Verification of Platform Independent Models

Gregory T. Eakman, Ph.D.
Pathfinder Solutions
grege@pathfindersol.com

Abstract

Model Driven Architecture raises the level of abstraction of embedded systems development, but also adds to the already difficult problem of testing these systems. This paper outlines a development and integration test approach to testing components and systems developed as executable platform independent models. The approach is based on research and practical experience, and built on model transformation processes and formalized design-for-test instrumentation code injected during transformation.

Introduction

Model Driven Architecture (MDA) is a framework developed by the OMG to specify applications and components as platform independent models (PIMs) and use transformations to create platform specific models (PSMs) and source code [1]. Modeling at a higher level of abstraction provides the benefits of technology independence, architectural integrity enforced by the transformations, improved communications, and maintainability, but introduces new challenges for testing.

The use of UML models as the basis for component reuse with domains differs from current approaches to component-based software engineering. Most common definitions of a software component deal with the binary image. But reuse at the binary and even source code levels has the limitation that platform specific design and implementation decisions are fixed.

MDA holds much promise as an extension to component technology, at a higher level of abstraction. Key issues include testability of platform independent models in new environments and platforms. PIM providers must address these issues if they hope to encourage developers to reuse the components.

Challenge to PIM Reuse: Trustworthiness

Building software and systems from pre-existing components has long been the goal of the software engineering community. Among other obstacles in achieving this goal is the difficulty of testing that the components are integrated correctly to produce a correct system. The challenges to PIM reuse within embedded systems are:

- Testability of embedded systems
- Testing components in a new environment
- Testing PIMs on a new platform

Embedded systems suffer from testability problems in the areas of *observability*, the ability to detect errors in control flow or internal state during a test, and *controllability*, the ability to cause the software to execute a particular path. System outputs are the only way to infer what is going on inside these systems. Embedded debuggers induce probe effect, affecting timing and results. Also, controlling the sequence and timing of inputs to generate a particular output can be costly, if possible at all. For example, one test case of a medical device required over 30 hours of setup to put the device in the proper initial state for the test.

PIMs suffer from the same reuse issues as any other software component in a new environment or application – verifying that the component behaves properly. Even if it works in one application, a new application will send it messages with different ordering, data, ranges of values, and timing. The new environment may not even behave properly with respect to the PIM's contract, explicit or implicit.

Even a trusted component should be retested in a new environment. The cause of the Ariane 5 rocket failure in France, which exploded soon after takeoff on its first flight, was attributed to software reused

from a previous version of the rocket. Although many practitioners have claimed that the use of one method or another would have prevented the failure, the reused software had simply not been adequately tested in its new environment [2].

Platform independence adds yet another obstacle to reuse and testability. PIMs are at a higher level of abstraction and require a set of transformation rules to map them onto the target platform. These transformation tools are relatively new and still maturing. Also, in order to meet diverse performance and memory of embedded systems, the transformation rules must be open and controllable, but this also leaves them susceptible to incorrectly applied optimization transformations.

The platforms of embedded systems are extremely diverse. It would be easy for a PIM developer to make incorrect assumptions about characteristics of target platforms, or for integrators to try to use a PIM on a platform for which it was not intended.

Mapping of the executable model semantics to different platforms and languages can introduce subtle variations in behavior. For example, a PIM that assumes a particular sequence of events will work properly when deployed on a single thread of control, but may introduce potential race conditions when deployed in a distributed or multi-threaded environment.

Testing of PIMs requires a systems approach, starting during initial development, mapping onto a platform, integration, and through subsequent reuse. We use a white box integration test approach to verifying PIM behavior in the development environment as well as the target platform. The approach checks the internals of a PIM for correctness as it executes, solving the observability and controllability obstacles. This results in integration tests that are simpler, since they can examine system internals and do not rely on a distinguishing sequence of test inputs to discern a correct state from an incorrect one using the input/output relationship.

The techniques described in this paper aim to make verifying the trustworthiness of PIMs simple for the embedded system developers. The remainder of this paper describes, at a high level, a development and test approach for developing PIMs for use in embedded systems, with an example application of the approach.

Model Driven Architecture

Application of MDA can be broken down into two patterns, interface specification and full behavioral specification.

MDA as an interface specification allows tools that share a common model but different implementation technology to interoperate. For example, in information technology, sales management software may be implemented in Enterprise Java Beans, and must integrate with customer support software implemented with Microsoft .NET. Applying transformations to their common model generates a bridge between the two tools.

Models can also fully specify components and applications behavior, resulting in executable models. While the interface specification is also a part of the behavioral specification, this approach focuses on the behavioral aspect of executable models.

Executable Models

Model-base Software Engineering (MBSE) [3] is a profile of UML that defines the structure of a PIM to create an executable model. The profile combines component, class, and state diagrams with a model level action language to create fully executable platform independent models.

Domains are the logical components of a system. A domain captures the rules and policies of a particular problem space, and creates a well defined boundary between components. A domain is a PIM of a specific problem space. Multiple domains are combined into a system PIM and transformed into the system PSM.

Previous component approaches have focused on architectural separation, but architecture is platform dependent. Domains and logical factoring are the best way to define platform independent components. Within a domain, real world objects and abstractions are represented as UML classes, attributes, and associations. The classes within a domain are highly cohesive and are heavily dependent on one another, whereas the classes are very much independent from classes in other domains.

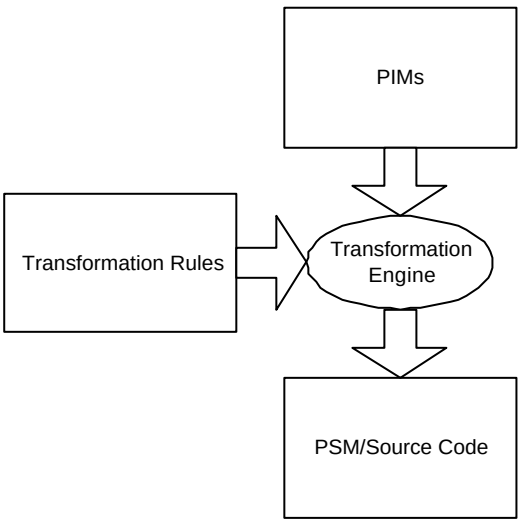
Statecharts and operations capture the coarse grained dynamic behavior, while fine grained behavior is captured in elemental processing units of conditional branches and flow of control, data access including attributes and management of instances, operations, and event generation. While a programming language can be used to capture the fine grained behavior, a model-level, implementation free language conforming to the UML action semantics [4] provides greater freedom when implementing the component for a particular platform. The definition of platform includes the implementation language.

The UML notation and action semantics describe a domain in a way that is complete and executable, but at a high enough level to allow multiple implementation choices. Multiple implementation choices allow the domain to be mapped to multiple platforms. The executable property of domains allows the transformation from one executable form (models) to another (source code), much the same way compilers transform source code into machine code.

Transformations

A transformation engine generates a PSM from the system PIM and the transformation rules, as shown in "PIM Transformation." The transformation rules instruct the transformation engine which transformations to apply to the models to generate the source code. These rules include the formatting of the generated code, the implementation language, and how to implement the model constructs, such as classes and statecharts. Transformation rules also include the platform specific concepts and constraints. Models are mapped onto the platform guided by markings, sets of tags associated with PIM elements.

The PIM contains all the information needed to execute the model. In this case, there is no need for developers to add platform specific code to the models, so there is no need for a UML implementation diagram of the generated source code. The MDA specification includes source code as an acceptable representation of a PSM.



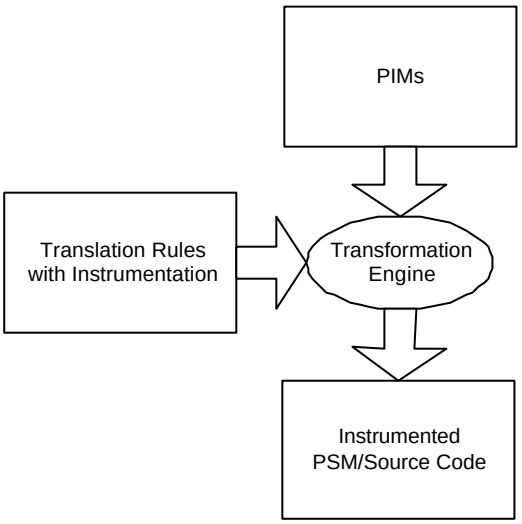
PIM Transformation

Instrumentation

A transformation engine translates the platform independent models into the implementation source code. The transformation rules include rules to generate test instrumentation code similar to a source code compiler adding a symbol table for debugging. Instrumentation can be enabled or disabled at transformation time, compile time, or run time, as required by the application and the phase of testing. The level of instrumentation and what model elements are instrumented can also be configured at transformation time. Instrumentation is largely platform independent and is really itself just another PIM.

The test instrumentation provides greater visibility into the correctness of the internal state than the examination of input-output test results. Errors are detected earlier and easier with visibility into the internal state, and without the need for a distinguishing sequence of test inputs. The instrumentation also automates collection of test results from the software under test. Unless the instrumentation code remains in the final product, the approach is strictly an integration test approach that will verify the PIM and platform specific implementation.

Instrumentation does induce the probe effect– the intrusive nature of some forms of measurement that changes the thing being measured. Probe effect in this case affects the timing of the application. The instrumentation includes the capabilities to minimize probe effect, including managing the sequence of internal and external incidents and control of the system clock as seen by the application.



PIM Transformation with Instrumentation

Model Execution, Debug, and Test

Model execution, test, and debug allow developers and integrators to focus on the correct behavior of the problem to be solved, independent of platform complications, memory management patterns, and other implementation clutter.

The test interface communicates with an external tool set that uses the instrumentation to setup, execute, and verify test cases, as shown in the figure “Test Instrumentation Interface.” These tools can be an interactive model debug environment, an automated test environment, or an application specific environmental emulation. Some of the capabilities provided through the TII interface are setting up and managing the initial instance populations, invoking operations, generating events, reading back instance populations and attribute values, and tracing execution flow.

Interactive Debug

An interactive debug environment at the model level allows developers to monitor and debug the models without getting into platform specific details. Of course, the PSM and/or code remain available if the need arises.

During initial domain development, executable models provide early feedback on requirements. Key scenarios are executed, sequence diagrams generated, and results reviewed with key stakeholders. If the model is being developed to test a new platform, the model testing provides feedback on the platform specific characteristics and benchmarks of the transformation.

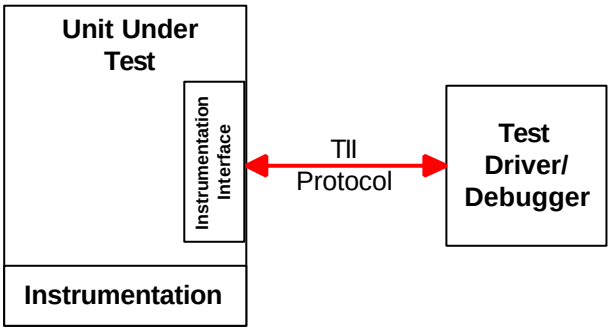
A PIM integrator may want to explore the behavior of a PIM to determine if it is the right component for the application. Again, the testing can happen either in the development environment (with a simulated platform) or on a target platform. The integrator interactively provides test stimulus to the executable model and steps through the execution to understand how it works.

Of course, as development and integration progress, interactive testing and debugging of models will continue, as will automated testing.

Automated Test

An external test driver applies stimulus and controls the sequence of events as seen by the software under test. The test driver emulates some or the entire execution environment. The test driver is a generic test harness that can drive any software application through the test interface.

In addition to the transformation-injected instrumentation, which gives internal visibility outside the application, PIMs can also include self-checks in the form of invariants, assertions, and pre- and post-conditions. Self-checks can be written in action language or OCL if transformation tools are available to translate the expressions into code.



Test Instrumentation Interface

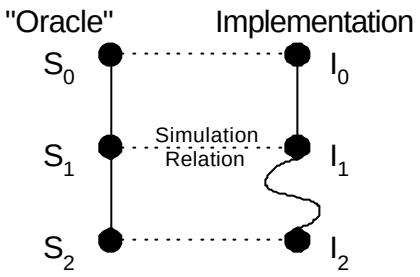
The OMG has issued an RFP for a standard test instrumentation interface (TII) to executable models, “Model-Level Testing and Debugging Interface”[5]. The goal of this RFP is to standardize the hooks into model execution to allow test setup, stimulus, and data collection, as shown in the diagram, “Test Instrumentation Interface”. Many executable modeling tools already have some support for model connection, such as statechart animation.

Failure Recovery Example

Charles Stark Draper Laboratory was built a system that required reliable and rapid failure recovery [6]. The effort resulted in parallel projects, with Team One implementing models in C++ and testing the system by generating system failures at each assembly instruction.

Team Two used UML and action language to model the system and a parallel execution oracle (not the database) to automatically verify the test results [7]. The parallel execution oracle is a requirements level model, expressed in a formal language, in this case, LOTOS [8]. The same inputs were applied to both the oracle and the UML models. The instrumentation was used to verify that the two models took equivalent paths, and at each step in the path, were in equivalent states. A simulation relation [9] was created between the two models that defined the expressions for comparing equivalent states.

The test oracle is constructed from a formal specification at the level of requirements, and is free of design and implementation defects. The simulation relation considers the internal states of the system and the oracle in comparing the results, and not just the input-output relationship. The approach can be used on individual domains within the system rather than the full system itself.



Synchronizing Execution

As shown in the figure "Synchronizing Execution," the specification and the implementation execute parallel. As the implementation executes, the simulation relation finds the corresponding state changes in the oracle's specification, and evaluates the state of each to determine if they are equivalent. If the simulation relation detects non-equivalent internal states, processing is halted, much closer to the source of the defect.

The scope of initial tests focused on single fault recovery, and neither team found any defects in the design. While Team Two resources were diverted, Team One progressed to testing multiple fault testing and found a defect when a fault was injected at a particular point in recovery. This defect was rapidly confirmed by Team One using multi-fault testing on the models.

This project leveraged the design-for-test instrumentation into an automated verification platform to model and verify aspects of a PIM for failure recovery.

Conclusions

MDA holds much promise in modeling at a higher level of abstraction, managing the transformation to target platforms, and creating PIM components. One of the obstacles to comp onent reuse is the "trustworthiness" of the component. If a development team feels that a component does not meet quality criteria, it will not be used.

Simplified PIM testing as described here should lower the perceived effort sufficiently to allow developers to overcome the "not invented here" syndrome and examine components for possible reuse, and also run tests on a trusted component in a new platform or application.

PIM providers must address these issues if they hope to encourage developers to reuse the components. If the component comes with its own set of test cases and the ability to do result validation for new test cases, developers would be much more likely to trust and reuse the component. PIMs should be packaged with platform independent test cases to allow the PIM to be tested on new platforms. Oracles and self test packaged with a PIM can make testing in a new environment or platform easier and produce higher quality systems built from component PIMs. Tool vendors providing platform specific mappings should also package test models for the platforms.

Instrumentation added to the PIM implementation (PSM) through transformation helps ease the test problem by increasing testability, observability, and controllability. However, it is a white-box integration test approach and not applicable to system or validation testing.

References

[1] Object Management Group, MDA Guide Version 1.0.1, Document Number: omg/2003-06-01, 2003.

[2] Elaine Weyuker, "Testing Issues for Component-Based Software - A Cautionary Tale," IEEE/Software, Sept.-Oct, 1998.

[3] Peter J. Fontana, "Model Based Software Engineering", white paper, Pathfinder Solutions, http://www.pathfindersol.com/, 2000.

[4] Object Management Group, OMG Unified Modeling Language Specification (Action Semantics), Document Number: ptc/02-01-09, 2001.

[5] Object Management Group, Model-level Testing and Debugging RFP, Document Number: realtime/03-01-12, 2001.

[6] Dianne Turney, Deborah Allinger, Robert Breton, Gregory T. Eakman, "Improved Testing of Real-Time Object-Oriented Systems", 21st Digital Avionics Systems Conference (DASC), Irvine, CA, 2002.

[7] Eakman, Gregory T., A Systems Approach to Automated Object-Oriented Integration Testing, Ph.D. Dissertation, Boston University, 2002.

[8] ISO/IEC. "LOTOS A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour". International Standard 8807, International Organization for Standardization Information Processing Systems, Open Systems Interconnection, Geneva, September, 1988.

[9] Nancy A. Lynch, Distributed Algorithms , Morgan Kaufman Publishers, Inc. San Francisco, 1996.