML state machine notation of Rational.

Reactis Simulator allows models to be simulated interactively and automatically so that design bugs may be uncovered without the designs having to be implemented. Once users create a model, they can step through the execution of the model using Simulator, which functions like traditional debuggers from programming languages: designers may explore the execution of their designs an execution step at a time, or they may set break points and let the system run autonomously until a break point is reached. Simulator also supports replay and reverse execution for models given in RSI’s notations, thereby allowing simulations to be stopped, “backed up” and restarted. The tool also includes a “smart simulation” mode that allows users to bypass parts of a model that have been visited during a prior simulation run, thereby avoiding duplicate simulation effort.

Reactis Validator rigorously checks for consistency properties of system models. This tool enables the early detection of design errors and inconsistencies and reduces the effort required for design reviews. Using Validator, an engineer can automatically check his or her models for any of a number of generic consistency violations, including the following.

- Undefined variables
- Type errors
- Missing cases
- Nondeterminism
- Dead code
- Deadlock

Validator performs such checks using an intelligent, systematic exploration of the model’s potential behaviors. If a consistency property is found to be violated, the tool provides the user with an explanation in the form of an execution sequence leading to the place where the violation occurs. The engineer may then use the simulator to step through this execution to locate the source of the problem.

Validator also allows engineers to define custom checks tailored to a particular application.

Reactis Tester generates test cases from models. These test cases may then be fed to the model and its implementation (source code) and the outputs compared to determine whether or not the code conforms to its model. The tool uses technology derived from Validator to systematically explore system models; by specifying different *coverage criteria* (such as decision coverage, statement coverage, modified condition/decision coverage), a user may precisely indicate how exhaustive the testing should be. The tool eliminates the need for producing test suites manually and reduces the time spent in testing by ensuring that unnecessary tests are avoided.

Reactis Coder generates executable code from embedded-system models. By guaranteeing conformance between the code it generates and the models it is given, Coder is intended to reduce both the time consumed in manual coding and the effort expended in testing.[1] The source code produced by Coder is also useful for prototyping purposes, and the structural similarity between Coder-generated code and the models from which it is derived makes it easy for software engineers to review and modify. Coder generates C and C++ code, and current plans call for the addition of Java-generation capability. For reasons of portability, Coder-produced source code targets an RSI-designed API (application programming interface); to run the produced code on their platforms users need an implementation of this API. The Company expects to provide implementations for popular commercial real-time operating systems such as Wind River's VxWorks and intends to work on a consulting basis to develop efficient implementations of the API for companies with proprietary platforms.

Reactis Tracker generates custom run-time monitors from models. Tracker is designed to catch subtle errors that escape analysis and testing and show up "in the field." A generated monitor is incorporated into an application to observe the application's implementation, for the purpose of identifying anomalous behaviors. When such behavior is detected, the monitor stores diagnostic information in the form of event sequences leading to the problem, takes associated corrective measures, and alerts the proper personnel of the situation. The technology in Tracker borrows from both Validator and Coder. Tracker-generated monitors can be thought of as "software black boxes" that, in analogy with the "black boxes" on aircraft, monitor embedded software for diagnostic purposes.

4 Conclusions

This document has argued for the use of mathematically rigorous, operationally based modeling in the design and development of control-intensive embedded software. It has also outlined the commercial opportunities we see in providing tools based on these ideas to embedded-software developers.

[1] In practice, Coder-generated source is still likely to be subjected to testing, both for regulatory reasons (agencies such as the Food and Drug Administration and the Federal Aviation Agency mandate software testing), and cultural ones (users may be wary about relying on untested, automatically produced code).