Integration and testing of software is difficult, and embedded systems provide the additional challenges of limited manipulation and visibility of the system through a small number of inputs and outputs. Abnormal system states, in particular, are difficult to test, because the system must be driven into the state before its behavior in that state can be determined.

This article introduces the idea of instrumentation code injected into the implementation of UML models for the purposes of increasing the controllability, observability, and testability of the system. The instrumentation is used in both the development and the target environments, and allows interactive system debugging at the model level. In batch mode, the instrumentation serves as the basis for data collection, initialization, and test automation. My goals are to:

- Provide a brief overview of model-based software engineering and implementation of these models
- Outline approaches for integration testing of model-based software
- Identify the interesting run-time data and execution points within modeled systems
- Define alternatives for collecting and manipulating model data at runtime
- Integrate the instrumentation with test automation

Integration testing
According to Roger S. Pressman, in *Software Engineering—A Practitioner’s Approach*, “Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover the errors associated with
interfacing. UML models and object-oriented software tend to have classes with many complex interactions, which hinder integration testing. Combining a structured approach to UML analysis modeling with a coherent integration and test strategy will make developing quality embedded systems easier.

A software fault is an erroneous instruction or computation within a program. Execution of that fault results in an error in the state of the software. When the error is propagated to the output, and becomes visible outside the system as an unexpected result, a failure has occurred. Controllability of a program is the ability of a suite of test cases to force the program under test to follow a particular execution path, possibly executing faults along the way. Observability of a program is the ability of the test suite to detect an error state, and thereby illuminate the existence of a fault.

The internal state of the system is important in determining the correctness of tests. The output of a system is dependent upon both the initial state of the system and the inputs applied to it. The same set of inputs applied to a different initial state will result in different outputs. The final state of the system must also be examined as part of evaluating the correctness of the test, as an incorrect internal state may eventually propagate to the system output, causing a failure. System complexity may also make it difficult to predict the correct outputs of the system.

Initial State + Inputs --> Final State + Outputs

Using black box test techniques, only the external inputs and outputs of the system are available. A distinguishing sequence of test stimuli is required to propagate an error to the output so as to distinguish a faulty program from a correct one. The longer the required distinguishing sequence, the less testable the program. Embedded systems are similar to black boxes in that controllability and observability are usually limited. Evaluating the final internal state of the system results in a test harness to greatly improve controllability and observability.

Phases of integration testing
Integration testing is broken into two main phases, dynamic verification and target integration. Dynamic verification is the execution of UML models in the development environment. It focuses on determining the correctness of the models. Target integration involves software and hardware integration in the target environment. Both dynamic verification and target integration are done at the analysis level, with the same tools, using the test support instrumentation.

There are many reasons to do as much dynamic verification testing as possible: hardware availability, hardware/software isolation, shorter debugging cycle times, and access to tools. If you have high confidence in your mod-
After running tests in dynamic verification, debugging in target integration can focus more heavily on the interfaces between system components and on platform-specific issues.

**Modeling embedded systems with UML**

The effective application of UML models to software engineering for challenging applications—especially in the embedded context—requires a development process that will ensure:

- Models are rigorous and complete
- The resulting system implementation can be optimized without impacting the models
- The overall architecture of the system is maintained by the process through multiple releases and requirement evolution

To achieve these goals, model-based software engineering employs a translational approach, defined below. This article focuses on adding test support into code using a translational approach, but the techniques can also be applied to manually implemented UML models. Specific aspects of this type of the translational process are introduced in the following section.

**Analysis**

The process of modeling an implementation-independent solution to a problem in terms of the problem itself is called analysis. Effective analysis models are rigorous and complete, and largely free of implementation bias. The Unified Modeling Language (UML) is a standard notation defined by the OMG for expressing analysis models. The work products produced during analysis are:

- **Domain model**: this is a UML class diagram showing the highest level decomposition of the system into areas of separate subject matter, called domains. These domains are represented as packages, and dependency arrows show bridges, which are
the flow of requirements between domains. A domain can be analyzed, or it can be developed via other means, such as hand-written code, legacy code, generated from another source, imported from a library, and so on. Domain services are methods that make up the interface of the domain. Since the domains define a complete specification of a single problem space, they can be independently tested, then combined with other domains for further testing.

- **Information model**: for each domain that is to be analyzed, a UML class diagram is used to define the classes that form the structure of the domain. Classes have associations with other classes, and inherit from other classes.
- **Scenario model**: key scenarios for this specific domain are captured with UML sequence charts and/or UML collaboration diagrams to show interactions between domain services (operations), class services (methods), class events messages, and services of outside domains used in this domain.
- **State model**: for each class that receives event messages, a UML state diagram is used to capture the class lifecycle, defining state-dependent behavior for that class.
- **Action model**: for each domain service, class service, and state action, a detailed, unambiguous behavioral description is created. This is expressed in an action language, an analysis-level “programming” language that provides a complete set of analysis-level execution primitives without biasing the implementation. By expressing behavioral detail in action language, considerable freedom is retained until the translation phase for how each analysis primitive is implemented, which is critical for optimization.

**Design**

Design is the creation of a strategy and mechanisms supporting the mapping of analysis constructs to a run-time environment. Design is conducted in a different concept space from analysis, and much of the preliminary design work can be completed independent of the analysis activities.

**Translation**

Translation is the process in which the UML models for each analyzed domain are mapped to implementation through design strategies. Design is conducted at two levels:

- **Structural design**: identify the execution units (threads/tasks/processes) of the system, allocate them to processors, and allocate domains to the units.
- **Mechanical design**: develop detailed patterns (expressed in templates) to map analysis to implementation and build base mechanisms to support this implementation.

**Instrumentation**

**Parallels to source code debugger**

Since the UML models represent a complete executable model of the system, the models can be translated into implementation automatically. A set of translation rules is applied to the model, much like a compiler translates high-level programming languages.

Following the language and model compiler analogy further, a model compiler can add instrumentation into the generated code, just as a language compiler adds a symbol table and debug information into the executable. Instrumentation from both compilers allows the resulting application to be tested and debugged by the developer at the same level of abstraction as it was developed. Only in very rare cases would a high-level language developer want to look at the assembly or machine code when debugging an application. Similarly, for UML models, the developer will want to debug at the higher level of abstraction of the models, rather than with the implementation code.

The translational approach uses information from the UML models to create code to support testing in addition to the application code. The instrumentation does not add any additional functionality to the software, other than enhanced testability. The test instrumentation is only available for test support and cannot be used during the normal operation of the software.

Because the instrumentation injected during model translation is based only on the UML model execution semantics, it provides a generic test harness that can be applied to any application. The instrumentation can be compiled out, or the code regenerated without the instrumentation, similar to the way debugging information is handled by the compiler.

For manually implemented systems, the level of instrumentation required is dependent on the complexity of the application, the test approach, the target environment, available memory, the support of other tools, and the time available. Thus, trade-offs must be made in order to deliver a quality product on time. In the UML models, one must identify important data values, attributes, inputs, and control points. For each of these items, add the appropriate instrumentation access. The remainder of this article will describe adding instrumentation using the translational approach, but the same principles can be applied to manual implementation.

**Instrumented application architecture**

The UML test architecture is broken up into two components, the dynamic verification user interface (DVUI) and the instrumentation agent. The DVUI is responsible for displaying information to the user and accepting user commands. The DVUI could be replaced with a batch interface for automation.

The instrumentation agent acts as the interface between the application and the DVUI. The communication mechanism between the agent and DVUI can be any protocol—TCP/IP, RS-232, and so on. The agent supports information marshaling and communication via generated instrumentation code. It interfaces with the instrumentation to set and retrieve instance.
Data and to support DVUI notification of break and trace points, and interfaces with the event processing to provide execution control, system stimulus, break points, and stepping.

Data instrumentation code is injected into the application during translation to monitor and update object instance population, attribute values, event queue population, event data item values, and service parameters.

**Data access**

During integration testing and debugging, assessing the system state is a key requirement. Since the system state is distributed among many class instances, all instances in the system as well as the values of the attributes should be accessible to the DVUI.

Keeping a list of instances in the system is fairly straightforward. Each class must include some structure to hold the instances of that class, commonly some form of a linked list. Within the constructor, the new instance is added to the list, and in the destructor, it is removed.

To view the state of an instance, the instance can support ASCII serialization, similar to the Java `toString()` method. The `toString()` method would put the value of each of the instance’s attributes into the string, as well as the instances with which it has associations. The string can then be transported from the application agent to the DVUI for display. To set the data of an instance, a `fromString()` method must be supported that can unpack the data and make the proper conversions and attribute assignments.

The state of an instance’s state machine is also very interesting during debugging, because it represents an aspect of the control flow of the system. It is special because it affects the response of an instance to an event. Therefore, it’s useful to separate the current state of the state machine from the rest of the class attributes.

**Dynamic behavior**

During system execution, instances are created and deleted, and events are exchanged; associations are created and deleted; domain services are invoked; and timers are set and fired. Each of these incidents is of potential interest during integration testing. Depending on the implementation, each of these incidents could trigger an interactive break point or a trace output.

The DVUI should be able to send to the agent the conditions on which to break. The break allows the DVUI to interactively browse the state of the system at that point in execution and adjust instances and attributes, if necessary. Trace points can also be set up by the DVUI. At a trace point, a mes-
sage describing the triggering incident is passed to the DVUI and stored in a log file. Trace points are a great way to follow system execution at a high level. They can even be post-processed to generate a sequence diagram describing the scenario executed, or they can be used in regression testing.

Break and trace point controls can be implemented using the publish-subscribe pattern. The agent supports registration of each type of application execution incident. The DVUI can then subscribe to incidents, such as the creation of a particular instance or initiation of a state machine transition. Instrumentation within the application code notifies the agent when an incident occurs, and the agent then notifies the subscriber with the appropriate action, break, or trace.

One implementation of a system modeled with communicating state machines includes an event queue. Events sent by objects and from outside the domain are placed on the queue. An event loop executes continuously, pulling the next event off the queue and passing it to its destination instance, which executes the specified action. With this implementation, the event loop serves as a central place to monitor system execution.

Within the event loop, an instrumentation interface is added that watches the events going by and looks up the next state to be executed by the receiving instance. These conditions can be compared against the set of break and trace conditions set by the DVUI.

The instrumentation in the event loop could also support a single event step mode. In this mode, the DVUI would command the application to execute one action, but stop before the next.

If an analysis-level action language is used to describe the UML model actions and services, you can expand the idea of event stepping to stepping of action language by instrumenting the code in a similar way. Every line of action language would be preceded by an instrumentation instruction, keeping track of the line number and local variable values. Of course, this would be tedious and error prone if done manually, but is easily accomplished by translation.

**Test execution**

This section describes different ways to use the instrumentation in the application to make testing and fault isolation easier.

**Initialization**

The instrumentation can be used to set the system to a known initial state. By creating and initializing class instances through the agent, the state can be set directly by the test case. This makes testing from hard-to-reach states easier, rather than having to apply a sequence of inputs to drive it to that state. The initialization is made possible by serializing instance data across the communications channel to the agent, where the new instance is created and initialized.

A soft reset capability would allow the system to be cleared, reinitialized with another initial state, and run with another set of test cases.

**Stimulus**

Stimulus is applied to the system from the DVUI through the instrumentation agent. Events can be initiated by the DVUI and delivered to a target instance through the instrumentation agent and event queue. An encoding scheme for the event and the data must be derived. Also, with some additional encoding, domain services can be invoked as system stimuli as well. This mechanism is similar to the encoding of parameters that CORBA and RPC use when calling functions across processes.

One of the more difficult problems in embedded systems tends to be reproducibility of failures due to timing or sequencing of events. By controlling the sequencing of event transmission through the event queue, orderings of actions can be tested and reproduced. Of course, the interface to the event queue would need to allow the reordering of events, in addition to viewing the events.

**Data collection**

The data collection interface can be used to test pre- and post-conditions of test cases. This is especially useful for determining and validating the final state of the system as part of test result evaluation.

Another side benefit of instrumentation is the ability to capture true target stimuli and replay it in the dynamic verification environment. If some part of the instrumentation were left in the finished product, it could record a subset of the system state and inputs, much like the “black box” on airplanes.

**Emulation**

In order to effectively test a domain in isolation during dynamic verification, the interface at the domain boundary must be well understood. The test cases that are defined for the domain will generally use this interface as the primary stimulus to the domain. The test cases and stimulus data are obviously application-specific, but use the test harness already provided by the instrumentation.

Figure 1 shows a test driver, either the DVUI or other program connected to the instrumentation agent, emulating the domain’s target environment. The driver initializes the class instances that are part of the test. The test driver applies the test stimuli and captures the response. The test driver also emulates the responses of other domains by trapping the service calls and substituting the return values. The calls are a form of output from the point of view of the domain under test, and the responses provided by the test driver provide more input to the domain under test. The test driver uses the test instrumentation in the domain under test to trap and substitute messages.

**Single domain to system testing**

This test approach is scalable from one domain to the integration of multiple domains and into system test (Figure 2). Single domains are first tested in isolation, using the test driver to emulate the environment of the domain under test. Domains can then be combined for testing through integration of domain services. Again, the test driver emulates the environment of the domains under test. The assumptions that one domain
makes about another, in the form of service calls, can now be verified. The interface and flow of data across the test boundary should be well understood.

As confidence is gained in lower level domains, the test support instrumentation can be disabled on those domains to reduce the number of checkpoints and increase throughput. If a problem is found, the test support can be re-enabled to collect more data on the specific test case.

**Leveraging potential**

Instrumentation added to the implementation of UML models allows the models to be leveraged into the testing phase of development. Developers can write test cases, execute, and debug and the level of the models. Ideally, instrumentation should provide a full debug environment at the level of analysis. Full interactive manipulation of instances, events, and data should be available, as well as ways to actively stimulate the system, rather than just watch it. The instrumentation provides full access to the system under test at the level of UML modeling, allowing a glass box approach to integration testing with greater observability, controllability, and testability for embedded systems. Increased observability enabled by instrumentation also allows detection of internal error states without having to propagate the error to the output. This results in easier test case development and shorter debugging time, as the error is isolated much closer to the fault.

Once the capability of the full interactive analysis debugger exists, there is an even greater potential for improved quality and productivity by adding a batch capability to it. This batch capability would allow automated regression testing, automated data collection of future test cases, or application of randomized test inputs and event sequences to obtain more analysis coverage.

---

**References**


---

*Gregory Eakman is a principle consultant with Pathfinder Solutions and is a PhD candidate at Boston University in the area of automated testing of model-based software. He has successfully applied model-based software engineering to a variety of applications. Contact him at greg@pathfindersol.com.*