Testing Embedded Systems — Do You Have the GuTs for It?

Vincent Encontre

Rational Software White Paper

TP 317, 11/01

Rational®

the software development company

Table of Contents

Introduction

This paper gives a general introduction to testing embedded systems followed by a discussion of how embedded systems' issues affect testing processes and technologies, and how Rational Test RealTime provides solutions to these issues.

Definitions for a Common Set of Concepts

Let's start with some definitions to settle on a common set of concepts.

What is testing? Testing is a disciplined process that consists of checking that your application's (including its components) behavior, performance, and robustness match expected criteria. One of the main standards, although usually implicit, is that your applications be as defect-free as possible. Therefore, expected behavior, performance, and robustness should be both formally described and measurable. Debugging literally means removing defects ("bugs") and is considered only part of the testing process.

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 infiltrates our daily lives, such as the cell phone in your pocket and all of the wireless infrastructure behind it; the Palm Pilot on your desk; the Internet router through which your emails are channeled; your big-screen home theater system; the air traffic control station, as well as the delayed aircraft it's monitoring! Software now makes up 90% of the value of these devices.

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 the computations 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. For some systems identified as safety-critical, failure is not an option. Therefore, ***testing timing constraints is as important as testing functional behavior for an embedded system***.

Even though the embedded system's testing process resembles that used for many other kinds of applications, some issues important to the embedded world impact it:

- separation between the application development and execution platforms
- a large variety of execution platforms and 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 paper.

These issues greatly affect the ability to test and measure an embedded system. This also explains why testing such systems is so difficult and consequently is one of the weakest points in current development practices. So it's no wonder that, according to a recent study (see [R1]), 50%+ of embedded systems development projects are months behind schedule and only 44% of designs are within 20% of both feature and performance expectations—even when 50%+ of the total development effort is spent on testing.

In the rest of this paper, 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 testing complex systems, as found in the embedded world, and, based on these considerations, we'll add capabilities to our ideal testing technology
- examine what makes embedded systems so difficult to develop and test
- assess how these issues add to the list of features 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

Preparing the Granule Under Test

The first mandatory step in any test is to identify the granule to be tested. The word "granule" is used to avoid using other words such as component, unit, or system, all of which have less generic definitions. For example, in 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 to use 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.

Identify the granule to test, then transform it to a testable granule or a 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 2-way access between the granule and the rest of the application.

Then, build a test driver. It stimulates the GuT with the appropriate input, then measures output information and compares it with the expected response. An adapter is used to allow the test driver to stimulate the GuT. 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 telecom industry. A PCO can be located at the border of, or inside of, the GuT.

Examples of PCOs for a C function granule are:

- Point of Observation inside of 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 of 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're 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.

Describing the Test Case

Describing 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
- and how to exploit them — what information to send and to expect to receive through them, and in what order, if any

The type of test, as well as sent or 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 conducting an appropriate set of tests (also referred to as "test ideas") to draw some requirements for testing the GuT. These requirements have to be translated into formal test cases that take advantage of available PCOs. By "formal", for simplicity, we mean *executable*.

Usually, requirements themselves are not formal and do not naturally translate into a formal test case. 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, especially 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 and C++ are universally known (so there's a reduced 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 do not enable you to write comprehensive 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 the both worlds—reduced learning curve and efficiency in writing test cases.

Another extremely valuable and productive way to write test cases is to use session recorders. When the GuT is stimulated (either manually or by its future environment), particular Points of Observation record information in and out of the GuT, which are automatically translated further into an appropriate test case to be replayed later. An example of such a session recorder is found in Rational Rose RealTime where model execution leads to generating a UML sequence diagram reflecting trace execution, which is then used as a test case model by Rational QualityArchitect RealTime.

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

Deploying and Executing Test

Once described, the test case is then transformed and integrated as the information (versus operative) part of the test driver and stubs. It is important to note that the stub description is an integral part of the test case. Test cases are executed 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* to provide details about which software part of the GuT has been covered
- *control graph* to follow the various logical branches that have been executed
- *information flow* to visualize the 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, or event handling performances

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

Deciding on the Next Steps

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

Test cases can fail for a number of reasons:

- *Nonconformance to the requirements* (including the implicit zero-crash requirement) — You'll have to go back to the implementation or, worse, to the design to fix the problem in the GuT.
- *The test case is wrong* — This happens much more frequently than you might think. Testing, like software, does not always work as expected the first time it's used. Modify the test case to fix the problem.
- *The test case cannot be executed* — Again, like software, everything seems correct, but you cannot deploy or start or connect your test harness to the GuT.

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

- *Reevaluate your tes*t — if you're 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 the reliability of the GuT. The objection can be made that reliability is part of the requirements, and that would be correct. However, the level of reliability is often directly correlated to the level of coverage of the GuT by the overall set of test cases.
- *Code coverage is the most widely used type of coverage* — as implemented in Rational Test RealTime. Testing based on code coverage helps define additional test cases that will increase the coverage level up to the level agreed to 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 granule*s — you'll then apply this generic process to a larger portion of your system as described in the next paragraph.

When does Testing Stop?

This is a perennial question for the software practitioner and it is not this paper's ambition to solve it. However, one heuristic that can be used is to consider the safety criticality of the system under test. Can the system be deemed safety-critical or not? For non-safety-critical systems, testing can be stopped based on more or less subjective criteria such as time-to-market, budget, and "good-enough". However, for safety-critical systems where failure is not an option, the decision to stop testing cannot afford to be made on such criteria. In the section titled *Emerging Quality and Certification Standards* near the end of this paper, there are some recommendations for handling this issue.

Requirement 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, for information sent to and expected from the GuT, and for 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 failure

Rational Test RealTime supports these features and goes beyond these requirements to address the testing of complex systems found in the embedded systems domain.

Six Incremental Steps for Testing Complex Systems

Generic Architecture and Implementation for Complex Systems

Embedded systems are complex systems that can be composed of extremely diverse architectures, ranging from tiny 8-bit microcontrollers up to large distributed systems comprised of multiprocessor platforms. However, two-thirds of these systems run on a real-time operating system (RTOS), available either commercially off-the-shelf or in-house, and implement the concept of threads that extend to the RTOS task or process. (A thread is a granule with an independent flow of control.) In the UML, this concept is referred as a Component, whereas 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% will be using C in 2002, 60% C++, 20% Java, 5% Ada, as noted in [R1]). It isn't 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, whereas Java or C++ are supposedly more productive, thanks to object-oriented concepts. However, it should be noted that programmers of embedded systems are not object devotees!

In the context of embedded systems, a granule can be one of the following. This list is sorted by incremental complexity.

- C function or Ada procedure
- C++ or Java class
- C or Ada (set of) modules
- 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 shows how this generic architecture impacts the various testing steps.

The Six Incremental Steps of Testing

Depending on the type of granule, and according to *common* usage in the industry, discussed later in this section, six testing steps are necessary to check that the application's behavior, performance, and robustness match expected criteria. These steps are:

- software unit testing
- software integration testing
- software validation testing
- system unit testing
- system integration testing
- system validation testing

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 the responsibility of the developer.

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

Software Integration Testing

The GuT is now a set of functions or a cluster of classes. Validating the interface is the essence of integration testing. The same type of Points of Control applies as for unit testing (data-intensive main function call or method-invocation sequences), whereas 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—the performance tests can be run, which should provide a good indication about the validity of the architecture. As for functional testing, the sooner the better. Each forthcoming step will then include performance testing. White-box testing is also the method used during this step. Software integration testing is the developer's responsibility.

Software Validation Testing

The GuT is all of the user code inside of a component. This can be considered the final step in software integration. The use cases are now approaching end-user scenarios and moving away from implementation details. 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 developer's responsibility.

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 or received using the RTOS message queues, for example.

The symmetry found in the message-passing paradigm implies that the distinction between test drivers and stubs can be considered irrelevant at this stage. We will call them *virtual testers,* because each one can both replace and act as another system component vis-à-vis the GuT. "Simulator" and "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 message content against expected messages and the 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. From this step, gray-box testing is the preferred testing method. Knowledge of the interface to the GuT is all that's required. Depending on the organization, system unit testing is either the responsibility of the developer or of a dedicated system integration team.

System Integration Testing

The GuT starts from a set of components within a single node up to and eventually encompassing 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 the responsibility of the system integration team.

System Validation Testing

The GuT is finally the overall complete embedded system. The objectives of this final step are several:

- *Meet end-user functional requirements*. Note that an end-user 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, nonfunctional testing such as load and robustness testing.* Virtual testers can be duplicated to simulate load and programmed to generate failures in the system.
- *Ensure interoperability with other connected equipment*. Check conformance to applicable interconnection standards

It is beyond the scope of this paper to go into details for these objectives. Black box testing is the preferred method—the tester concentrates on both frequent and dangerous use cases.

Deciding How to Sequence These Steps

Now that we have the six incremental steps, how do we use them? There are plenty of criteria used to decide:

- whether all of these steps apply to your systems
- whether all of the selected steps should be performed for all, or only for some, parts of the systems
- the order in which the selected steps should be applied to the selected parts

The decision, based on gathering this criteria, depends heavily on the kind of embedded system you are developing: safety-critical or not, time-to-market, a few or millions deployed, and so on. A paper that provides guidelines to help you with this selection process will be written in the future. You will be able to access through the Rational Developer Network when it's available.

Another way to address these testing steps 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's just a matter of how you can access the GuT, what 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 leave that to another discussion…

Additional Requirements for a Complex Systems Testing Technology

To address the challenge of testing complex embedded systems, the testing technology must add the following capabilities:

- *Manage multiple types of Points of Control* — To stimulate the GuT, do so by using function calls, a method invocation and message passing, or Remote Procedure Calls (RPCs) through different language bindings such as Ada, C, C++, or Java.
- *Offer of 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 versus actual assessment capabilities.

How Embedded Systems Issues Affect Testing Process and Technology

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

Separation Between the Application Development and Execution Platforms

One of the multiple definitions for an embedded system is:

> *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.*

On the development side of things, platform typically means an operating system such as Windows, Solaris, or HP-UX. It should be noted that the percentage of UNIX and Linux users is much higher (40% as noted in [R1]) in the embedded systems domain as compared to other, more 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 (could be a real person or a 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 yet 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, and Linux; the development platforms that leading companies in the device, embedded system, 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 microcontroller-based board to a large distributed and networked system. All platforms will 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.

In general, development for this kind of environment is referred to as "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. Ten years of experience in this technology has resulted in highly efficient code instrumentation.

Another direct consequence of this variety is that vendors of cross-development tools 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 or 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 is 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's 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 microprocessor is just good enough to handle the job. The testing tool faces a difficult tradeoff—either have test data on the development platform and send data over the connectivity link at the expense (usually unbearably) of performance or have the test data interpreted by the test driver on the target platform.

The technology used by Rational Test RealTime is to embed 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 using in-target location reduces real-time impact on performance.
- Avoiding any data information circulating on the connectivity link during the test-case execution minimizes host-target communication. 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.

Although 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 Clearly Designed Visual Models

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 to do most of their work using plain old programming language. And Java is not bringing too many people to design!

To assist developers to work in a way they prefer, Rational Test RealTime focuses on helping them design test cases based on the application's source code 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. Rational Test RealTime is also paving the way for this technique 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 of long ago, the aircraft industry and government agencies (such as the Federal Aviation Authority in the US) joined to describe *Software Considerations in Airborne Systems and Equipment Certification* referenced as the RTCA's DO-178B standard, described in [R2]. 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, or transportation businesses.

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 paper, we have presented an overview of the process of six incremental steps used to test embedded systems. Considering this process (taking into account 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. Rational Test RealTime, the new Rational offering to the embedded systems domain, exemplifies large portions of this ideal technology.

This paper has provided a general introduction for testing embedded systems and will be followed over the year by other articles focusing on the various topics discussed.

Terminology

Paul Szymkowiak wrote this section.

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

Terms Used in This Paper	Equivalent/Related Terms
Granule under Test (GuT): A system element that has been isolated from its environment for testing purposes.	*related terms*: Test Item, Target, Target of Test
Point of Control and Observation: A specific point in a test at which either an observation of the test environment is recorded or a decision is made regarding the test's flow of control.	closest equivalent term: **Verification Point**
Postamble: The actions taken to bring the system to a particular stable end state, required to execute the next test.	Postcondition, Reset
Preamble: The actions taken to bring the system to a particular stable starting state required for to execute the test.	Precondition, Setup
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.	*related terms*: Test Suite or Test Driver, Test Stub
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 a person’s actions.	*related terms*: Test Script or Test Driver, Test Stub, Simulator, Tester

References

[R1] *"Critical Issues Confronting Embedded Solutions Vendors”*, *Electronics Market Forecast*, http://www.electronic-forecast.com/, April 2001.

[R2] DO-178B, *Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics — RTCA, http://www.rtca.org/, January 1992.

About the Author

Vincent Encontre is the Director for Embedded and RealTime, Automated Testing Business Unit, based in Rational's new engineering center in Toulouse, France. When he's not traveling, attending meetings, or answering zillions of emails, Vincent looks at the destiny of Rational Test RealTime as a Rational product and as reusable technologies for next generations of Rational products. Vincent has extensive experience in embedded modeling and testing technologies and best practices. Prior to Rational and ATTOL, Vincent spent 13 years at Philips, then at Verilog, designing, marketing, and supporting software engineering tools, believing these tools could help build better software faster.

In his spare time, Vincent plays soccer with his three boys and a few others, and just enjoys the wonderful art de vivre from the south of France (before it's too late…).