Analysis Practices of Agile for Safety-Critical Software Development

Dr. Bruce Powel Douglass, Ph.D.
Chief Evangelist
IBM Rational

Abstract:
Because of their discipline and efficiency, agile development practices should be applied to the development of safety-critical software. This paper discusses a few of 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. In fact, 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 paper will talk about how agile 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, 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, i.e., that proper development practices have been applied toward “system correctness” as the final outcome, and that adherence to the objectives of the relevant standards can be demonstrated.

Since 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. DO-178C

(*Software Considerations in Airborne Systems and Equipment Certification*), the recently released avionics standard, for example, 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 additional objectives if those technologies are employed. Most such objectives relate to 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, and 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 desire. This leaves the option available to use state-of-the-art agile methods, with all their accompanying advantages, provided that the safety objectives can be achieved.

Overview of the Harmony Process

The Harmony process[1] is a model-driven agile process for real-time and embedded systems development. It has a strong concept of architecture and provides enough rigor to be used to develop system whose failure can be extremely costly – such as safety-critical systems.

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

- Analysis-Oriented Practices
    - Initial Safety Analysis
    - Continuous Safety Assessment
    - Executable Requirements Models
    - Traceability Analysis

[1] See my book *Real-Time Agility* (Addison-Wesley, 2009) for a detailed discussion of the Harmony Process.

- 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

We'll discuss only the first set – Analysis Practices – in this paper.

The overview of the Harmony Software Process[2] is shown in **Figure 1**. The activities shown in the figure represent the realization of practices – either more standard agile practices or more traditional practices done in an agile way. The software development *per se* takes place within the iterated step known as the *Microcycle.*

[2] Harmony also supports agile systems engineering but this paper will focus on software development only.

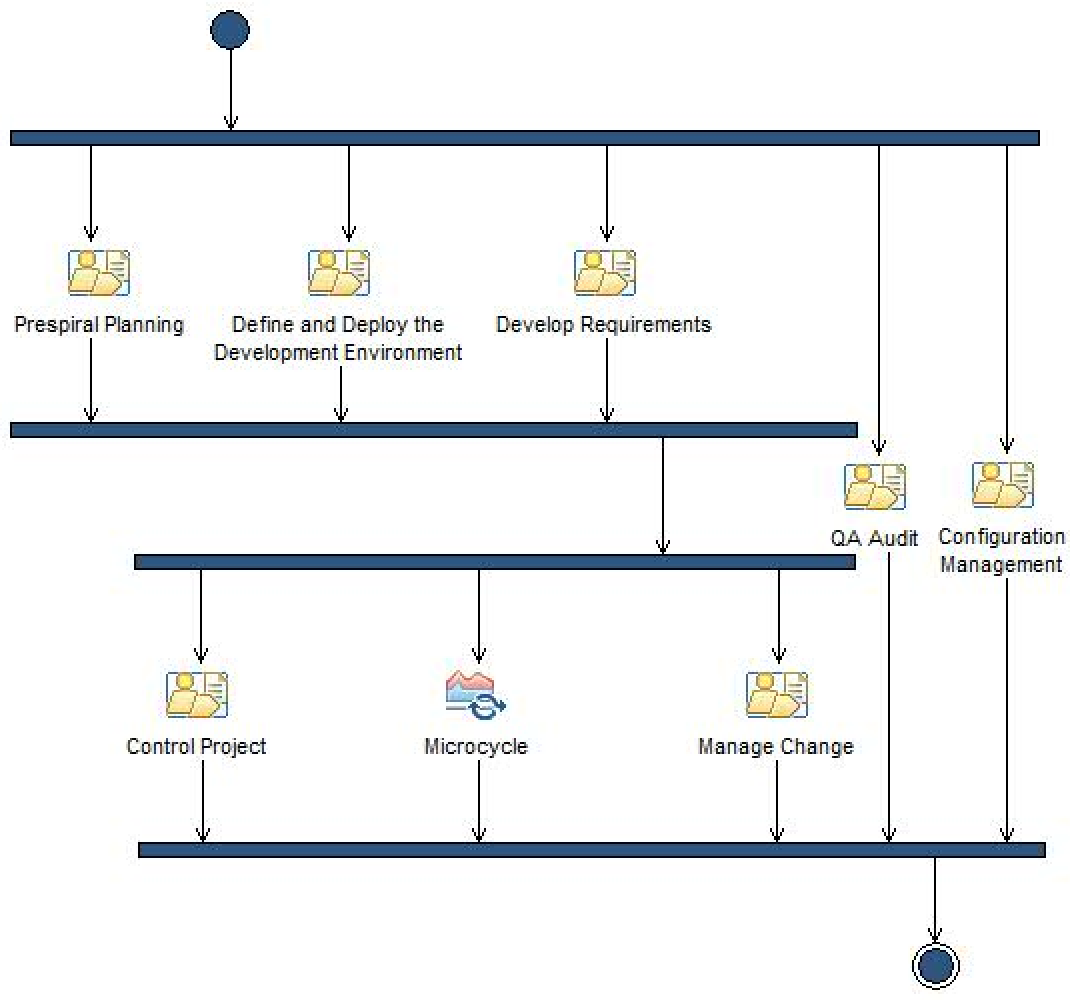


Figure 1: Harmony Software Process Overview

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), but those will be identified and analyzed as a part of another practice, *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 and, as such, has a number of basic concerns about the safety of passengers and people around the vehicle, including:

- 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 vs backwards motion
- Crashing due to lack of visual feedback for the driver to make automotive control actions
- Falling out of the vehicle

Note 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 have to due to crashing and the release of kinetic energy resulting in injury or death to passengers or standers by. 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 known as “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 practice agile?

The Harmony implementation of this practice is agile in that

1. It is done incrementally, in parallel and in collaboration with the development of requirements
2. 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
3. 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

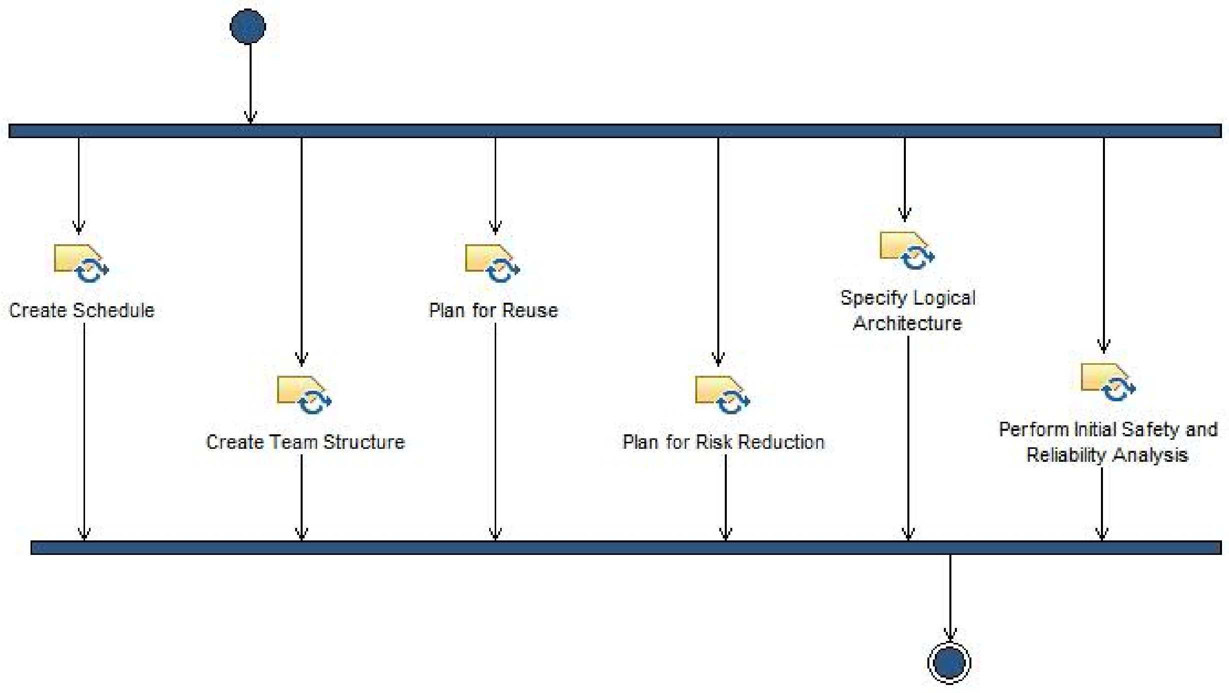

Create Schedule
Create Team Structure
Plan for Reuse
Plan for Risk Reduction
Specify Logical Architecture
Perform Initial Safety and Reliability Analysis

Figure 2: Prespiral Planning

The key outcomes of this practice include

- Initial Hazard Analysis (including both unmitigated and mitigated safety concerns)
- FTA
- 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 together 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 the Rational Rhapsody tool with its FTA Profile[3]. In **Figure 3**, we see how different fault conditions logically combine to manifest a hazard. Besides 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 a number of different intermediate conditions, such as unreduced noise in the image, corrupted communications,

[3] **Formerly known as the "Safety Analysis Profile."**

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. Down at the bottom of the FTA tree are primitive elements such as normal events and conditions, and basic and undeveloped faults. This kind of analysis – repeated for each hazard the system presents – allows us to reason about the kinds of safety mechanisms the system must possess to fulfill its mission in a safe way. For example, we can see that we’ve put into place a safety measure – CRC computation over message contents – to detect message corruption, which could ultimately lead to target misidentification. For the originating fault to manifest the hazard, a second fault (CRC fails to detect the corruption) must occur. Thus, in this case, we’ve made is less likely for the hazard to manifest since both the original fault and a failure of our 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.

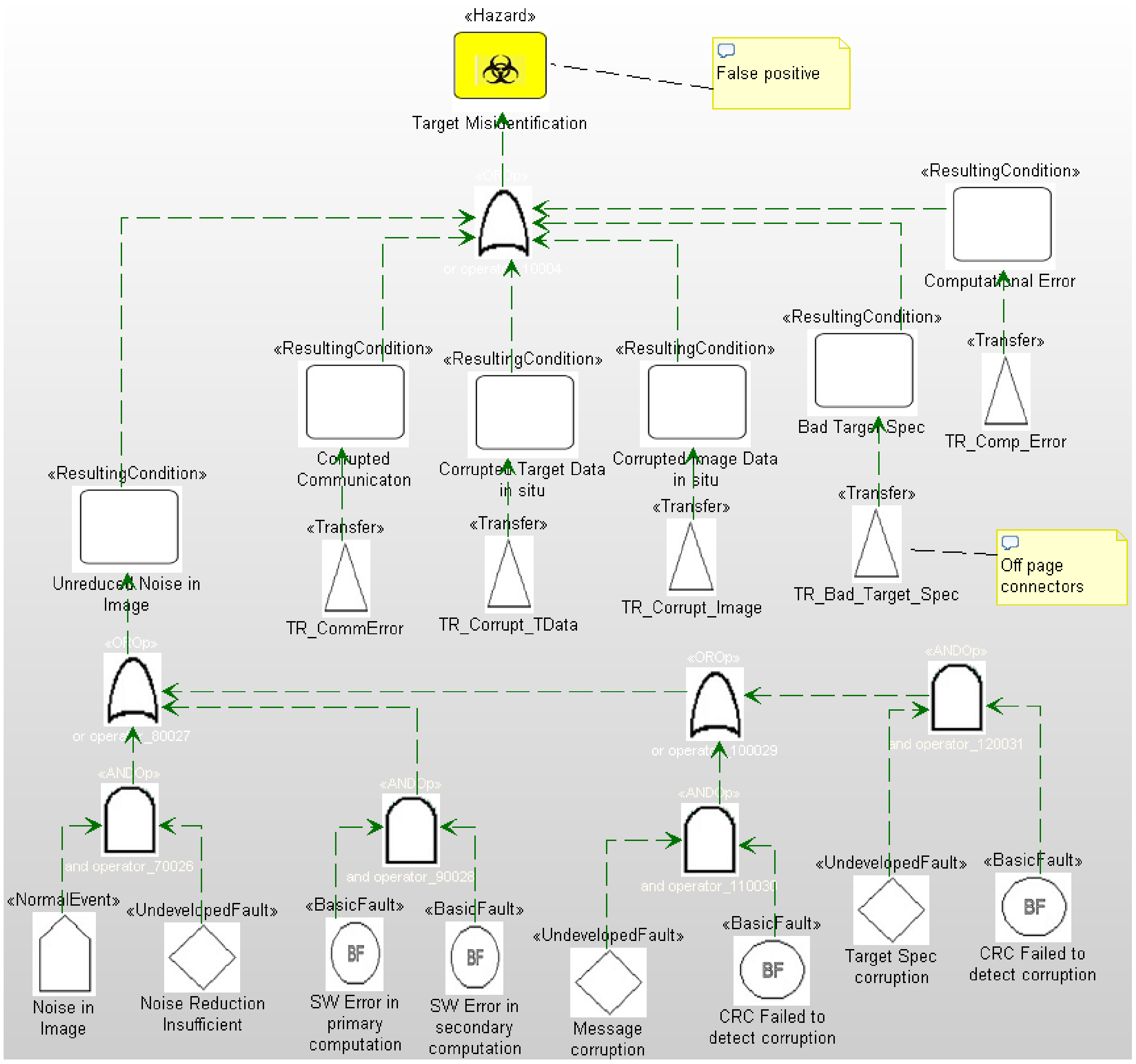


Figure 3: FTA Diagram in Rational Rhapsody using the FTA Profile

Each of the graphical elements on the FTA diagram is more than just a pretty picture. It is a model element with specific semantics and information (known as meta-data) about the thing it represents. Hazards, for example, commonly have the following meta-data:

- 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
- Probability
- Mechanism of action
- Manifesting element
- Detecting element
- Detection means
- Mitigating element
- Mitigating strategy

As FTA diagrams are created for all the hazards a system may manifest, this metadata can be summarized in tabular form. When done at the start of a project, this 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 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 it is not done at the end of the project – as is commonly in waterfall-style projects – but instead seeks 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.

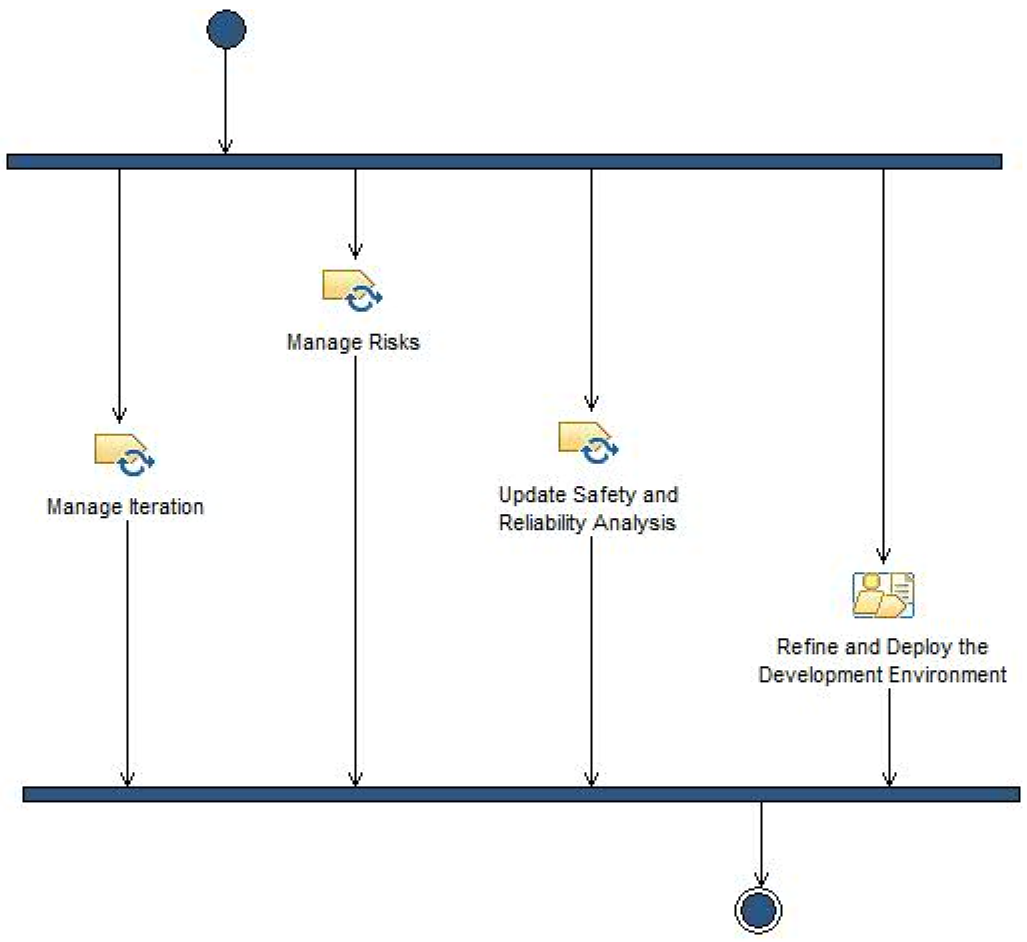

Manage Risks
Manage Iteration
Update Safety and Reliability Analysis
Refine and Deploy the Development Environment

Figure 4: Control Project Activity

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

The key outcomes of this practice are

- 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 are 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, requiring 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. While 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 as to support *verification* either through formal analysis (such as reachability analysis) or execution and test. Executable models are precise enough to support simulation or execution so that 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 containing over 2000 requirements for a patient ventilator that 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, then 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 – in the document to find all 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.

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

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 the 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 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 out 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 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, 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 commonly done in waterfall-style processes.

This state machine forms the normative specification of the use case and is – if properly formed – executable. **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 Simulink.

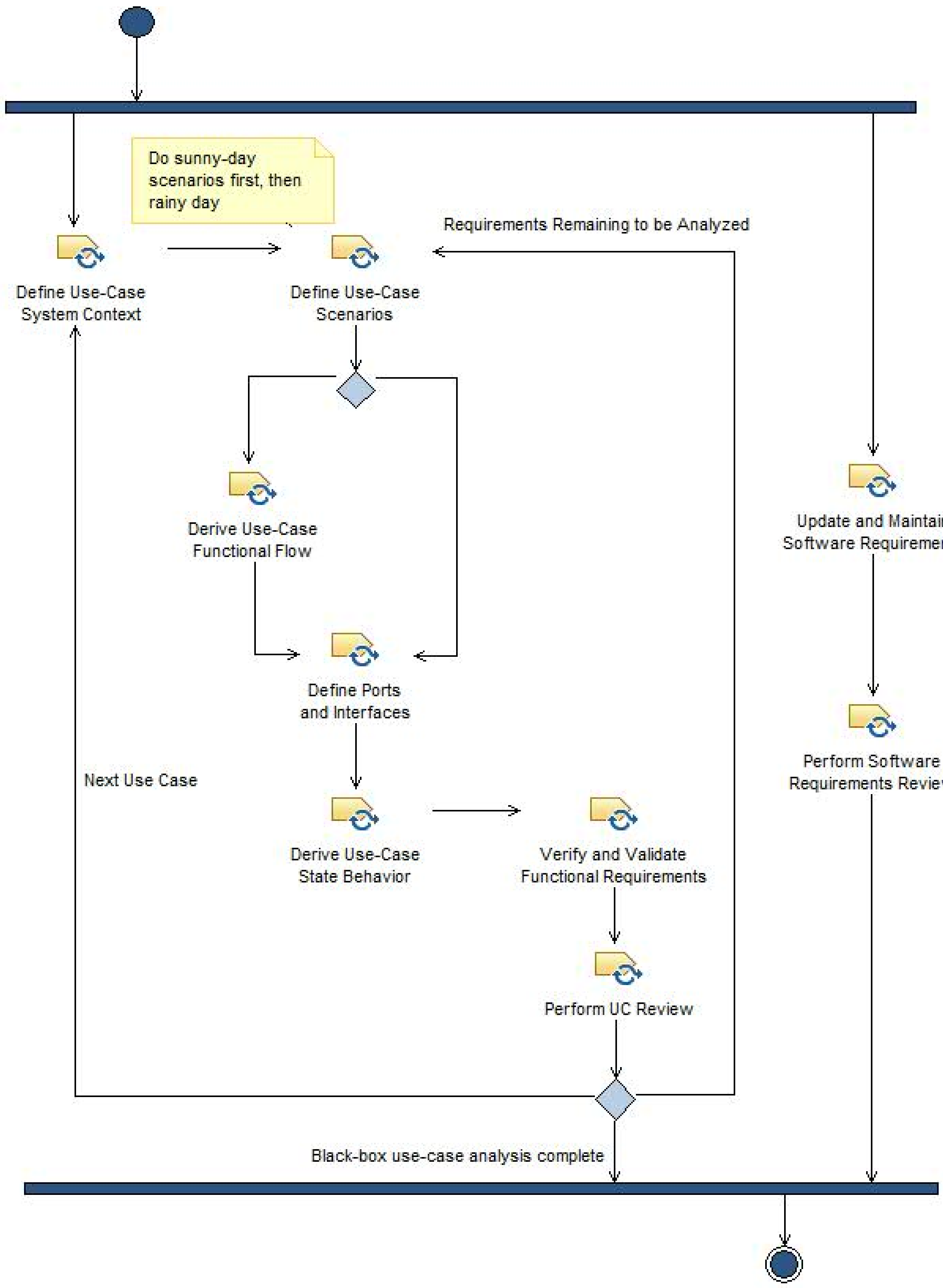

Do sunny-day scenarios first, then rainy day
Requirements Remaining to be Analyzed
Define Use-Case System Context
Define Use-Case Scenarios
Derive Use-Case Functional Flow
Update and Maintain Software Requirements
Define Ports and Interfaces
Perform Software Requirements Review
Next Use Case
Derive Use-Case State Behavior
Verify and Validate Functional Requirements
Perform UC Review
Black-box use-case analysis complete

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 traceability links added via the Traceability Analysis practice, discussed next.

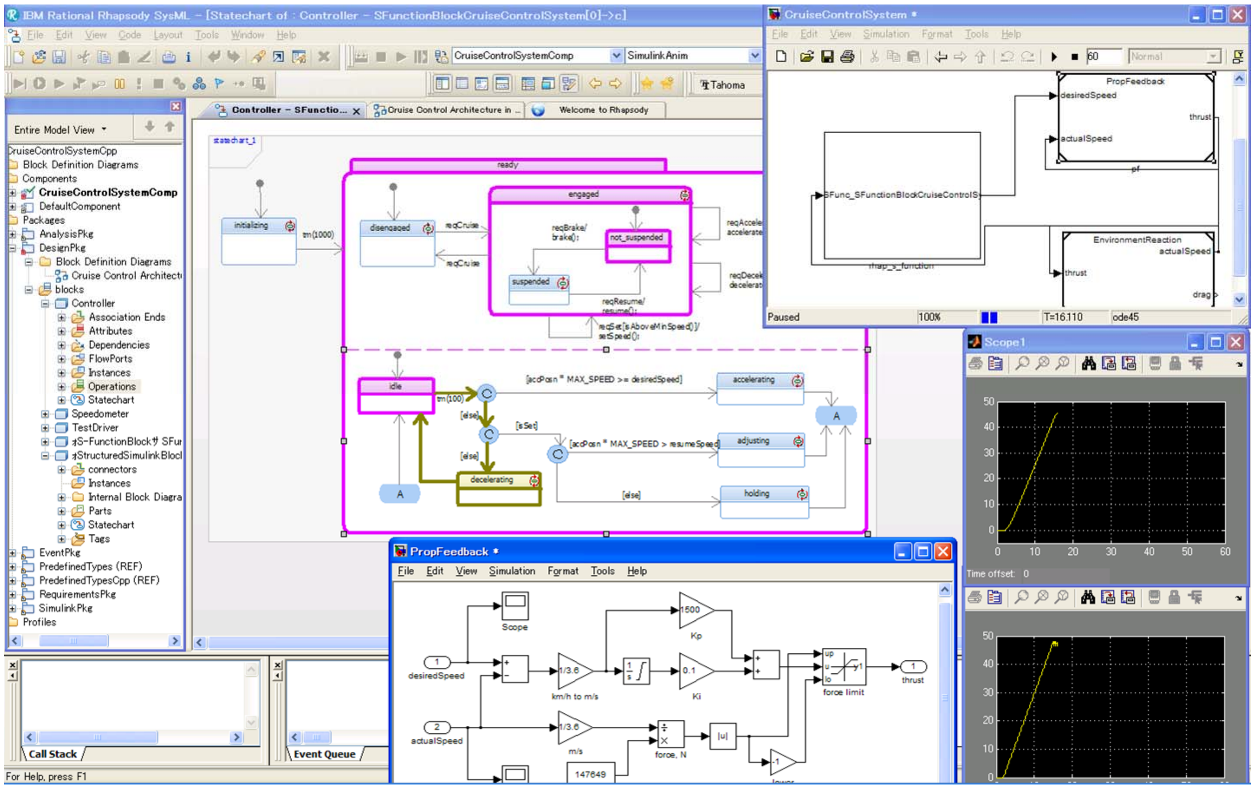


Figure 6: Executable Requirements Model Example

Traceability Analysis

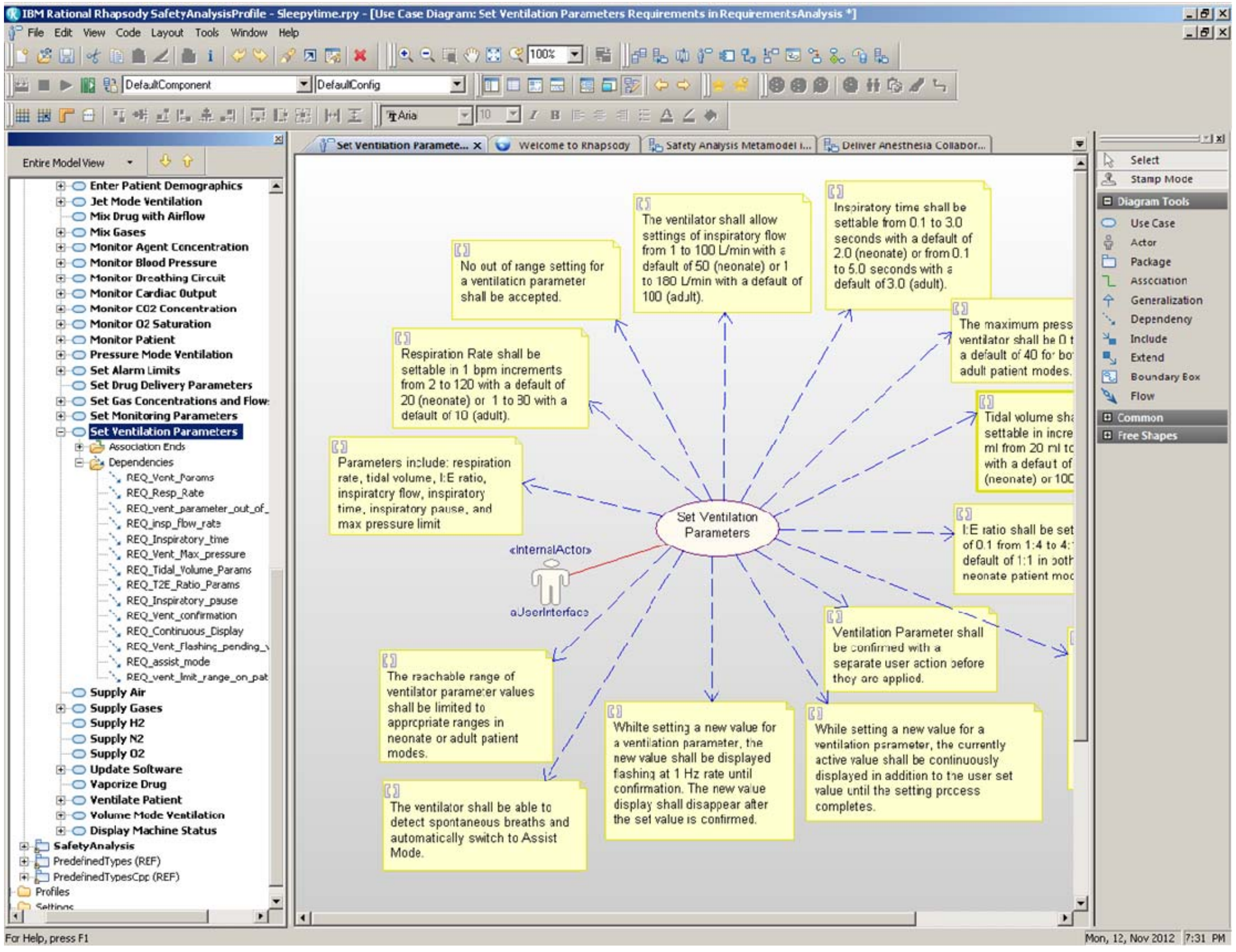
The development of a safety-critical system produces a large number of work products, each focusing on different aspects of the system. In order to ensure that they are consistent, traceability links are added to show how each element from one viewpoint relates to elements in the other viewpoints. Some 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 tracing from a design element to all the requirements 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. Trace links can also be added 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 itself is process-agnostic, but it can be used in an agile evolutionary development lifecycle to incrementally develop the requirements and manage their evolving properties.



No out of range setting for a ventilation parameter shall be accepted.

The ventilator shall allow settings of inspiratory flow from 1 to 100 L/min with a default of 50 (neonate) or 1 to 180 L/min with a default of 100 (adult).

Inspiratory time shall be settable from 0.1 to 3.0 seconds with a default of 2.0 (neonate) or from 0.1 to 5.0 seconds with a default of 3.0 (adult).

The maximum press ventilator shall be 0 t a default of 40 for bo adult patient modes.

Respiration Rate shall be settable in 1 bpm increments from 2 to 120 with a default of 20 (neonate) or 1 to 30 with a default of 10 (adult).

Tidal volume sha settable in incre ml from 20 ml tc with a defaut of (neonate) or 10C

Parameters include: respiration rate, tidal volume, I:E ratio, inspiratory flow, inspiratory time, inspiratory pause, and max pressure limit

I:E ratio shall be set of 0.1 from 1:4 to 4: default of 1:1 in both neonate patient mod

Ventilation Parameter shall be confirmed with a separate user action before they are applied.

The reachable range of ventilator parameter values shall be limited to appropriate ranges in neonate or adult patient modes.

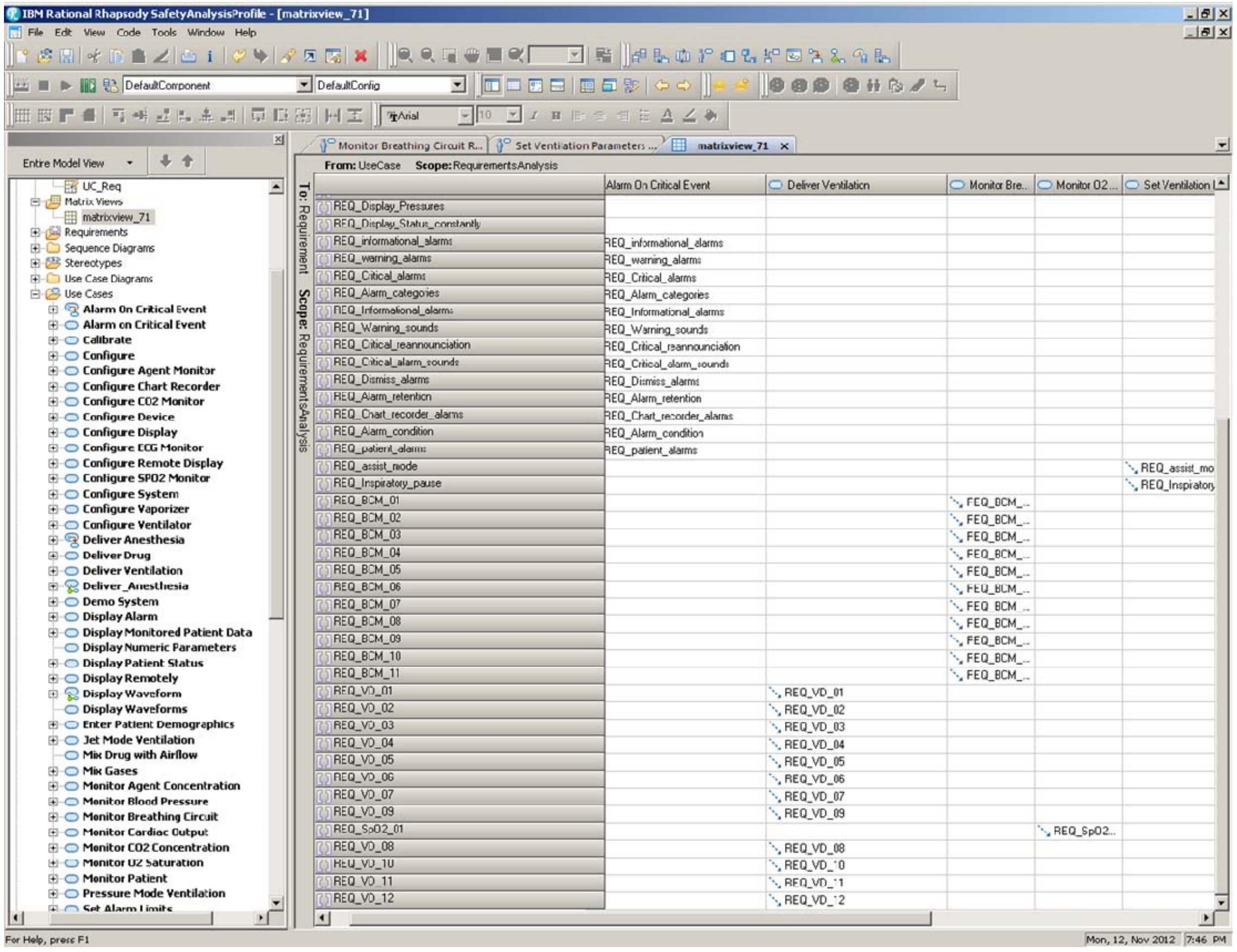
Whilte setting a new value for a ventilation parameter, the new value shall be displayed flashing at 1 Hz rate until confirmation. The new value display shall disappear after the set value is confirmed.

While setting a new value for a ventilation parameter, the currently active value shall be continuously displayed in addition to the user set value until the setting process completes.

The ventilator shall be able to detect spontaneous breaths and automatically switch to Assist Mode.

Figure 7: Trace links in UML

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



IBM Rational Rhapsody SafetyAnalysisProfile - [matrixview_71]

File Edit View Code Tools Window Help

DefaultComponent DefaultConfig Arial 10

Entire Model View

- UC_Req
- Matrix Views
  - matrixview_71
- Requirements
- Sequence Diagrams
- Stereotypes
- Use Case Diagrams
- Use Cases
  - Alarm On Critical Event
  - Alarm on Critical Event
  - Calibrate
  - Configure
  - Configure Agent Monitor
  - Configure Chart Recorder
  - Configure CO2 Monitor
  - Configure Device
  - Configure Display
  - Configure ECG Monitor
  - Configure Remote Display
  - Configure SPO2 Monitor
  - Configure System
  - Configure Vaporizer
  - Configure Ventilator
  - Deliver Anesthesia
  - Deliver Drug
  - Deliver Ventilation
  - Deliver_Anesthesia
  - Demo System
  - Display Alarm
  - Display Monitored Patient Data
  - Display Numeric Parameters
  - Display Patient Status
  - Display Remotely
  - Display Waveform
  - Display Waveforms
  - Enter Patient Demographics
  - Jet Mode Ventilation
  - Mix Drug with Airflow
  - Mix Gases
  - Monitor Agent Concentration
  - Monitor Blood Pressure
  - Monitor Breathing Circuit
  - Monitor Cardiac Output
  - Monitor CO2 Concentration
  - Monitor O2 Saturation
  - Monitor Patient
  - Pressure Mode Ventilation
  - Set Alarm Limits

Monitor Breathing Circuit R... | Set Ventilation Parameters ... | matrixview_71

From: UseCase **Scope:** RequirementsAnalysis

To: Requirement **Scope:** RequirementsAnalysis

	Alarm On Critical Event	Deliver Ventilation	Monitor Bre...	Monitor O2...	Set Ventilation
REQ_Display_Pressures					
REQ_Display_Status_constantly					
REQ_informational_alarms	REQ_informational_alarms				
REQ_warning_alarms	REQ_warning_alarms				
REQ_Critical_alarms	REQ_Critical_alarms				
REQ_Alarm_categories	REQ_Alarm_categories				
REQ_Informational_alarms	REQ_Informational_alarms				
REQ_Warning_sounds	REQ_Warning_sounds				
REQ_Critical_reannounciation	REQ_Critical_reannounciation				
REQ_Critical_alarm_sounds	REQ_Critical_alarm_sounds				
REQ_Dismiss_alarms	REQ_Dismiss_alarms				
REQ_Alarm_retention	REQ_Alarm_retention				
REQ_Chart_recorder_alarms	REQ_Chart_recorder_alarms				
REQ_Alarm_condition	REQ_Alarm_condition				
REQ_patient_alarms	REQ_patient_alarms				
REQ_assist_mode					REQ_assist_mo
REQ_Inspiratory_pause					REQ_Inspiratory
REQ_BCM_01			FEQ_BCM_...		
REQ_BCM_02			FEQ_BCM_...		
REQ_BCM_03			FEQ_BCM_...		
REQ_BCM_04			FEQ_BCM_...		
REQ_BCM_05			FEQ_BCM_...		
REQ_BCM_06			FEQ_BCM_...		
REQ_BCM_07			FEQ_BCM_...		
REQ_BCM_08			FEQ_BCM_...		
REQ_BCM_09			FEQ_BCM_...		
REQ_BCM_10			FEQ_BCM_...		
REQ_BCM_11			FEQ_BCM_...		
REQ_VD_01		REQ_VD_01			
REQ_VD_02		REQ_VD_02			
REQ_VD_03		REQ_VD_03			
REQ_VD_04		REQ_VD_04			
REQ_VD_05		REQ_VD_05			
REQ_VD_06		REQ_VD_06			
REQ_VD_07		REQ_VD_07			
REQ_VD_09		REQ_VD_09			
REQ_SpO2_01				REQ_SpO2...	
REQ_VD_08		REQ_VD_08			
REQ_VD_10		REQ_VD_10			
REQ_VD_11		REQ_VD_11			
REQ_VD_12		REQ_VD_12			

For Help, press F1

Mon, 12, Nov 2012 7:46 PM

Figure 8: Generated Traceability Table

Traceability is best 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/or state machines. This is actually done as steps within the tasks shown in **Figure 5**.

Conclusion

Safety-critical systems are more difficult to develop than their non-safety-critical brethren. 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. Agile methods are a set of practices that improve both quality and productivity and can be employed in the development of safety-critical systems as well. 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 we discussed are the Initial Safety Analysis, Continuous Safety Assessment, Executable Requirements, and Traceability Analysis practices. These practices enhance the more usual agile practices by bring necessary rigor and completeness to the development process necessary for safety critical systems development.

For more information, visit:
http://www.ibm.com/software/rational/agile/embeddedagile/