



Testing Embedded Systems: Do You Have The GuTs for It?

by [Vincent Encontre](#)

Director, Embedded and Real-Time
Automated Testing Business Unit
Rational Software -- Toulouse, France

This article offers a general introduction to testing embedded systems, including a discussion of how embedded systems issues affect testing process and technologies, and how Rational Test RealTime provides solutions for these issues. Many thanks to Paul Szymkowiak, Rational's process engineer involved in the testing portion of the Rational Unified Process® (RUP®), for his improvements to my original text, for his continuous support, and for his glossary of real-time embedded terminology at the conclusion of this article.

In order to settle on a common set of concepts, let's start with some definitions.

What is testing? Testing is a disciplined process that consists of evaluating the application (including its components) behavior, performance, and robustness -- usually against expected criteria. One of the main criteria, although usually implicit, is to be as defect-free as possible. Expected behavior, performance, and robustness should therefore be both formally described and measurable. Debugging literally means removing defects ("bugs"), and is thus not considered part of the testing process directly. Testing is one of the *detective* measures, and debugging one of the *corrective* measures of quality.

What exactly is an "embedded system"? It's difficult, and highly controversial, to give a precise definition. So, here are some examples. Embedded systems are in every "intelligent" device that is infiltrating our daily lives: the cell phone in your pocket, and all the wireless infrastructure behind it; the Palm Pilot on your desk; the Internet router your e-mails are channeled through; your big-screen home theater system; the air traffic control station as well as the delayed aircraft it is monitoring! Software now makes up 90 percent of the value of these devices.

- [▶ subscribe](#)
- [▶ contact us](#)
- [▶ submit an article](#)
- [▶ rational.com](#)
- [▶ issue contents](#)
- [▶ archives](#)
- [▶ mission statement](#)
- [▶ editorial staff](#)



Figure 1: The World Runs on Embedded Software

Most, if not all, embedded systems are "real-time" (the terms "real-time" and "embedded" are often used interchangeably). A real-time system is one in which the correctness of a computation not only depends on its logical correctness, but also on the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred. And for some systems, identified as safety-critical, failure is not an option.

Thus testing timing constraints is as important as testing functional behavior for an embedded system.

Even though the embedded systems testing process resembles that used for many other kinds of applications, it is impacted by some issues important to the embedded world:

- Separation between the application development and execution platforms
- A large variety of execution platforms and thus of cross-development environments
- A wide range of deployment architectures
- Coexistence of various implementation paradigms
- Tight resources and timing constraints on the execution platform
- Lack of clear design models
- Emerging quality and certification standards

A more detailed discussion of these issues is found near the end of this article.

These issues greatly affect testability and measurability of an embedded

system. This explains why testing such systems is so difficult and consequently is one of the weakest points of current development practices. So it is no wonder that, according to a recent study: 50 percent plus of embedded systems development projects are months behind schedule and only 44 percent of designs are within 20 percent of feature and performance expectations, and all this even when 50 percent plus of total development effort is spent in testing.

In the rest of this article, we will:

- Go through a generic test iteration from which we will derive a minimal set of desired testing technology.
- Instantiate this iteration to address the testing of complex systems as found in the embedded world; based on these considerations we will add capabilities to our ideal testing technology.
- Moving one step further, examine what makes embedded systems so difficult to develop and test, and assess how these issues add to the list of features that need to be fulfilled by the technology used to test them. Rational Test RealTime will be used as an example of a product that implements a large portion of this ideal technology.

A Generic Test Iteration



The first step we will consider is to identify the granule to be tested. I use the word *granule* to avoid other words such as component, unit, or system, all of which have less generic definitions. For example, in the Unified Modeling Language (UML) a component is a piece of an application that has its own thread of control (that is, a task or a Unix process). I also like the word granule as the root for granularity, which translates well over the wide range of elements that can be tested: from a single line of code up to a large distributed system.

Preparing the Granule Under Test

So, first identify the granule to test, then transform it to a testable granule, or [granule under test](#) (GuT). This step consists of isolating the granule from its environment by making it independent with the help of *stubs* and, sometimes, *adapters*. A stub is a piece of code that simulates two-way access between the granule and another part of the application. Next, build a test driver. This will stimulate the GuT with the appropriate input, measure output information, and compare it with the expected response. An adapter is sometimes used to allow the GuT to be stimulated by the test driver. Stimulation and measurement follow specific paths through gates in the GuT: we will call them [Points of Control and Observation](#) (PCOs), a term that comes directly from the

telecommunications industry. A PCO can be located at the border of, or inside, the GuT. Examples of PCOs for a C function granule are:

- Point of Observation inside the granule: coverage of a specific line of code in the function.
- Point of Observation at the border of the granule: parameter values returned by the function.
- Point of Control inside the granule: change of a local variable.
- Point of Control at the border of the granule: the function call with actual parameters.

Stubs and even test drivers can be other parts of the application (if they are available) and don't necessarily need to be developed for testing a C function or a C++ class. The GuT can be accessed or stimulated through another part of the application, which then plays the role of stub or test driver. Stubs and test drivers constitute the test harness environment of the GuT.

Defining the Test Case

Defining a test case is a matter of figuring out:

- The appropriate PCOs: this depends on the kind of testing to be performed, such as functional, structural, load, and so on
- How to exploit these PCOs: which information to send and to expect to receive through them, and in what order, if any

The type of test as well as sent/expected information is driven by the requirement set for the GuT. In the case of safety-critical systems, formal and precise requirements are an essential part of the development process. While formal requirements are an important motivation source for tests, they don't always explicitly identify the tests that will discover important flaws in the system. Using the formal requirements, and for less critical applications when specific requirements are lacking, the tester must consider an appropriate set of tests to conduct (also referred to as "test ideas") to identify some requirements for testing the GuT. These requirements can then be translated into concrete test cases that take advantage of available PCOs. By "concrete," we primarily mean *executable*.

Usually, requirements themselves are not formal and do not naturally translate into formal test cases. This translation process often introduces errors in which test cases do not accurately reflect requirements. Specification languages have become more formal since the introduction of UML, and it has now become possible to express formal requirement-based test cases to avoid the translation pitfall. Rational QualityArchitect and a part of Rational Test RealTime provide good examples of using such model-based testing techniques.

Unfortunately, not all requirements are described using UML: in the

embedded world, the most common formal description technique for a test case is simply to use a programming language such as C or C++. While C or C++ are universally known (which reduces the learning curve), they don't do a good job of taking into account test case needs such as PCO definition or expected GuT behavior. As such they make it difficult for someone to define and implement comprehensive, formal test cases. This problem has been addressed by the design of specific high-level testing languages, which are well adapted to specific testing domains such as data-intensive or transaction-based testing. Rational Test RealTime proposes a mix of native 3GL and dedicated high-level scripting languages, bringing the best of both worlds: reduced learning curve and efficiency in defining and implementing test cases.

Another extremely valuable and productive way to implement one or more test cases is to use session recorders: While the GuT is stimulated (either manually or by its future environment), particular Points of Observation record information passing in and out of the GuT. This information is automatically post-processed and translated into an appropriate case to be replayed later. An example of such a session recorder is found in Rational Rose RealTime where model execution leads to the generation of a UML sequence diagram reflecting trace execution, which is then used as input to Rational QualityArchitect RealTime.

Each test implementation must bring the GuT to a particular starting state that allows the test to be run. This part of a test is known as the *preamble*. Symmetrically, at the end of the effective test and whatever the result, the test implementation must bring the GuT to a final stable state that allows the next test to execute. This part of the test is called the *postamble*.

Deploying and Executing the Test

Once implemented, the test case is then integrated with the test driver and stubs. It is important to note that stub definition and implementation is an integral part of the test, enabling the test case to be realized. The test case forms the informational (vs. the operational) part of the test, and is validated during test harness execution.

Observing Test Results

Results from test execution are monitored through Points of Observation.

At the border of the granule, typical types of Points of Observation include:

- *Parameters returned by functions or received messages.*
- *Value of global variables.*
- *Ordering and timing of information.*

Inside the granule, typical types of Points of Observation include:

- *Source code coverage*, providing details about which software part of the GuT has been covered.
- *Information flow* to visualize exchange of information with respect to time between the different parts of the GuT. Typically this kind of flow is represented as a UML sequence diagram as in Rational Test RealTime.
- *Resource usage* showing nonfunctional information such as time spent in the various parts of the GuT, memory pool management (as exemplified in the Rational PurifyPlus family), or event handling performances.

All these observations can be collected for a single test case and/or aggregated for a set of test cases.

Deciding on Next Steps

Once all test output data have been gathered and test results synthesized, there can be one of two outcomes: One or more of the test cases failed, or all of the tests passed.

Test cases can fail for a number of reasons:

- *Nonconformance of the GuT to the requirements.* You'll have to go back to the implementation -- or worse, to the design -- to fix the problem in the GuT. This includes fixing implicit requirements such as "zero-crash" reliability.
- *The test case is wrong.* This happens much more frequently than you might think. Tests, like software, do not always work as expected the first time they are executed. Modify the test case definition and/or corresponding implementation to fix the problem.
- *Test case implementation cannot be executed.* Again, like software, everything seems correct, but you cannot deploy or start or connect your test harness to the GuT. You'll need to analyze the failure using debug and diagnostic tools.

If all tests have passed, you might want to consider these courses of action:

- *Reevaluate your test.* If you are early in your test process, you might question the value and goal of your test. Tests are supposed to find problems (especially early in the development effort), and if you don't get any, well...
- *Increase the number of test cases.* This should increase visibility of, and confidence in, the reliability of the GuT. Some may object that reliability is part of requirements, and they are correct. However, level of reliability is often directly correlated to the level

of coverage of the GuT by the overall set of test cases.

Some may object that each necessary reliability test should correspond to a formally stated requirement, and they are arguably correct. The problem is that some reliability requirements are implicit, which means there won't be formally stated requirements for them. Furthermore, it has been observed that the actual reliability of the GuT is often directly correlated to the level of coverage of the GuT by the overall set of test cases: A sufficient number of tests is necessary to validate reliability.

Arguably, the most widely-used type of coverage is code coverage, as implemented in Rational Test RealTime. While not a complete approach to test coverage by itself, basing our testing on code coverage helps to define additional test cases that will increase the coverage up to a level agreed in the requirements. This part of testing is often referred to as structural testing: Test cases are based on the content of the GuT, not directly on its requirements.

- *Increase the scope of the test by aggregating granules.* You'll then apply this generic process to a larger portion of your system, as described in the next paragraph.

When to Stop Testing?

This is a perennial question for the software practitioner, and it is not the ambition of this article to solve it! However, here is one heuristic that can be used: Consider the safety criticality of the system under test. Can the system be deemed safety critical or not? For nonsafety-critical systems, testing can be stopped based on more or less subjective criteria such as time-to-market, budget, and a somewhat reactive assessment of "good-enough" quality. However, for safety-critical systems, for which failure is not an option, the decision to stop testing cannot afford to be made on subjective criteria. For these systems, the good-enough "quality bar" remains very high. We will see in the "Emerging Quality and Certification Standards" section near the end of this article some recommendations dealing with this issue.

Requirements for a Generic Testing Technology

From the generic test iteration previously described, we can infer a minimal set of features that must be fulfilled by testing tools. They must:

- Help to define and isolate the GuT.
- Provide a test case notation, either 3GL or visual or high-level scripting, supporting definition for PCOs, information sent to and expected from the GuT, and preamble/postamble.
- Help to accurately derive test cases from requirements or test ideas.

- Provide alternative ways to implement test cases using session recorders.
- Support test case deployment and execution.
- Report observations.
- Assess success or analyze failure.

Needless to say, Rational Test RealTime supports these features. But Rational Test RealTime is also going beyond these requirements to address the testing of complex systems as found in the embedded systems domain.

Generic Architecture and Implementation of Complex Embedded Systems

Embedded systems are complex systems that can be composed of extremely diverse architectures ranging from tiny 8-bit micro-controllers up to large distributed systems made of multi-processor platforms. However, two-thirds of these systems run on a real-time operating system (RTOS), either commercial off-the-shelf or in-house, and implement the concept of threads (a thread is a granule with an independent flow of control) that extend to the RTOS task or process. In the UML, this concept is referred to as a *Component*, while a *node* refers to an independent processing unit running a set of tasks managed by an RTOS. Any communication between nodes is usually performed using message-passing protocols such as TCP/IP.

The vast majority of developers of embedded systems use C, C++, Ada or Java as programming languages (70 percent will be using C in 2002, 60 percent C++, 20 percent Java, 5 percent Ada). It is not unusual to see more than one language in an embedded system, in particular C and C++ together, or C and Java. C is supposed to be more efficient and closer to the platform's details, while Java or C++ are (supposedly) more productive thanks to object-oriented concepts. However, it should be noted that embedded systems programmers are not object devotees!

In the context of embedded systems, a granule can be one the following (sorted by incremental complexity):

- C function or Ada procedure
- C++ or Java class
- C or Ada (set of) module(s)
- C++ or Java cluster of classes
- an RTOS task
- a node
- the complete system

For the smallest embedded systems, the complete system is composed of a set of C modules only and doesn't integrate any RTOS-related code. For the largest ones (distributed systems), networking protocols add another level of complexity.

The next section will show how this generic architecture impacts various aspects of the test effort.

Six Aspects of Testing Complex Embedded Systems

Depending on the type of the granule, and according to *common* usage in the industry (we will discuss this usage later in this section), it is common to consider testing from six different aspects to evaluate whether the application's behavior, performance, and robustness match expected criteria. These aspects are:

1. Software unit testing
2. Software integration testing
3. Software validation testing
4. System unit testing
5. System integration testing
6. System validation testing

What follows is a discussion of each of these six aspects in relation to testing complex embedded systems.

1. Software Unit Testing

The GuT is either an isolated C function or a C++ class. Depending on the purpose of the GuT, the test case consists of either:

- *Data-intensive testing*: applying a large range of data variation for function parameter values, or
- *Scenario-based testing*: exercising different C++ method invocation sequences to perform all possible use cases as found in the requirements.

Points of Observation are returned value parameters, object property assessments, and source code coverage. White-box testing is used for testing units, meaning that the tester must be familiar with the content of the GuT. Unit testing is thus the responsibility of the developer.

Since it is not easy to track down trivial errors in a complex embedded system, every effort should be made to locate and remove them at the unit-test level.

2. Software Integration Testing

The GuT is now a set of functions or a cluster of classes. The essence of integration testing is the validation of the interface. The same type of Points of Control applies as for unit testing (data-intensive

main function call or method-invocation sequences), while Points of Observation focus on interactions between lower-level granules using information flow diagrams.

As soon as the GuT starts to be meaningful, that is when an end-to-end test scenario can be applied to the GuT. First, performance tests can be run that should provide a good indication about the validity of the architecture. As for functional testing, the earlier the better. Each forthcoming step will then include performance testing. White-box testing is also the method used during that step. Software integration testing is the responsibility of the developer.

3. Software Validation Testing

The GuT is all the user code inside a component. This can be considered one of the activities that occur toward the end of each software integration or build cycle. Partial use-case instances -- also called partial scenarios -- begin to drive the test implementation. The test implementation is less aware of and influenced by the implementation details of the GuT. Points of Observation include resource usage evaluation since the GuT is a significant part of the overall system. Again (and finally), we consider this step as white-box testing. Software validation testing is still the responsibility of the developer.

4. System Unit Testing

The GuT is now a full system component -- that is, the user code as tested during software validation testing plus all RTOS- and platform-related pieces: tasking mechanisms, communications, interrupts, and so on. The Point of Control protocol is no longer a call to a function or a method invocation, but rather a message sent/received using the RTOS message queues, for example.

The similarities in their message-passing paradigms implies that the distinction between test drivers and stubs can, from the perspective of the GuT, be considered irrelevant at this stage. We will call them Virtual Testers because each one can replace and act as another system component vis-à-vis the GuT. "Simulator" or "tester" are synonyms for "virtual tester." Virtual tester technology should be versatile enough to adapt to a large number of RTOS and networking protocols. From now on, test scripts usually: bring the GuT into the desired initial state; then generate ordered sequences of samples of messages; and validate messages received by comparing (1) message content against expected messages and (2) date of reception against timing constraints. The test script is distributed and deployed over the various virtual testers. System resources are monitored to assess the system's ability to sustain embedded system execution. For



this aspect, grey-box testing is the preferred testing method. In most cases, only a knowledge of the interface (the API) to the GuT is required to implement and execute appropriate tests. Depending on the organization, system unit testing is either the responsibility of the developer or of a dedicated system integration team.

5. System Integration Testing

The GuT starts from a set of components within a single node and eventually encompasses all system nodes up to a set of distributed nodes. The PCOs are a mix of RTOS- and network-related communication protocols, such as RTOS events and network messages. In addition to a component, a Virtual Tester can also play the role of a node. As for software integration, the focus is on validating the various interfaces. Grey-box testing is the preferred testing method. System integration testing is typically the responsibility of the system integration team.

6. System Validation Testing

The GuT is now a complete implementation subsystem or the complete embedded system. The objectives of this final aspect are several:

- *Meet external-actor functional requirements.* Note that an external-actor might either be a device in a telecom network (say if our embedded system is an Internet Router), or a person (if the system is a consumer device), or both (an Internet Router that can be administered by an end user).
- *Perform final non-functional testing* such as load and robustness testing. Virtual testers can be duplicated to simulate load, and be programmed to generate failures in the system.
- *Ensure interoperability with other connected equipment.* Check conformance to applicable interconnection standards.

Going into details for these objectives is not in the scope of this article. Black-box testing is the preferred method: The tester typically concentrates on both frequently used and potentially risky or dangerous use-case instances.

Deciding How to Address These Aspects

Now that we have defined these six aspects, how do we use them? There are plenty of criteria that are used to decide:

- Whether all these aspects apply to your systems.
- Whether all the aspects should be performed for all or only some parts of the systems.
- The order in which the aspects should be applied to the selected parts.

The answers to these questions depend heavily on the kind of embedded system you are developing: safety-critical or not, time-to-market, few or millions deployed, and so on. A future article will provide guidelines to help you with this selection process.

Another way to address these testing aspects is to melt them into one. Validation testing can be considered as unit testing a larger GuT. Integration can also be checked during validation testing. It is just a matter of how you can access the GuT, which kind of PCOs you can insert and where, and how you can target a specific portion of the GuT (possibly very remote) from the test case. But let's also leave that to another discussion.

Additional Requirements for a Complex Systems Testing Technology

In order to address the challenge of testing complex embedded systems, the testing technology must add the following capabilities:

- *Manage multiple types of Points of Control* in order to stimulate the GuT. Do this using: function call; method invocation; and message passing or RPC through different language bindings such as Ada, C, C++ or Java.
- *Offer a wide variety of Points of Observation* such as: parameters and global variable inspections; assertion, information, and control path tracing; and code coverage recording or resource usage monitoring. Each of these Points of Observation should provide expected vs. actual assessment capabilities.

How Embedded Systems Issues Affect Testing Process and Technology

In this section, we will highlight the specific issues of embedded systems and assess how they affect the technology used to test them. We will use Rational Test RealTime as our example testing tool.

Separation Between the Application Development and Execution Platforms

One of the multiple definitions for an embedded system is the following: *An embedded system is any software system that must be designed on a platform different from the platform on which the application is intended to be deployed and targeted.* By platform, on the development side, one typically means an operating system such as Windows, Solaris, HP-UX, or Linux. It should be noted that the percentage of Unix and Linux users is much higher (40 percent ¹) in the embedded systems domain as compared to other IT systems domains. For the target, platforms include any of the devices mentioned earlier. Why the constraint? Because the target platform is optimized and tailored exclusively for the end-user (a real person or another set of devices), it

will not have the necessary components (such as keyboards, networking, disks, and so on) for the development to be carried out.

To cope with this dual platform issue, the testing tool must provide access to the execution platform from the development platform in the most transparent but efficient way possible. In fact, the complexity of such access must be hidden to the user. Access includes test-case information download, test execution remote monitoring (start, synchronize, stop), and test results and observation upload. In Rational Test RealTime, all target platform accesses are controlled by the Target Deployment technology.

In addition, Rational Test RealTime is available on Windows, Solaris, HP-UX, and Linux, the development platforms that leading companies in the device, embedded systems, and infrastructure industries are using.

A Large and Growing Variety of Execution Platforms and Cross-Development Environments

The execution platform can range from a tiny 8-bit micro-controller-based board to a large distributed and networked system. All platforms need different tools for application development due to the profusion of chip and system vendors. It is increasingly common that multiple platforms are used within the same embedded system. From a development perspective, this kind of environment is referred to as a "cross-development environment."

The large variety of execution platforms implies the availability of a correspondingly large set of development tools such as compilers, linkers, loaders, and debuggers.

The first consequence of this is that Points of Observation technology can only be source-code based. As opposed to Object Code Insertion technology used by the Rational PurifyPlus family and available for a small set of native compilers, Rational Test RealTime uses source-code instrumentation to cope with the number factor. Of course, ten years of experience in this technology has resulted in highly efficient code instrumentation.

Another direct consequence of this variety is that cross-development tool vendors tend to offer Integrated Development Environments (IDEs) to hide complexity and make the developer comfortable. It's a strong requirement that any additional tool be closely integrated into the corresponding IDE. For example, Rational Test RealTime is well integrated into WindRiver's Tornado and GreenHills' Multi IDEs.

Another characteristic is that, driven by the dynamic computer-chip industry, new execution platforms and associated development tools are released extremely frequently. This imposes a requirement that testing technology be highly flexible to adapt to these new architectures in record time. A Rational Test RealTime Target deployment for a new target platform is usually achieved in less than a week, often within two days.

Tight Resources and Timing Constraints on the Execution Platform

By definition, an embedded system has limited resources over and above the application it is supposed to run. This is especially true about tiny platforms where: available RAM is less than 1 Kb; the connection to the development environment can only be established using JTAG probes, emulators, or serial links; or when the speed of the micro-processor is just good enough to handle the job. The testing tool faces a difficult trade off: Either have test data on the development platform and send data over the connectivity link at the expense (usually unbearable) of performance, or have the test data interpreted by the test driver on the target platform.

The technology used by Rational Test RealTime involves embedding the test harness onto the target system. This is done by compiling test data previously translated into the application programming language (C, C++ or Ada) within the test harness, using the available cross-compiler, and then linking this test harness object file to the rest of the application. This building chain is made transparent to the user by using the Rational Test RealTime command line interface in makefiles. Optimized code generation, smart link map allocation, and low memory footprint all work to minimize required resources by the test harness on the target platform while providing the following benefits:

- *Timing accuracy is improved and real-time impact on performance is reduced* by using in-target location.
- *Host-target communication is minimized* by avoiding any data information circulating on the connectivity link during the test-case execution. If necessary, observation data are stored in RAM and uploaded with the help of a debugger or emulator when the test case is strictly finished.

Even if cross-development environments are becoming friendlier, a large part of the embedded systems currently shipped are still tricky enough to develop and test that they give most of us headaches. The goal of Rational Test RealTime is to simplify the tough routine of the embedded systems developer.

Lack of Widespread Use of Visual Modeling

Embedded developers like code! Unfortunately, post-secondary graduates are not well trained in visual modeling, and they tend to believe that code is the "real stuff." Although visual modeling, through languages such as the UML, has made a lot of progress toward incorporating embedded knowledge into a design, a majority of embedded systems programmers still love doing most of their work using plain old programming language. And Java is not bringing too many people to design!

To enable developers to work in the way they prefer, Rational Test

RealTime is focusing on helping them design test cases based on the source code of the application, by providing wizards such as test template generators and API wrappers. Making sure the test can apply to the application is the main benefit of this code-based test building process. The drawback is that there is no guarantee that the test case reflects the requirement it should check. Clearly, broader adoption of visual modeling would reduce the gap between requirement and test case, and, along with other Rational products, Rational Test RealTime is also paving the way for this technique named "Model Driven Testing" by offering a Rational Rose RealTime UML sequence diagram to test-case compiler.

Emerging Quality and Certification Standards

For a certain category of embedded systems -- safety-critical systems -- failure is not an option. We find these systems in the nuclear, medical, and avionics industries. Stepping on the zero-failure objective long time ago, the aircraft industry and government agencies (such as the Federal Aviation Administration in the US) joined to describe *Software Considerations in Airborne Systems and Equipment Certification*, referenced as the RTCA's DO-178B standard.² This is the prevailing standard for safety-critical software development in the avionics industry worldwide. As one of the most stringent software development standards, it is also becoming partially adopted in other sectors where life-critical systems are manufactured, such as automotive, medical (FDA just released a standard close to DO-178B), defense, and transportation.

DO-178B classifies software into five levels of criticality related to anomalous software behavior that would cause or contribute to a system function failure. The most critical is level-A equipment, in which failure results in a catastrophic failure condition for the overall system. DO-178B includes very precise steps for making sure level-A equipment is safe enough, in particular in the testing area. Rational Test RealTime meets all mandatory DO-178B test requirements, for all levels, up to and including level-A equipment.

Summary

In this article, we have presented an overview of six aspects of embedded systems testing. Considering the full spectrum from very small to very large systems, and the specific characteristics and constraints of embedded systems, we have deduced a set of requirements that an ideal technology must possess to address the testing of embedded systems. Large portions of this ideal technology are exemplified by Rational Test RealTime, the new Rational test offering to the embedded, real-time, and networked systems domain.

This article is a general introduction to the testing of embedded systems, and will be followed over the course of the year by more articles focusing on some of the topics discussed.

Terminology

by Paul Szymkowiak

Unfortunately, many terms used in the software engineering world have different definitions. The terms used in this article are probably most familiar to those who work in the real-time embedded systems domain. There are, however, other equivalent/related terms that are used in software testing. We provide a mapping in the following table.

Terms Used in This Article	Equivalent/Related Terms
Granule under Test (GuT): A system element that has been isolated from its environment for the purpose of testing.	<i>Related terms:</i> Test Item, Target, Target of Test
Point of Control and Observation: A specific point in a test at which either an observation is recorded of the test environment, or a decision is made regarding the test's flow of control.	<i>Closest equivalent term:</i> Verification Point
Preamble: The actions taken to bring the system to a particular stable starting state required for the test to be executed.	Pre-condition, Setup
Postamble: The actions taken to bring the system to a particular stable end state required for the next test to be executed.	Post-condition, Reset
Virtual Tester: 1. Something external to the GuT that interacts with the granule during a test. 2. An instance of a running test, typically representing the actions of a person.	<i>Related terms:</i> Test Script or Test Driver, Test Stub, Simulator, Tester
Test Harness: An arrangement of one or more test drivers, test-specific stubs, and instrumentation combined for the purpose of executing a series of related tests.	<i>Related terms:</i> Test Suite, Test Driver, Test Stub

Footnotes

1 "Critical Issues Confronting Embedded Solutions Vendors," *Electronics Market Forecast*, <http://www.electronic-forecast.com/>, April 2001.

2 DO-178B, *Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics - RTCA, <http://www.rtca.org/>, January 1992.



For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!