Agile analysis practices for safety-critical software development

Bruce Douglass

February 19, 2013

Because of their discipline and efficiency, agile development practices should be applied to the development of safety-critical software. Bruce Douglass, author of the IBM Rational Harmony for Embedded RealTime Development process, explains the key analysis practices for the development of safety-critical systems and how they can be realized in an agile way.

Agile methods have a reputation for being fast and adaptive but undisciplined and lacking in robustness. However, agile methods require a great deal of discipline, and these practices enhance both quality and team productivity. Because of this, agile development practices should be applied to the development of safety-critical systems. This article describes how agile analysis methods can be used in the development of safety-critical systems.

Properties of safety-critical systems

The term safety-critical refers to systems that either can cause harm or are responsible for preventing harm. Such systems range from medical devices to automotive braking systems, nuclear power plant control to avionic flight management systems. Most safety-critical systems must be certified by a regulatory agency to ensure that they are fit-for-purpose. This means that proper development practices have been applied toward "system correctness" as the outcome and that adherence to the objectives of the relevant standards can be demonstrated.

Because it is virtually impossible to demonstrate deterministic correctness for any significant piece of software, most of these standards have concentrated on specifying process objectives and requirements for evidence that the process standards have been followed. For example, the recently released avionics standard, DO-178C, Software Considerations in Airborne Systems and Equipment Certification, requires that a system project supply evidence in several categories for up to 71 objectives, depending on the safety-criticality level. Supplements for this standard, such as DO-331, Model-Based Development and Verification, and DO-332, Object-Oriented Technology and Related Techniques, add objectives if those technologies are employed. Most such objectives relate to the following phases:

- Planning (including the specification of the safety level of the device)
  - Software development process definition
• Requirements management
• Software design and coding
• Configuration management
• Quality assurance
• Integration
• Verification
• Tool qualification
• System certification

It is important to note that while these standards specify the objectives that any process must meet if it is used to develop a safety-critical system, the standards do not specify the processes themselves. As long as a process can be demonstrated to meet the needs of the relevant standard, the development team is free to use whatever processes they want to use. This leaves the option available to use state-of-the-art agile methods, with all of their accompanying advantages, provided that the safety objectives can be achieved.

Overview of the Harmony process

The Harmony process is a model-driven agile process for real-time and embedded systems development (see the Real-Time Agility citation in Resources for more information). It has a strong concept of architecture and provides enough rigor to be used to develop a system when failure can be extremely costly, such as safety-critical systems.

The Harmony process has many practices that integrate into a cohesive method for developing software. These are among the important practices:

• Analysis-oriented practices
  • Initial Safety Analysis
  • Continuous Safety Assessment
  • Executable Requirements Models
  • Traceability Analysis
• Design-oriented practices
  • Model-Based Engineering
  • 5 Views of Architecture
  • Defensive Design
• Quality-oriented practices
  • Continuous Execution
  • Test-Driven Development
  • Continuous Integration
  • Incremental Development with the Harmony Microcycle
  • Work Product Reviews
  • Worker Task Audits
• Evidence-oriented practices
  • Manage Traceability Records
  • Test Coverage Analysis

This article covers only the first set, analysis-oriented practices.
Note:
Harmony also supports agile systems engineering, but this article focuses on software development only.

Figure 1 shows an overview of the Harmony Software Process. The activities shown in the diagram represent the realization of practices — either more standard agile practices or more traditional practices done in an agile way. There three key parallel flows: iterative design, quality assurance audit, and configuration management. The iterative design flow is further decomposed into two stages of parallel tasks. The first stage comprises Prespiral planning, definition and deployment of the development environment, and requirements development. The second stage comprises project control, the Microcycle, and change management. The software development takes place within the iterated step known as the Microcycle.

Figure 1. Harmony Software Process overview

Initial safety analysis

The purpose of the initial safety analysis is to identify safety concerns that are inherent to the system, its use, and the environment in which it operates. These concerns do not include the safety impact of technological decisions made during design (unless they are already known before work begins on the project). However, those will be identified and analyzed as a part of another practice covered in this article, Continuous Safety Assessment.
Inherent safety concerns are, for the most part, independent of the designs necessary to realize them. For example, an automobile is a device that conveys people along roads. As such, there are several basic concerns about the safety of passengers and people around the vehicle:

- Crashing due to inability to stop the vehicle in a timely and effective way
- Crashing due to inability to steer the vehicle accurately
- Crashing due to inability to control forward or backward motion
- Crashing due to lack of visual feedback for the driver to make automotive control actions
- Falling out of the vehicle

Notice that none of these concerns has anything to do with how the car applies, removes, or directs motive force. It is just in the nature of cars to move, and the essential safety concerns that arise due to crashing and the release of kinetic energy can result in injury or death to passengers or people nearby. Safety concerns arising from design decisions are addressed in the next section on the Continuous Safety Assessment practice.

The primary means for capturing the safety-relevant metadata is the hazard analysis, and the most common means for acquiring this data is through a fault tree analysis (FTA). The task that performs this analysis is "Perform Initial Safety and Reliability Analysis" (see Figure 2) and is part of the Prespiral Planning activity shown in Figure 1. So, in what way is this an agile practice?

The Harmony implementation of this practice is agile in these ways:

- It is done incrementally, in parallel and in collaboration with the development of requirements.
- It incrementally addresses the safety concerns in a risk-based priority scheme, addressing the highest safety risks first. As risks are addressed, the requirements model is updated, modeled, and verified.
- It integrates into the Executable Requirements practice so that a test-driven development approach can incrementally verify the completeness, consistency, and correctness of the requirements created from the FTA.

Prespiral planning practice comprises the six parallel tasks shown in Figure 2:

- Create Schedule
- Create Team structure
- Plan for Reuse
- Plan for Risk Reduction
- Specify Logical Architecture
- Perform Initial Safety and Reliability Analysis
Figure 2. Prespiral planning

The key outcomes of this practice include:

- Initial hazard analysis (including both unmitigated and mitigated safety concerns)
- Fault tree analysis
- Safety-relevant requirements (including required safety measures)

FTA is shown as a set of diagrams that graphically show how events and conditions (both normal and fault) combine to manifest hazardous conditions. This can be done in a stand-alone FTA tool, or it can be integrated with the requirements and design model in IBM® Rational® Rhapsody® software, with its FTA Profile (formerly called the Safety Analysis Profile).

In Figure 3, you can see how different fault conditions logically combine to manifest a hazard. In addition to standard FTA conditions, event, and logical operators, the FTA profile also provides traceable links to associated requirements, as well as design elements that could manifest, detect, or mitigate the safety concern. In this case, the hazard is *target misidentification*, which could be caused by different intermediate conditions, such as unreduced noise in the image, corrupted communications, corrupted target data, a bad target specification, and so on.

Each of these resulting conditions is due to a combination of causal events and conditions. At the bottom of the FTA tree, there are primitive elements, such as normal events and conditions, and basic and undeveloped faults. By performing this kind of analysis — for each hazard the system presents — you can reason about the kinds of safety mechanisms that the system must possess to fulfill its mission in a safe way.

For example, you can see that we've put a safety measure into place, CRC computation over message contents, to detect message corruption that could lead to target misidentification. For the originating fault to manifest the hazard, a second fault must occur (CRC fails to detect the corruption). Therefore, in this case, we've made it less likely for the hazard to manifest, because both the original fault and a failure of the protection mechanism must occur. Thus, we end up adding a safety requirement for a means to detect message corruption (specifically, a CRC) based on this initial safety analysis.
Each of the graphical elements on the FTA diagram is more than just a drawing. Each one is a model element with specific semantics and information (known as metadata) about the thing that it represents. Hazards, for example, commonly have the following metadata:

- Fault tolerance time
- Probability (unmitigated)
- Probability (mitigated)
- Severity (unmitigated)
- Severity (mitigated)
- Risk (mitigated)
- Risk (unmitigated)
- Safety integrity level

Faults are usually characterized by a different set:

- MTBF (Mean Time Between Failures)
- Probability
- Mechanism of action
- Manifesting element
- Detecting element
• Detection means
• Mitigating element
• Mitigating strategy

As FTA diagrams are created for all of the hazards that a system might manifest, this metadata can be summarized in tabular form. When this is done at the start of a project, it is known as the initial hazard analysis.

Continuous Safety Assessment

A design that implements a means to apply motive force introduces additional safety concerns, known as design safety concerns. For example, we can cause an automobile to move with a diesel engine (adding the risk of fire and explosion because of the use of flammable fuel), with a large battery (adding the risk of electrocution and hazardous chemical exposure), nuclear power (with a risk of radiation and hazardous materials exposure) or a flux capacitor (risking the creation of a black hole). These are considered design or technological hazards and are typically identified and analyzed as the design decisions are made.

In the Harmony process, the Continuous Safety Assessment practice is realized by the Update Safety and Reliability Analysis task within the Control Project activity that was shown in Figure 1. The tasks in the Control Project activity are shown in Figure 4. As you can see, this task runs in parallel with the entire Microcycle in which software is developed. This means that this analysis is done as the software is specified, designed, and implemented. This task is agile in that, rather than being performed at the end of the project as is common in waterfall-style projects, the objective is to identify and address technological and design safety concerns continuously, throughout the process. The outcome of this task is updated requirements that will be accepted in the current or subsequent design iterations.
Figure 4. Control Project activity

The control project activity comprises the four tasks cited in Figure 4:

- Manage iteration
- Manage risks
- Update safety and reliability analysis
- Refine and deploy the development environment

The purpose of this practice is to ensure that the system remains safe as design patterns and technologies are applied to and implemented in the system.

These are the key outcomes of this practice:

- Design hazard analysis
- FTA (updated)
- Safety-relevant requirements
- Safety assessment report

Executable Requirements Modeling

Requirements are problematic. They are typically captured in human-readable text and used to specify the required properties of the system, both functional and quality-of-service. The problem is that systems are enormously complex, and doing a full and complete specification in human language is difficult, because it requires hundreds or thousands of individual statements to capture the richness of the system functionality and performance. Human languages are highly expressive
but suffer from ambiguity and a lack of precision. Furthermore, there is no reliable means by which a large set of requirements can be verified for completeness, accuracy, and consistency.

Models, on the other hand, can be very precise and unambiguous. Although they lack the richness of expression found in human languages, they make up for that lack with increased rigor. Models can be constructed in such a way that supports verification, either through formal analysis (such as reachability analysis) or execution and tests. Executable models are precise enough to support simulation or execution, so you can specify the system data and control transformations under all relevant circumstances and conditions, and then verify that the specification meets the needs of the users.

Imagine looking at a document that contains more than 2000 requirements for a patient ventilator. It details what the system does in a variety of cases for setting system parameters, such as tidal volume (amount delivered per breath), oxygen concentration, balance gas composition, respiration rate, and so on. How might you determine how the system behaves if the user first sets respiration rate to 20 breaths/min, with a tidal volume of 600 ml, and then wants to set oxygen concentration? The range of permitted concentrations is likely to vary, depending on these settings, to ensure an adequate total oxygen flow.

In a typical requirements document, you would have to search — long and hard — to find all of the relevant requirements relating to this question to determine their consistency and completeness. And you might not find a conclusive answer. In an executable requirements model, you can run the simulation to see what has been specified for this case. Ultimately, it is important to remember that the system will do something very specific, regardless of whether it is specified or not. If you want it to be right, it is important to define "right" within the specification.

This is the Harmony way to specify requirements in terms of models:

1. Cluster the requirements in terms of use cases, which are independent (in terms of the requirements, not necessarily in terms of the implementation).
2. Detail each use case with scenarios in sequence diagrams. Each message on the sequence diagram should relate to some requirement that would otherwise be specified textually. The lifelines in the sequence diagrams should be the use case and the actors (elements outside of the system with which it interacts).
3. Detail the use case with a normative state machine in which each scenario is a path through that state machine. The triggering events on the state machine are (mostly) messages from the actors; the actions on the state machine are (mostly) messages back to those same actors.
4. Because the requirements can be added to scenarios and the state machine, and then verified incrementally, loop back to Step 2 to add more requirements until all of the behavior and constraints for the use case have been modeled.

This workflow is shown in Figure 5. The agile nature of the workflow shows up in a couple of ways:

1. We do a "depth-first" development of the requirements model. This is both at a gross level by developing executable use case specifications, but also at a fine-grained level
by incrementally constructing this executable use case specification a little bit at a time, continuously verifying it as it is being developed.

2. Traceability is likewise added incrementally, rather that at the end of the requirements phase as is common in waterfall-style processes.

This state machine forms the normative specification of the use case and is executable if properly formed. Figure 6 shows a snapshot of a running model, with parts implemented in Rhapsody (the highlighted color in the state machine identifies the current state of the use case in the simulation), with some control logic being co-simulated in MathWorks' Simulink.

Figure 5. Executable Requirements Modeling practice

So, the richness of text or the precision of models? The best-practice answer is to use the best of both. We do that with trace links added in the Traceability Analysis practice, discussed next.
Figure 6. Executable requirements model example

Traceability analysis

The development of a safety-critical system produces a large number of work products, each focusing on different aspects of the system. In order to ensure that they are consistent, traceability links are added to show how each element from one viewpoint relates to elements in the other viewpoints. Commonly required trace links include

- Stakeholder requirements to system requirements
- System requirements to system use cases
- System requirements to software requirements
- Software requirements to software use cases
- Software requirements to architecture
- Software requirements to design
- Software requirements to source code
- Software requirements to test cases
- Architecture to design
- Design to source code
- Source code to EOC (executable object code)
- Test cases to source code
- Test cases to test results

These trace links should be bidirectional. For example, it should be possible to trace from a requirement to the design elements that realize it, as well as to trace from a design element to all of the requirements that it realizes.

Creating and managing these trace links can be quite a challenge, although requirements management tools, such as Rational® DOORS®, automate much of the work. There are many ways to represent the set of trace links. Spreadsheets are the most common, but other tabular forms can be used in addition to DOORS software. Trace links can also be added by using
the `<trace>` dependency in UML in models. Figure 7 shows how trace links are represented graphically in a use case diagram, displaying the links of individual requirements to the use case. DOORS is agile-agnostic, but it can be used in an evolutionary development lifecycle to incrementally develop the requirements and manage their evolving properties.

**Figure 7. Trace links in UML**

The links from a set of such diagrams can be shown in a generated table, if desired, as shown in Figure 8.
Figure 8. Generated traceability table

Traceability should be added incrementally. It is best done in real time, as soon as the design elements stabilize with the Test-Driven Development practice. For example, traceability from software requirements to software use cases can be added as the requirements are incrementally added to the use case scenarios and state machines. This is done as steps within the tasks shown in Figure 5.

Summary

Safety-critical systems are more difficult to develop than their non-safety-critical counterparts. In addition to normal concerns about quality and time-to-market, safety-critical systems must also meet the demanding objectives of relevant safety standards and are subject to rigorous certification. For the most part, teams and organizations move to agile methods for improved quality, project predictability, and engineering efficiency. The common agile practices apply well to safety-critical systems, but they must be tailored and customized to ensure that safety objectives are met.

The Harmony process includes a large number of practices, clustered into Analysis, Design, Quality, and Evidentiary. The key analysis practices discussed in this article are the Initial Safety Analysis, Continuous Safety Assessment, Executable Requirements, and Traceability Analysis practices. These practices enhance the most commonly used agile practices by bringing the rigor and completeness to the development process that are necessary for safety critical systems development.
Related topics

- For more information, see Delivering agility in real-time and embedded development on IBM.com.
- For more in-depth information, read Real-Time Agility by Bruce Powel Douglass (Addison-Wesley, 2009) for a detailed discussion of the Harmony process.
- Also see this related article: Safety-related software development using a model-based testing workflow by Paul Urban and Udo Brockmeyer, PhD (IBM developerWorks, January 2013)
- Download a free trial version of software.

© Copyright IBM Corporation 2013
 Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)