

The Use of Modeling for Flight Software Engineering on SMAP

Alexander Murray, Chris G. Jones, Leonard Reder, Shang-Wen Cheng
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
818-454-0111
{atmurray, cgjones, reder, scheng}@jpl.nasa.gov

Abstract—The Soil Moisture Active Passive (SMAP) mission proposes to deploy an Earth-orbiting satellite with the goal of obtaining global maps of soil moisture content at regular intervals. Launch is currently planned in 2014. The spacecraft bus would be built at the Jet Propulsion Laboratory (JPL), incorporating both new avionics as well as hardware and software heritage from other JPL projects. [4] provides a comprehensive overview of the proposed mission.

Recently there has been increasing interest at JPL in using model-based techniques for systems and software engineering. In what is something of a departure from past practice in JPL flight projects, the flight software (FSW) is being engineered with a decidedly model-based approach, relying heavily on the Unified Modeling Language (UML) and System Modeling Language (SysML). In this paper, we will describe our applications of UML and SysML to most aspects of the flight software engineering effort. These include not only subsystem and software architecture expression and description, software design at all levels, requirements management and traceability, but also modeling of the hardware with which the software interacts, as well as the verification approach and implementation, the conceptualization and description of work processes and design constraints, and model checking.

We will describe our usages of modeling techniques for all of these activities, as well as problems and difficulties involved in our approach. We believe that overall effect of this integrated modeling approach will be a more reliable, robust, and maintainable FSW product, as well as a predictable development schedule and cost. Early indications are positive, as we will describe.^{1,2}

TABLE OF CONTENTS

1. INTRODUCTION	1
2. RELATED WORK AT JPL	2
3. THE SMAP FSW PROFILE	2
4. SUB-SYSTEM ARCHITECTURE	4
5. REQUIREMENTS AND VERIFICATION	7
6. SOFTWARE ARCHITECTURE AND DESIGN	12
7. STATE MACHINES	14
8. TOOLS AND PROCESSES	15
9. DOCUMENT GENERATION FROM MODELS	16

10. CONCLUSIONS AND ONGOING WORK	17
REFERENCES	18
ACKNOWLEDGEMENTS	18
BIOGRAPHY	19

1. INTRODUCTION

The UML is such a rich language that it can be used to describe anything that has structure or relationships or logic. This makes it apt for descriptions of anything from a long-term, high-level human activity to a very detailed specification of behavior of some particular software routine.

Our techniques encompass the use of capabilities provided by our SysML/UML tool (MagicDraw), which include code round tripping to and from C++, as well as extensions that we are developing using customization facilities, including document generation and model checking. We will also touch on capabilities that we have inherited and adapted to transform UML state machines expressed in UML into either scripts that drive system test scenarios, or into flight code.

A brief list of applications of these tools and techniques follows:

- (2) Architecture expression and documentation
- (3) Management of traditional requirements: maintenance, traceability.
- (4) Description of work processes and tools
- (5) Modeling of hardware components
- (6) Sharing of models among teams
- (7) Expression of design constraints, use of model checking to enforce them
- (8) Software design expression, documentation
- (9) Expression of detailed software requirements
- (10) Code generation from classifier model elements
- (11) Code generation from state machines
- (12) Verification planning and organization

¹978-1-4244-7351-9/11/\$26.00 ©2011 IEEE.

² IEEEAC paper #1479, Version 7, Updated January 5, 2011

- (13) Management and expression of verification scenarios
- (14) Definition of logic of test scenarios as behaviors
- (15) Generation of test scripts from state machines
- (16) Document generation from models
- (17) Profiling for all of the above

In this paper we will describe all of these applications of SysML and UML.

The profiling mechanism of UML allows the language to be extended and made more specific for a particular project or domain. We have created a profile to allow us to capture some of the concepts and information that are key to our task. Since our profile underlies much of our modeling work, we begin with a short description of the profile. As we discuss particular applications of the profile throughout the paper, we describe more detail about particular stereotypes as needed.

We continue into a description of architecture of the FSW subsystem, and how we describe it in UML and SysML. This leads into a discussion of the use of models to manage requirements and relationships among them. This leads to the topic of verification, and how we use modeling to plan and describe verification activities.

This is followed by a discussion of software architecture and design and our use of modeling techniques to generate and document them. This will include a few special topics, e.g. code generation from UML state machines.

We then briefly discuss the modeling of processes employed in the FSW engineering effort, followed by a discussion of document generation from the models.

We conclude with a discussion of benefits of an integrated modeling approach, as well as problems and disadvantages. Problems include difficulties in efficiently meshing an integrated model-based approach with a prevalent document-based approach, or inheriting software that was not developed or documented with model-based techniques, as well as tool limitations, such as the difficulty of sharing data between MagicDraw and the requirements management tool that is used on the project.

2. RELATED WORK AT JPL

We mentioned in the introduction that there is a great deal of interest in model-based engineering at JPL. The most important and comprehensive effort in this area within JPL is undoubtedly the Integrated Model-Centric Engineering initiative (see [11]). The IMCE initiative has taken on the ambitious task of examining all aspects of systems engineering as currently practiced at JPL, and

systematically developing methods and techniques for replacing or improving current practice with a model-centric approach. IMCE is also developing infrastructure to support projects in adopting these methods and techniques, including a set of ontologies and UML profiles.

IMCE has also produced a general and powerful document generation capability for use on MagicDraw models. Had this capability been ready six months earlier, we would probably have used it on our project.

In general, UML and SysML are becoming commonplace at JPL, particularly in software engineering. It is becoming more common too to see UML or SysML diagrams pop up in system engineering documentation. And some software projects are using UML extensively for software design. But we are not aware of any project at JPL that has consistently and comprehensively used model-based engineering techniques.

3. THE SMAP FSW PROFILE

A UML profile is a special kind of UML model whose purpose is to extend and/or constrain the UML language. This is useful to refine concepts in the UML language to better represent concepts in the problem domain at hand. The SysML language itself consists largely of a UML profile. Profiles consist of special model elements called *stereotypes*. Each stereotype refines one or more UML concepts (or more precisely, *metaclasses*). See [1] for a complete description of the UML profiling mechanism.

The SMAP FSW UML profile allows us to express concepts and details about them that are specific to our project. As shown in Figure 1, the profile is organized in packages according to the types of concepts and activities for which we need to define stereotypes.

Relationships to Common or Standard Profiles

Our profile is largely “home grown”, but it does leverage the SysML profile by making several of our stereotypes derive from the SysML Requirement stereotype. This is particularly useful for stereotypes that need a text tag and an ID tag, both of which the derived stereotypes inherit from SysML Requirement.

We used the OMG’s MARTE (Modeling and Analysis of Real-Time Embedded Systems) profile (see [10]) for a while, but we found scant tool support for MARTE’s analytical capabilities. Moreover, we found it an annoyance to have so many stereotypes that we never used complicating our modeling efforts (e.g. when selecting a stereotype from a pull-down, having to scroll through several dozens of stereotypes that we never used). So we discontinued using MARTE, and instead defined a half-dozen or so stereotypes to cover the concepts we needed.

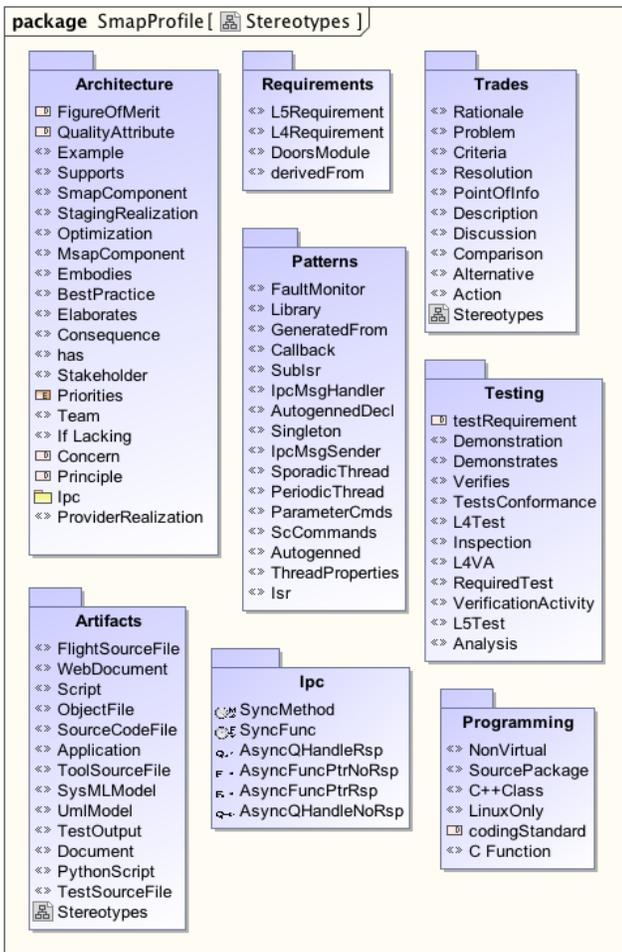


Figure 1 - The organization of the SMAP Profile

These are described below in the discussion of the *Patterns* package.

SMAP Profile Descriptions

This section, organized by profile package, provides a description of key stereotypes in our profile. We cannot discuss all of the stereotypes due to space.

The *Architecture* package contains stereotypes used in our FSW Architectural Description Document (ADD), which discusses *quality attributes*, *stakeholders*, and their *concerns*. These concepts are typical in discussions of architecture, but we are not aware of a well-established standard UML profile for modeling them.

So for example, the concept of a *Stakeholder* is a key one, and it typically represents a user of a system, or someone affected by the development or operation of the system in some way. As we began to analyze our stakeholders, we found that their concerns were characteristics of teams of people, rather than of individuals. Thus we decided to model stakeholders as teams rather than individual actors,

which led us to define the *Team* stereotype. We similarly introduced stereotypes *Concern* and *QualityAttribute* as stereotypes that extend the metaclass *Class* (being derived from SysML *Requirement*). Using these three stereotypes, we can model a team caring about specific quality attributes. Examples of applications of these stereotypes are

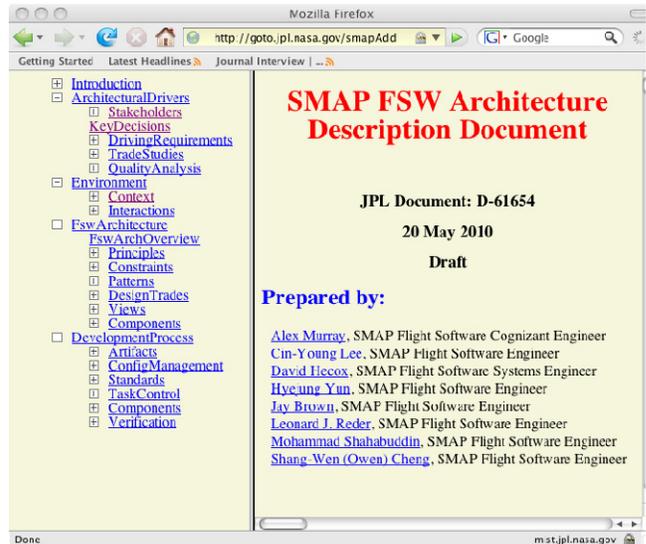


Figure 2 - The cover page of the Architectural Description Document (ADD)

shown in Figure 5.

In the *Requirements* package, the *L4Requirement* stereotype, derived from SysML's *Requirement*, is used to mark level 4 requirements imported into the model from IBM Rational DOORS [9] (used consistently for requirements at JPL, at least for level 4 and higher). Section 4 discusses the meaning of requirement levels 4 and 5 in the context of SMAP FSW. The *L5Requirement* stereotype, applicable to any metaclass, is not derived from SysML's *Requirement* as might be expected, to avoid some of the constraints on that stereotype that SysML imposes.

The stereotypes of the *Trades* package mark parts of a tradeoff study: a tradeoff generally has a few or more *Alternatives*, have *Criteria* to guide the decision, *PointsOfInfo*, and a *Resolution*. We express these in the model as *Comment* elements, which we mark with the applicable stereotype from the *Trades* package. We summarize trade studies with a single diagram containing nothing but Comments. This results in a crisp and easily-digestible summary of the trade.

The *Patterns* package contains stereotypes that are key for describing software design patterns. For example, a crucial concept in embedded and real-time software engineering is the *thread*, which is an independently-schedulable unit of control in the software, owning their own memory resources. At runtime, threads are the scheduled and

executed by the operating system, based on priorities assigned each thread by the software architect. Thus, the modeling of threads is key in architecting FSW. Threads can run in response to sporadic events, or they can run periodically.

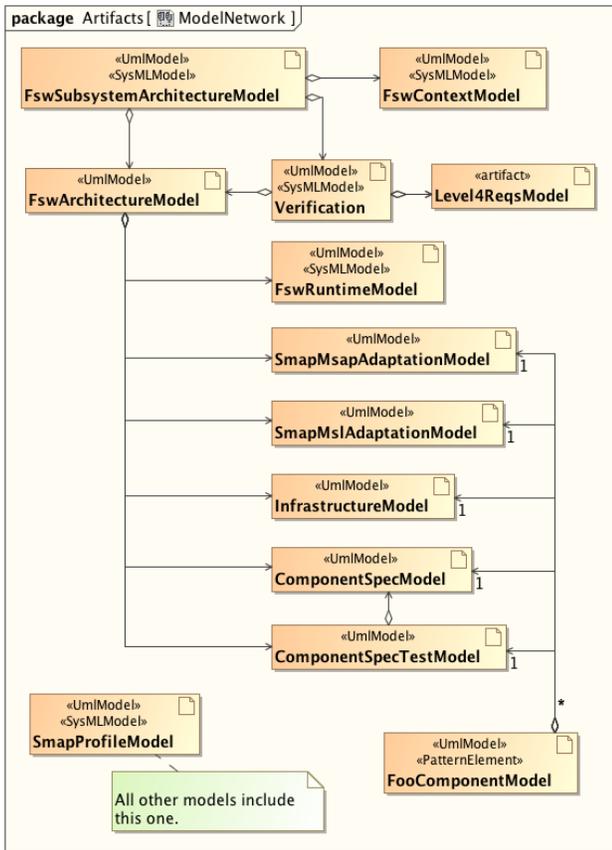


Figure 3 - The system of models making up the FSW architecture and design

We introduced stereotypes *SporadicThread* and *PeriodicThread* to mark classifiers in our model that represent threads. Both stereotypes are derived from *ThreadProperties*, and provide a useful way to identify, and search the model for, tasks in the design. *ThreadProperties* contains a priority tag, which represents a key property of a thread in embedded software design. The UML attribute *IsActive* on metaclass *Class* is not adequate because it doesn't capture priority or release style (periodic, sporadic). *Isr* (interrupt service routine), is another key concept in embedded software, and it is important to have these operations clearly identified in the design. Thus we defined the *Isr* stereotype to mark functions in our model that are responsible for handling interrupts from the hardware.

The *Ipc* package contains stereotypes that are applied to *connectors*: logical connections between software components, representing internal software communications channels in the implementation, in order to describe the style of communication over that connector. For example, a

connector that is implemented as a synchronous method call through an object is marked with the *SyncMethod* stereotype, while one implemented asynchronously with a queue handle for a response passed is marked with the *AsyncQHandleRsp* stereotype. These stereotypes help a reader understand how the software communicates internally among its components.

Our ADD describes various aspects of how we do our software engineering work, and the products of our work. This is the reason for the *Artifacts* package and the stereotypes contained therein. An example is the stereotype *WebDocument*, which we use in diagrams showing how our UML/SysML models are used to generate documents, the ADD being one example. Document generation is discussed in a subsequent chapter.

The ADD and models also describe verification, which includes the description of verification techniques, as well as tests or other verification activities and mappings between tests and requirements. We use the stereotypes in the package *Testing* for these purposes. Examples of application of these are given in Chapter TBD.

Our profile also includes the *Programming* package, which contains stereotypes that apply specifically to the implementation of the software (which is in C and C++).

Our profile was used extensively in all of our models, and these were the basis of the ADD. The profile then, was an important contributor to the clarity of the ADD.

4. SUB-SYSTEM ARCHITECTURE

Just as there is increasing interest in model-based engineering at JPL, there is also a push to concentrate and focus more on system and software architecture in our work.

Our project is probably the first at JPL to have held a specific review dedicated to FSW subsystem and software architecture, and this will likely be standard practice moving forward.

Our architecture review, while guided with the use of viewgraphs, was focused on the ADD itself; the viewgraphs were full of hyperlinks into the on-line ADD, and a web browser was used heavily in the presentation.

Thus, the reviewers spent the majority of their time examining and discussing diagrams in the ADD, rather than looking at viewgraphs. This seems to be a hallmark of model-based engineering: the engineering products themselves are reviewed more than viewgraphs about the products.

The concept of “Architecture” which we’ve employed includes not only the structural and behavioral principles and patterns of the software itself, but also an analysis of the context of the FSW as a subsystem in the larger flight system, from both operational and programmatic viewpoints. We used MagicDraw to generate our ADD in the form a web document with a navigation tree in one frame, and the content in another (document generation is discussed below). The outline of the document, shown in Figure 2, gives a good overview of the scope and content of our architecture, and of our uses of modeling.

One of the difficulties associated with this emphasis on modeling is that models can become large and unwieldy. Fortunately MagicDraw allows one model to “use” another, in the sense of importing another model as a read-only library model. We exploited this capability to partition our

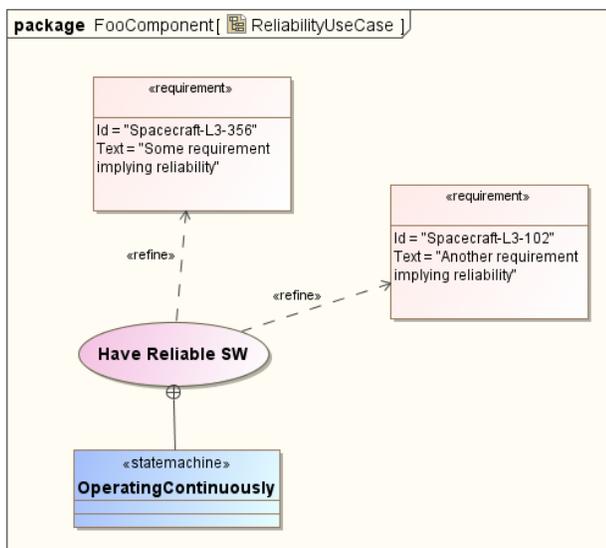


Figure 4 - A use case, elaborated by a state machine, refining requirements

architectural model into a collection of sub-models addressing particular aspects of the architecture, as shown in Figure 3. In that diagram, we use composition to model the inclusion by one model of another, as the *FswArchitecture* model does the *FswRuntime* model.

The *FswSubsystemArchitectureModel* is the root model from which the ADD is generated. It contains modeling of FSW context, quality attributes and stakeholders. The *FswArchitectureModel* contains the software high-level design (also via inclusion): the *FswRuntimeModel* contains the structural decomposition of the FSW at runtime, the *InfrastructureModel* describes a set of software support classes used throughout the rest of the FSW. The *ComponentSpecModel* specifies the interfaces among all of the components. *FooComponentModel* stands for any of several models of the design of individual components. We have twenty-some individual component models.

Architectural Drivers

Our ADD includes an exposition of key and driving requirements. This is organized with a set of Use Cases, which serve to tie a related set of higher-level requirements together, and then to elaborate the behavior, at a conceptual level, showing the FSW subsystem achieving that Use Case. We say higher-level to emphasize that we used level 3 requirements – which are requirements on the entire spacecraft, not just the FSW – for this exposition of driving requirements, in part because the higher level of abstraction was appropriate for this document, but also because the FSW subsystem requirements (at level 4) were not yet ready.

An example of one of these use cases is shown in Figure 4: the use case groups and refines a pair of requirements that have implications on FSW design. The state machine *OperatingContinuously* (shown only as a classifier box) specifies characteristics of FSW operation that enable it to run reliably for indefinite periods of time.

Architectural Tradeoffs and Decisions

In exploring and documenting tradeoffs that led to key decisions, we sometimes used modeling, but more often used simple text, depending on the type of decision. A lower-level tradeoff between two software design alternatives would inevitably depend heavily on UML, whereas a higher-level issue with programmatic implications would be expressed in English, usually as the documentation attribute of a UML package.

But for any type of trade, we found it useful to have a compact and indivisible summary of the trade and decision, in a way that was easily shared. For this we used a UML diagram containing only Comment objects, tagged with one of the special stereotypes Description, Alternative, Discussion, Action, PointOfInfo, or Resolution, and color-coded. Using diagrams as containers for comments only struck some of the reviewers of the ADD as odd, but nonetheless useful.

No ADD is complete without a discussion of stakeholders, concerns, and quality attributes. As shown in Figure 5, an excerpt of a diagram showing a few of these concepts, we modeled stakeholders as teams (rather than as individuals), tagging *Class* instances with the *Team* stereotype. *QualityAttribute* and *Concern* are both derived from the SysML *Requirement* stereotype, which gives them the text field tag.

Quality Attributes, Stakeholders, and their Concerns

We displayed quality attribute priorities by simply displaying the QA’s, color-coding (red meaning high, yellow medium, and green, low) on a diagram.

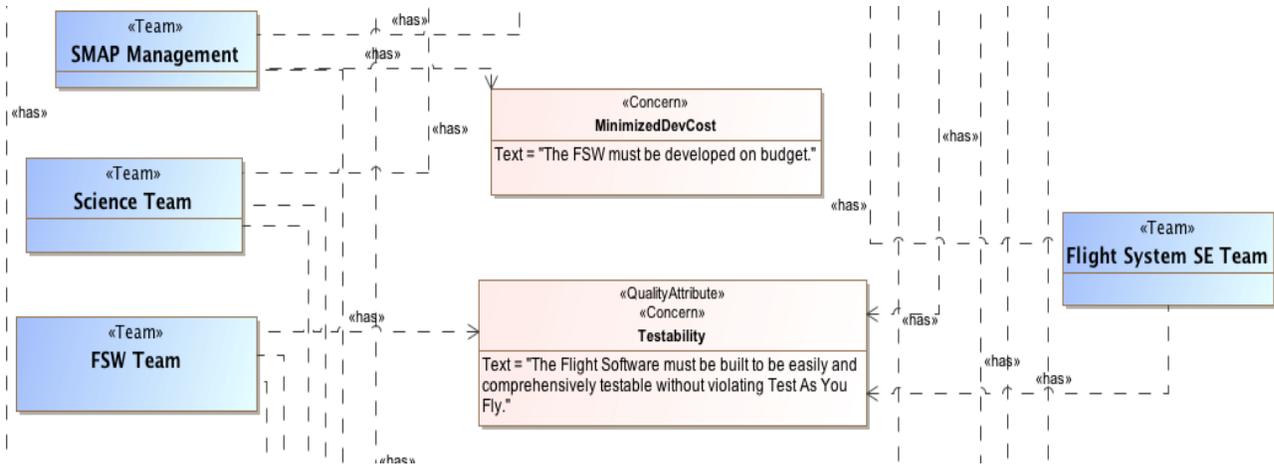


Figure 5 - Part of a diagram depicting stakeholders, their concerns, and quality attributes

Another piece of the quality analysis in the ADD is a subsection called *Realization*, and this has the goal of presenting an argument that our architecture achieves the QA's and meets the concerns of the stakeholders.

We defined a viewpoint consisting of a custom diagram called a *Success Tree Diagram*. Inspired by the concept of a fault tree, in which the possible paths to the fault are analyzed, the success tree shows the QA at the root, and shows all of the things that must be done, or must be true, in order for that QA to be achieved. As shown in Figure 6, the QA is shown being supported by design principles defined in the architecture (in other Success Trees, other

architectural elements are also shown supporting the attainment of QAs, including *designConstraints*, *testRequirements*, patterns, or other architectural features).

We also defined a stereotype to identify *Figures of Merit* – criteria to measure how well the FSW meets the QA. An example is the *AssessabilityOfImpact* figure of merit, shown in yellow in Figure 6. We plan to take benchmarks of these figures at major milestones.

The Environment Section

The concept of the *Environment* appears in [3], and this section of our ADD describes the FSW subsystem's context in the rest of the flight system, using a series of context

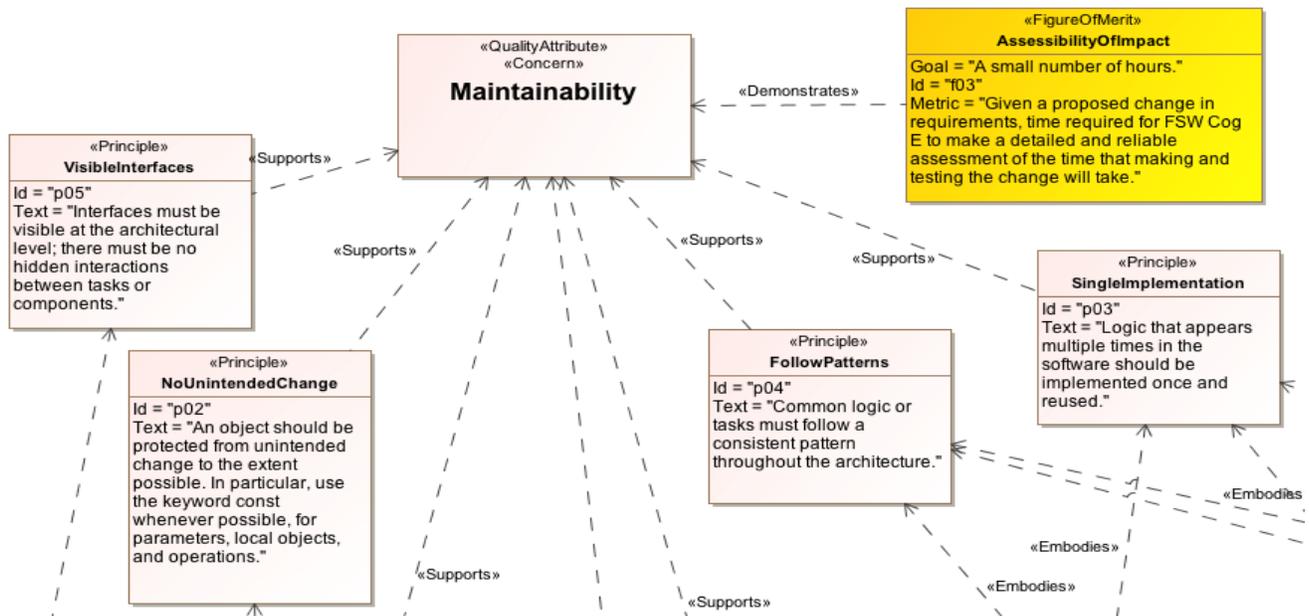


Figure 6 - A Success Tree Diagram excerpt

diagrams.

An example that is similar to one of these context diagrams, shown in Figure 7, in which SysML blocks represent hardware components. In SMAP we maintain a separate model of the hardware components of the spacecraft. This hardware model is used by the software models as needed to represent the hardware and the interrupts and data flows between it and the software. The hardware models are also used to represent many of the details of the hardware components themselves and their connections to each other – a topic for another paper.

5. REQUIREMENTS AND VERIFICATION

The use of DOORS is pervasive and embodied in institutional procedures at JPL. DOORS enables the expression of textual requirements and the linking of requirements from parent to child and vice versa. It also supports enumerating test cases and other verification activities, and linking requirements to them.

SysML and UML provide a much richer environment for developing, managing, describing, and tracing requirements. They provide all of the capabilities of DOORS, as well as the capability to elaborate and refine requirements with various behavioral and structural diagrams, to map requirements to design elements, and to specify the behavior of verification scenarios (rather than just name and summarize them).

Perhaps most importantly, UML can be used to express

expectations on system interfaces and behavior (i.e. requirements) in more concise and precise ways than with English statements, especially in the case of detailed software requirements. Our detailed software requirements, which we call level 5, are generally expressed as UML model elements, not necessarily as text. The level 5 requirements reside only in our models, and not in DOORS. The models are maintained in the same configuration management system as our source code.

The authoritative repository for the requirements on the FSW subsystem (level 4) is necessarily DOORS. We replicate the requirements in SysML, using MagicDraw; DOORS can export requirements in spreadsheet form, and MagicDraw can import them in that form. The level 4 FSW requirements are in the model in the form of Class instances tagged with our *L4Requirement* stereotype. This is derived from the SysML Requirement stereotype but contains an additional attribute that identifies the subsection to which the requirement belongs in DOORS.

Level 5 Requirements

Level 5 detailed software requirements are levied on parts of the FSW subsystem. They can be at the level of a component (which is the first level of decomposition of the software; something like the level of an application), or at the level of an individual class or function.

These requirements tend to fall into two types: interface requirements and behavioral requirements. A typical interface requirement is for a software component to provide some interface for use by other components.

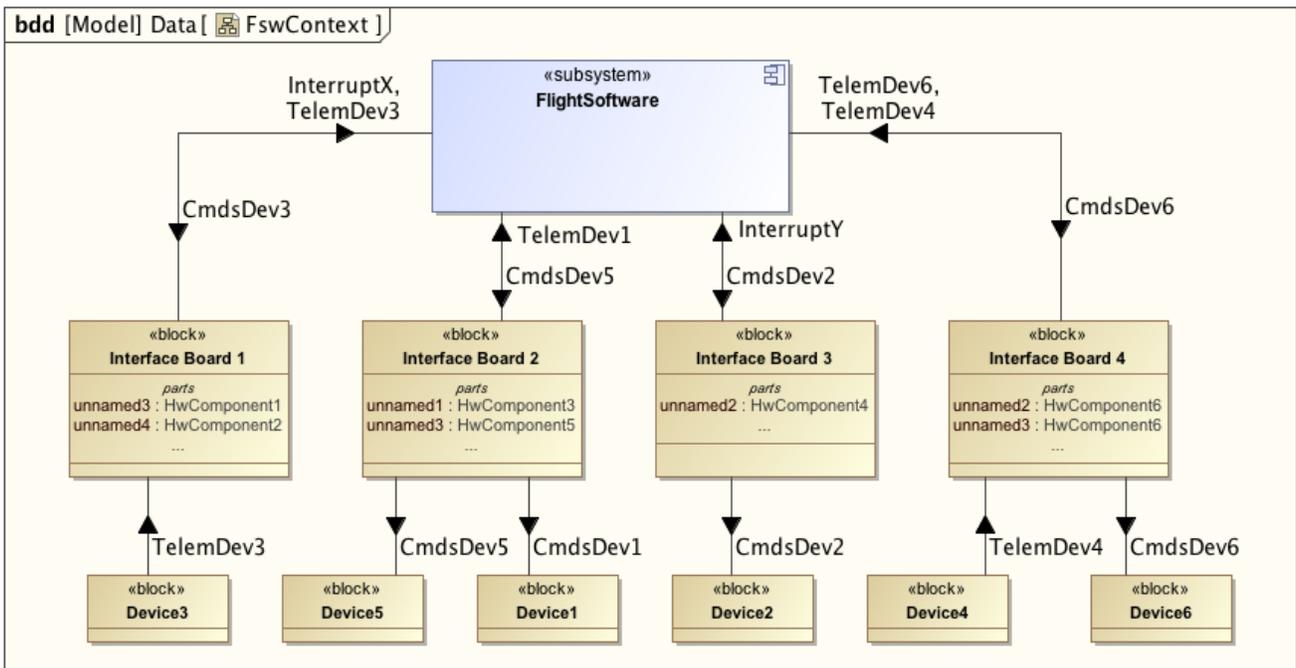


Figure 7 - A sample flight software context diagram

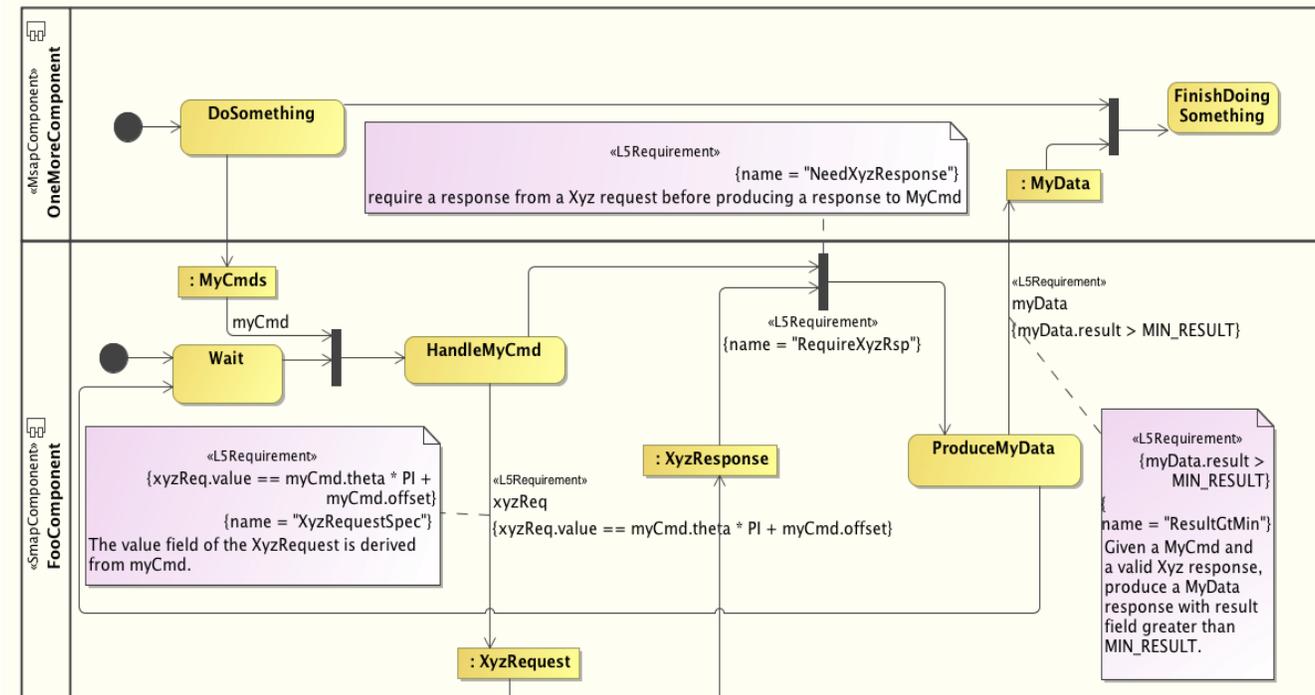


Figure 8 - Two examples of UML constraints as level 5 requirements

Another typical interface requirement is that the implementation of a given interface operation enforce constraints on the arguments to the function. Examples of these are shown in Figure 16. The requirements are expressed by the “providing” relation in the component diagram; for example, the component *SomeComponent* is required to provide an implementation of the interface *AnInterface1*. The providing relation (the connector from the component to the “lollipop”) is tagged with the *L5Requirement* stereotype.

The interfaces themselves, as well as the component diagram showing the component providing it, are marked as *L5Requirements*: the existence of the interfaces, independent of providers, is also an interface requirement.

We couldn’t use the SysML *Requirement* stereotype for this, because that can only be applied to classifiers. This is one reason we needed to define an *L5Requirement* stereotype: we apply it to associations, diagrams, interfaces, UML constraints, connectors, and other model elements.

Behavioral requirements express a constraint on logic, calculations, transformations (of inputs to outputs), or timing. A few examples of computational requirements are shown in Figure 8, which contains three requirements. One requirement states a relation between input *myCmd* and output *xyzReq*, the second a requirement to wait (at the join node stereotyped *L5Requirement*) for a response before proceeding. The third requirement is a constraint on the minimum value of the output *myData.result* (which is the response to the input *myCmd*).

The actual requirements in these three cases are the two control flows (including the UML Constraints applied to them) and the join node that are tagged with the *L5Requirement* stereotype. There is no UML Constraint on the join node, but its existence at that position in the diagram constrains the logic of the activity: the logic must wait for a control flow from *HandleMyCmd*, and for a data flow of type *XyzResponse*, before proceeding.

The *Comment* elements associated with the two flows are also constrained by the *Constraint* on the respective control flow. The *Comment* associated with the join node requirement has no UML *Constraint* on it, but it is annotating an *L5Requirement*. These tagged comments can be displayed in other diagrams, and in fact they appear in diagrams depicting the logic of the test programs that test the requirements.

It’s important to note that the requirement in each case is not the text body of the comment; the text in the comments is analogous to an informational paragraph accompanying a “shall” statement in a traditional requirements document.

Performance requirements are an important type of behavioral requirement. Examples of these are shown in Figure 9. The timing requirements are expressed as a UML *DurationConstraint*, also tagged as a level 5 requirement. Other examples of UML model elements tagged as level 5 requirements appear in Figure 12.

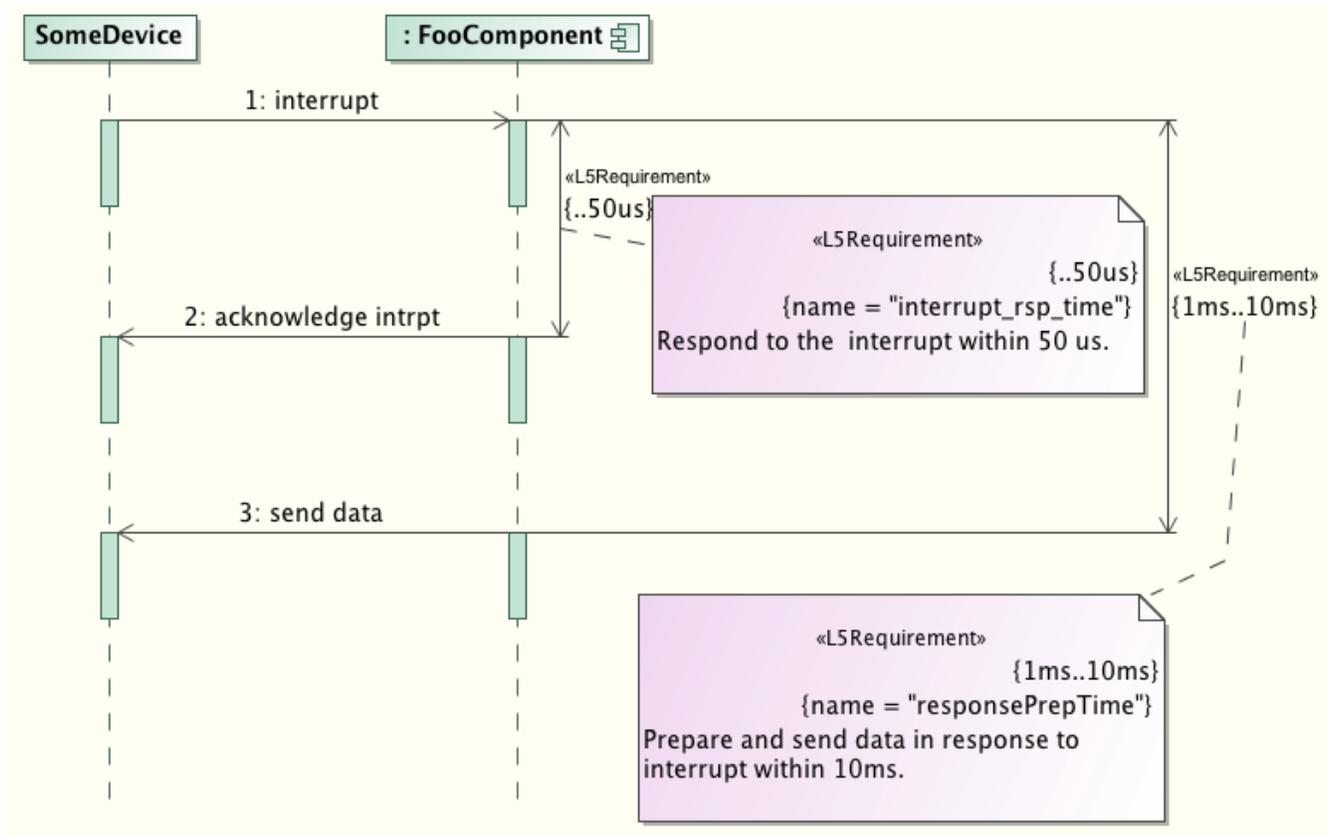


Figure 9 - Two level 5 performance requirements

Verification Scenarios

Verification takes place at the FSW subsystem level (verifying level 4 requirements), and at component or sub-component level. Component or sub-component level testing is done both for level 5 requirement verification, and to satisfy a self-imposed requirement that every element of the software design be mapped to a test, as a way of achieving high code coverage in testing.

Subsystem (Level 4) Verification

At level 4, verification scenarios are of three types: tests, analyses, and inspections. We model tests as UML behaviors, primarily as state machines. We use state machines because we have the capability to generate an implementation of the state machine in Python (more on this below), which allows us to use a test environment Python API that provides the capabilities of sending commands to the FSW and receiving telemetry from it. An example of one of these state machine scenarios is shown in Figure 10. The function calls in the entry actions, e.g. `sendCmd()`, are utility Python functions that interface with the ground system software to send commands or retrieve and parse telemetry. These functions are used by all scenarios.

Analyses and inspections are human activities, and we model them as UML *Activities*. Having them in the model allows tracing them to level 4 requirements.

Our *Analysis* stereotype is derived from the SysML *Test Case* stereotype. We tag our level 4 test cases, whether defined as state machines or other behaviors, with that SysML stereotype.

Component and Sub-Component Level Testing

Verification at this level is almost completely done by testing (as opposed to analysis or other methods). These tests are white-box tests, designed and implemented just as flight software is, and run as applications (under Unix) or functions (under VxWorks).

The logic of white-box tests is specified using UML behaviors – state machines, activities, or interactions in sequence diagrams. State machines are preferred, since we can use them to generate C++ code to execute the test. White-box tests, then, can appear in the model as classes (that own behaviors), or as behaviors (state machines, sequences, etc). They are not tagged with stereotypes to identify them: our source tree is split at the top between

flight and test, so the package membership of a classifier in the test tree identifies it as a test.

To ensure complete verification at this level, our architecture model specifies a set of testing methodology requirements expressed using the *testRequirement*

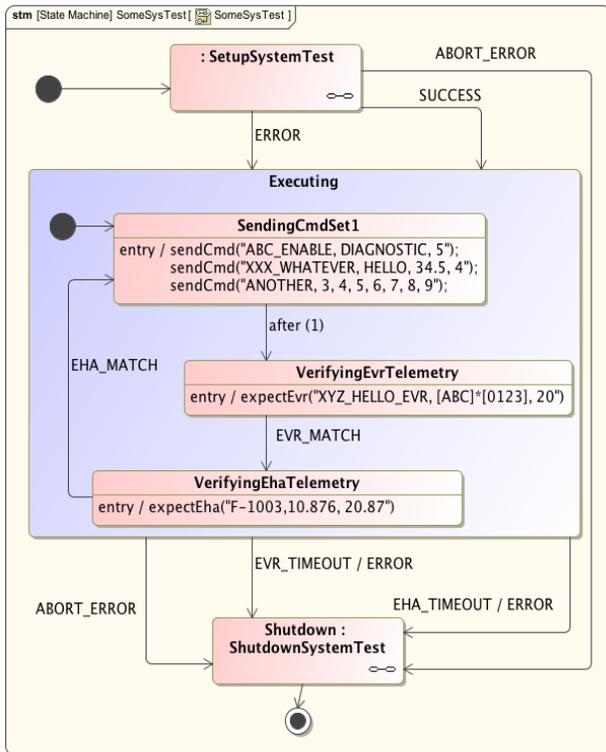


Figure 10 - a FSW subsystem-level state machine test scenario

stereotype from the profile. A few examples: “Every *Interface* must have a test program that can verify that any given implementation of the interface conforms to the *ProtocolStateMachine* for that interface”, or “Every flight classifier must be mapped to at least one test class via the *Verifies* relationship. Classifiers include *Class*, *Interface*, and *Component*”, or “For every method of every *Interface* implementation, there must be an adaption of a required test that verifies complete and correct parameter checking.”

To help achieve these requirements, we define design patterns for testing. An example is shown in Figure 11, which shows a pattern for designing reusable tests. In the figure, *SomeReusableTest*, defines an interface *SubjectAdapter* that allows it to communicate with the subject. To be able to run the test for a given flight class, the programmer has only to implement the *SubjectAdapter* interface for that particular class.

A specific example of a reusable test is the *ResourceUsageTest*, designed to ensure that if a resource becomes unavailable (full), the software should behave

predictably. The *SubjectAdapter* interface in that case, has a *exhaustResource()* operation, which the test will invoke.

If the test is adapted for, say, a class that implements a container, then *exhaustResource()* will mean filling up the container, where as implementing *exhaustResource()* for a communication channel might mean causing the channel to go offline.

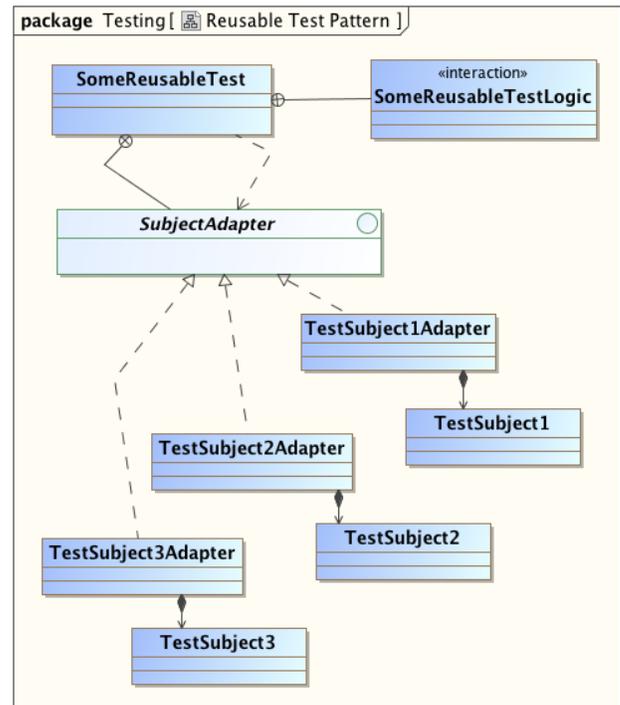


Figure 11 - a pattern for the design of reusable tests

Traceability

Institutional standards require the tracing of requirements to the parents from which they’re derived, and to some sort of verification activity (a test or analysis, for example). For detailed software requirements, an additional mapping from requirements to the design element(s) that satisfy them is required.

The trace of level 4 FSW requirements to parent level 3’s (and to other types of parents, such as Interface Control Documents), is managed in DOORS. The other traces are done with UML and SysML relationships in the models. Level 4 requirements are traced to level 5’s using our profile’s *derivedFrom* stereotype.

Because the SysML *derivedFrom* stereotype requires that both participants in the relation be SysML *Requirements*, and our level 5 requirements are not, we cannot use the SysML relationship. This mapping technique is shown in Figure 13.

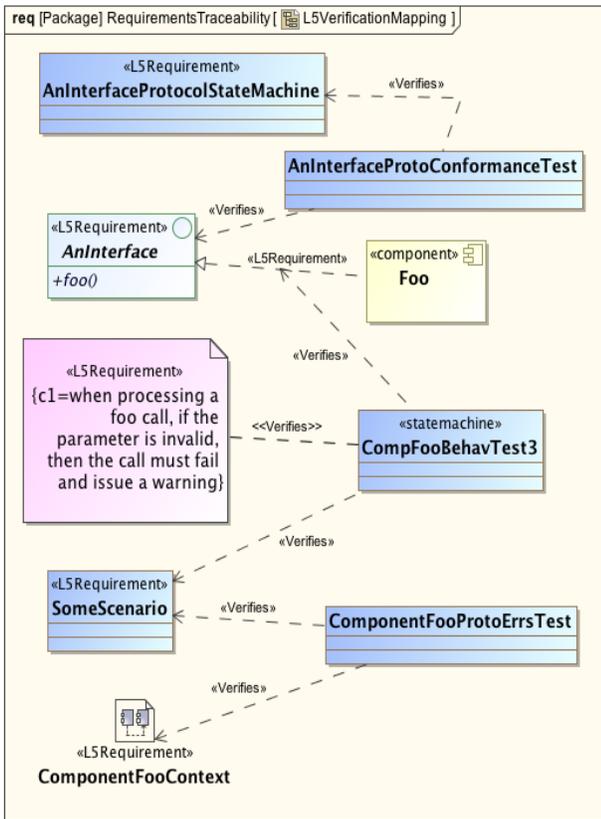


Figure 12 - The trace of white-box tests to level 5 requirements

The trace of test cases and analyses to level 4 requirements is done similarly, as shown in Figure 14. For this mapping, we use the SysML *verify* relationship.

The mapping of white-box, component- and subcomponent-level tests to level 5 requirements is done using our *Verifies* stereotype, as shown in Figure 12. Again, we could not use the SysML *verify* relationship because of constraints on the related elements.

All of these traces done in MagicDraw can be compactly represented in a matrix. The example shown in Figure 15 embodies the relations depicted in the mappings of Figure 14. MagicDraw can export these matrices as comma-separated value files. This allows a relatively simple export of the list of level 4 verification scenarios, along with lists of linked requirements, back to DOORS.

Tracing Level 5 Requirements to Design

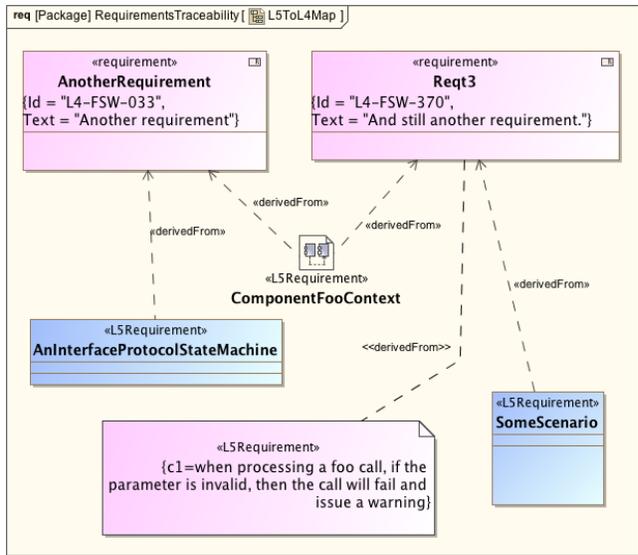


Figure 13 - Mapping of level 5 requirements to level 4's

Since level 5 requirements are expressed as constraints on elements of the design, such as Component interfaces, diagrams depicting the behaviors of Components and other elements of the FSW implementation, the association of these level 5 requirements to the design is clear. We haven't seen a need or benefit in extracting some more explicit trace of level 5 requirements to design.

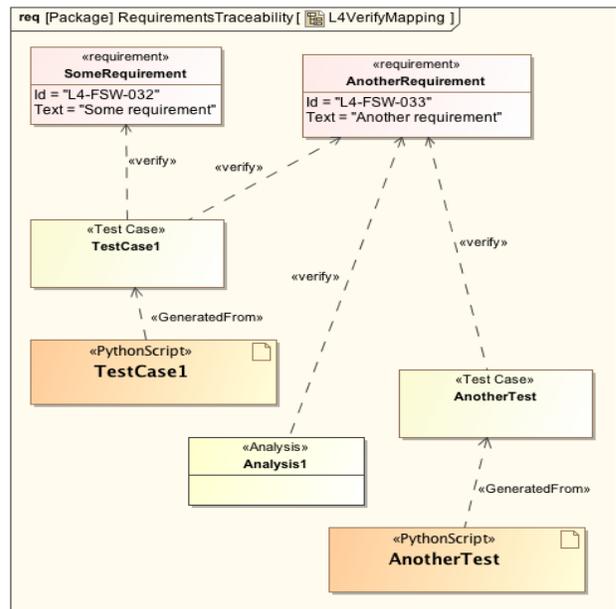


Figure 14 - The mapping of level 4 requirements to scenarios

6. SOFTWARE ARCHITECTURE AND DESIGN

We distinguish software architecture from subsystem architecture as being solely concerned with the structure and behavior of the software. This is a narrower concept of architecture than *subsystem* architecture, encompassing as it does programmatic context, stakeholders, etc. Software architecture does include guiding principles, design and test constraints and requirements, patterns, and detailed software requirements. We've discussed two of these concepts (*testRequirements*, and detailed software requirements, already).

Guiding Principles

Our software design is guided by a set of principles, expressed in the architecture model as classes tagged with the stereotype *Principle*, which is derived from SysML *Requirement*, which lends them name and text properties. Here are a few examples:

Least Visibility: An object should be visible only in the scopes where it is needed

Follow Patterns: Common logic or tasks must follow a consistent pattern throughout the architecture

Minimize Dependencies: Keep dependencies among packages to a minimum. Dependencies should be avoided unless necessary.

	L4-FSW-033 AnotherRequirement	" " FuncReq1	FuncReq2	" " PerfReq1	L4-FSW-370 Req3	L4-FSW-219 Requirement4	L4-FSW-032 SomeRequirement
ScenarioGroups	5				1	1	1
Analysis1	↗						
AnotherTest	↗						
FooBarTest					↗		
SomeSysTest							
TestCase1	↗						↗
TestCase2Activity	↗						
TestCase2Interaction	↗						
TestCase4Interaction							↗

Figure 15 - Trace matrix of level 4 requirements to tests

The advantage of having them in the model rather than a text document is that they can be placed in various

diagrams, e.g. in support of a design decision. A few of these appeared in the Success Tree diagrams shown above.

Design Constraints

Design constraints are similar to principles, but they are more specific, and intended to be specific enough to be expressible in UML's Object Constraint Language (OCL), though we haven't done that. We handle constraints similarly, marking them in the model with the SysML *Requirement*-derived stereotype *designConstraint*.

Here are a few examples:

Interface Protocols: Every interface must have an associated *ProtocolStateMachine* that specifies how that interface may be used.

Interface Ops Virtual: Every operation of an interface must have the *virtual* property set to true.

Circular Dependencies 1: The source package *CompileTime::Flt* may not depend on source package *CompileTime::Test* (dependencies considered recursively).

No Multiple Inheritance: A classifier may be the source of at most one *Generalization* relationship

These constraints are intended to prevent programming errors, or to preserve important architectural properties (such as the capability to build and load components individually – impossible if certain circular dependencies arise). The last item forbidding multiple inheritance is directly derived from a C++ sub-setting coding standard.

These design constraints (there are approximately 30 of them), are amenable to automatic checking by analyzing the model of the software design.

Design Patterns

Our architecture model contains several packages of design patterns. Among the most important ones are: Component definition and specification, initialization, inter-task communication, thread management, sharing data between threads, state machine implementation, handling ground commands, and white-box testing.

Our usage of UML for software design patterns is not unusual. We use the stereotype *PatternElement* to clearly mark model elements that are not part of the actual design, but exist only to illustrate a pattern or principle. We use several of the types of UML diagrams available, and many of these diagrams are too complex to fit in a paper format very well. But we can show a few meaningful examples.

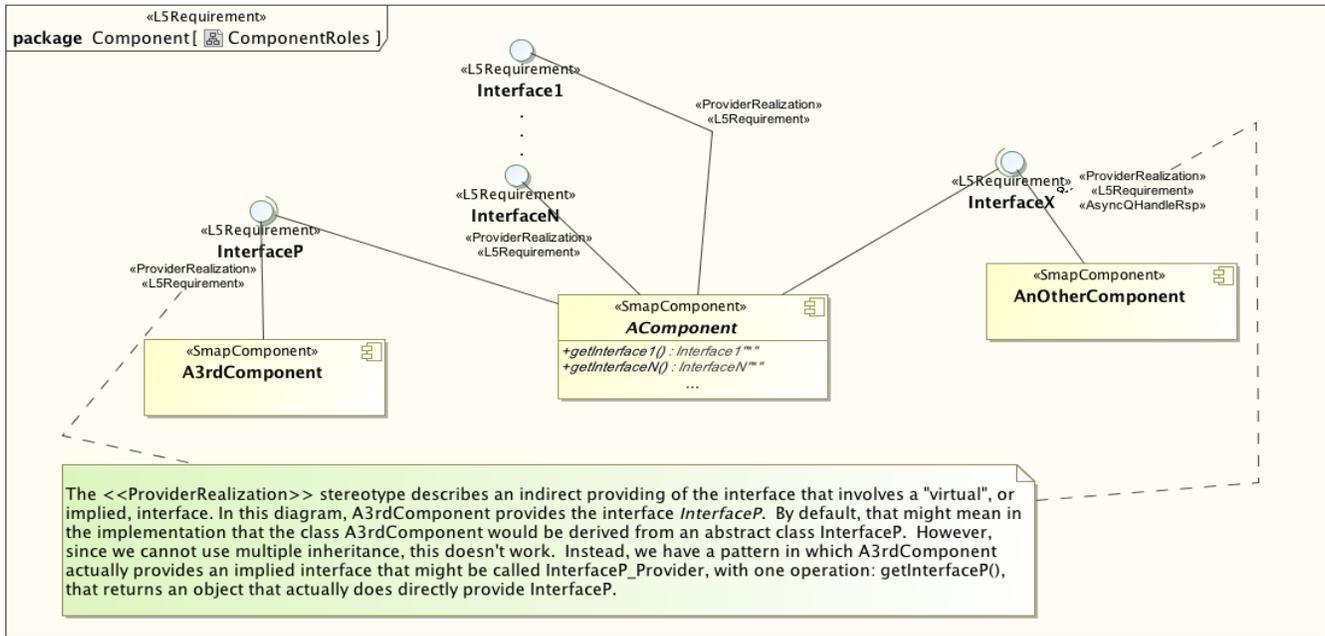


Figure 16 - the pattern for specifying the role of a Component

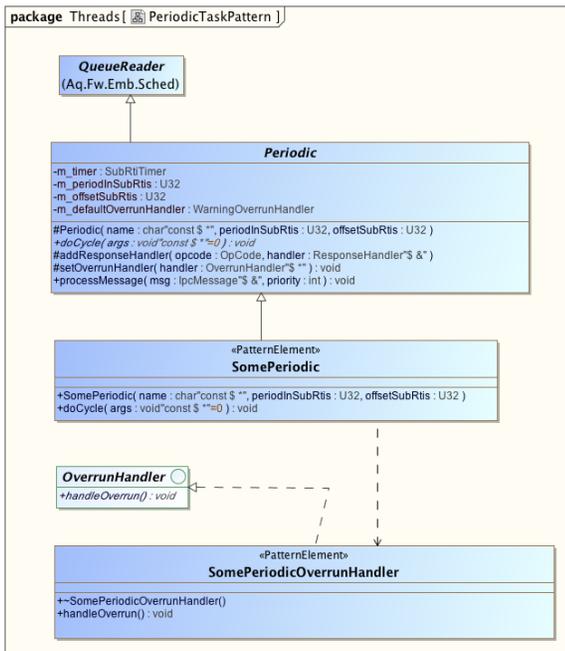


Figure 17 - The classes involved in managing a periodic task

The first of these is the pattern for defining the interface specification, or role, of a Component in the system. The top-level depiction of the pattern for this is shown in Figure 16. The Component is the top-level decomposition concept of the FSW subsystem, both at compile time (source level), and run time. Components are independently compilable and loadable.

The diagram defines the role of *AComponent* as providing the interfaces *Interface1* through *InterfaceN*. The provider relationship is tagged as an *L5Requirement*, since it is an important interface requirement on the component. We do not treat a *Component's* usage of an *Interface* as a requirement at the interface specification level. Requirements for a Component to use an interface appear first in specifying the behavioral requirements for a Component.

We use the term *Interface* to mean what UML means by *Interface*: a *Classifier* that has no *Properties* and only abstract *Operations*. Our understanding of the provider and user relationships between Components and Interfaces is also the UML specification's version of these concepts. In our implementation, *Interfaces* are mapped to abstract C++ classes, but this doesn't affect our usage of *Interfaces* in the model.

Yet another key pattern is the management of threads that need to run at a regular period. Figure 17 shows how the logic of arranging to wake up periodically is handled in an abstract base class, *Periodic*, and the application for a specific context is accomplished by making a derived class (*SomePeriodic* in the diagram), and providing an implementation of the abstract *doCycle()* operation.

The classifiers shown in Figure 17, except for *OverrunHandler*, are *Classes*, not *Interfaces*. These classes

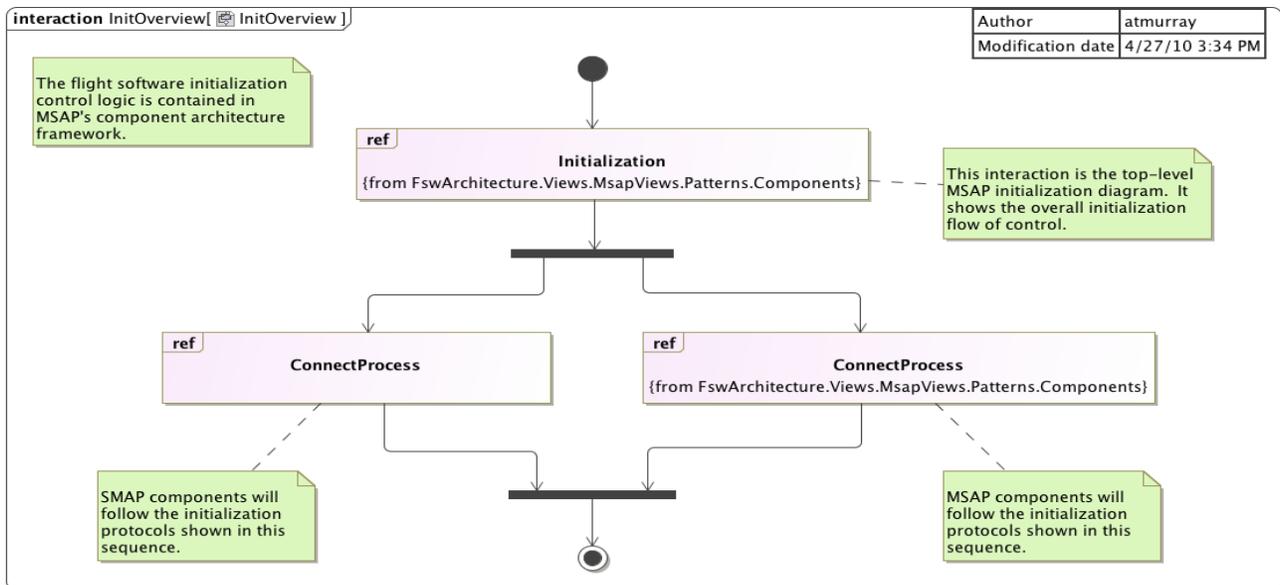


Figure 18 - a high-level view of the component initialization sequence

are utility elements for internal use in Components that need to own and manage threads; they do not form part of the interface of any Component.

Still another key pattern is the implementation of state machines. This is described in the next section.

Another key behavioral pattern is the initialization sequence in which every Component participates. An overview is shown in Figure 18. This sequence involves the use of a set of interfaces that all Components implement. Each of the referenced behaviors is in turn interaction overviews, ultimately resolving into UML sequence diagrams too detailed to show here.

7. STATE MACHINES

The JPL StateChart Autocoder (see [7]) is a software development tool that auto-generates programming language implementations of UML *StateMachines*. The Autocoder can be used to produce flight-ready C++ code, or Python code that provides a graphical interface for the depiction and control of the execution of the state machine. UML StateMachines are routinely used by systems and software engineers for documentation and as a tool for analyzing and simulating system behavior.

In order to unify and streamline the translation of state machines into code so that the process is less error-prone and costly, efforts to leverage this widely accepted design notation culminated in an initial version of the JPL StateChart Autocoder. This initial version was first conceived and utilized as part of the Space Interferometry Mission (SIM) project, and later retrofitted for Mars Science Laboratory (MSL) flight software. Other projects

that used early variants of this tool included the Advanced Mirror Development and the Electra-Radio projects. Prior flight software lessons taught us that, once such an autocoding tool was in place, few software defects were found that were due to coding errors. Errors were mainly attributed to design errors. Positive SIM experiences motivated further development of the tool as well as process.

The second-generation JPL StateChart Autocoder improves upon the initial work with an extensible plug-in architecture that incorporates a number of design patterns and new technologies for flexibility and robustness. A reworked UML meta-model implementation enables support for any UML XMI (XML Metadata Interchange) model file. A template-based generation technique facilitates flexible implementations for the resulting code. Code is generated using a state-chart implementation pattern based on the Quantum Framework for embedded systems, which boasts efficiency, robustness, and support for many platforms. A combination of Java interfaces, annotations, reflection, and design patterns facilitates modifiability and extensibility, and a sizable suite of test cases ensures robustness.

Currently, the second-generation StateChart Autocoder generates C, C++, Python, and *Promela*³ (see [TBD]) code from UML StateMachine models. It allows rapid prototyping of executable, verifiable state behavior. In particular, step-by-step state transitions and event communication between multiple state machines can be visualized. Certain concurrency, liveness, and safety properties can be verified using *Spin* (see [TBD]). Finally,

³ Promela is a model specification language used to create models for analysis of real-time and concurrent software systems, for use with the SPIN tool.

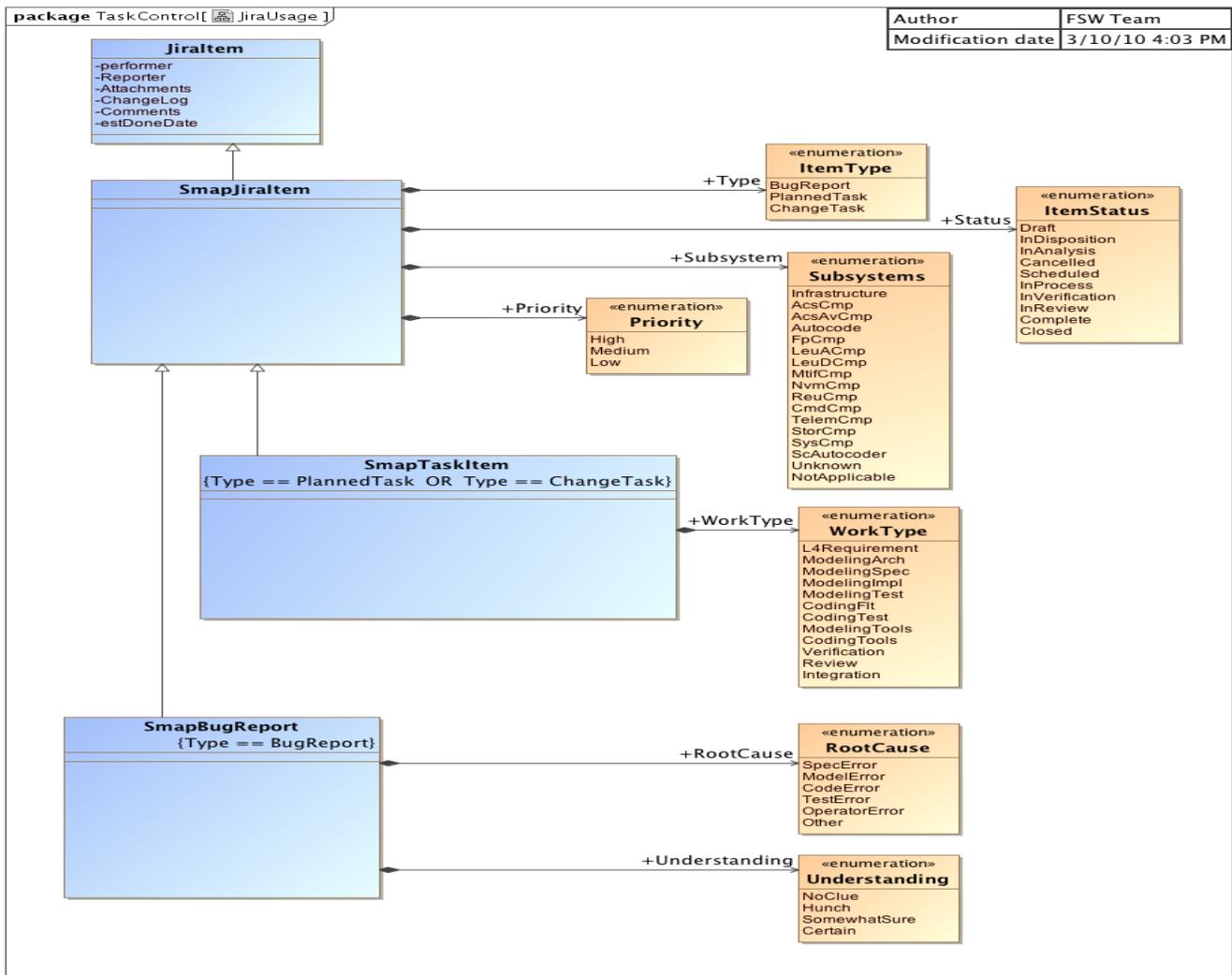


Figure 19 - the attributes of a JIRA item

more often than not, state machines do not fully specify all the behavior of the described module, so the Autocoder facilitates the “fleshing-in” of StateMachines with manual flight code. The Ares-1 requirements validation project and SMAP flight project both fund and use this tool. An example of the application of the Autocoder, which is being upgraded for SMAP, is shown in Figure 10. The Autocoder generates a Python implementation of a level 4 test scenario.

8. TOOLS AND PROCESSES

We’ve also used modeling to describe our work process. In the interest of space, we will shown only one or two examples of these applications of diagrams, and describe a few more.

Structural diagrams representing artifacts are helpful in describing inputs and outputs of a team’s processes. One such diagram shows all of the sources of FSW

requirements. These include level 3 requirements, various design description documents produced by flight systems engineering, as well as ICDs. All of the inputs are represented as UML artifacts.

Activity diagrams are well suited to describe human processes, as they can be used to show not only the actions taken by the actor, but also outputs and inputs. They can also show parallel actions of multiple actors. For example, we have a diagram showing a software engineer going through the process of engineering a Component, interacting with leadership and systems engineering. It shows inputs, outputs, and gates. This diagram is too large to include here.

We use a highly configurable issue tracking tool, JIRA, for both anomaly reporting and planned work management. We went through a process of specifying the configuration of this tool. We specified the types of issues and attributes of each in the class diagram shown in Figure 19.

The tool also allows the configuration of the life cycle of an issue. We specified that in the state machine shown in Figure 20. The configuration was done by an external system administration group, and these diagrams helped us quickly and clearly communicate to them what we needed in our configuration.

9. DOCUMENT GENERATION FROM MODELS

The generation of well-organized and usable documents from MagicDraw models is somewhat involved. MagicDraw has built in generators of web documents, but the generated documents include every model element, producing a web document that is very handy for reference and internal use in the FSW design team, but not at all suitable for documentation of the software architecture and design.

We experimented with a few different approaches before settling upon the one we've taken. One involved having an external model of the document that made reference to annotated elements in the target model. In the end we opted for a somewhat simpler approach, which consisted of modifying *Velocity* templates provided by MagicDraw for the generation of web documents.

MagicDraw's templates provide a library of macros for traversing the model, finding elements, and examining their attributes, as well as for producing Hypertext Markup Language (HTML) text from model elements. The primary

problem we had to solve was selecting which parts of the model should go in the generated document.

We solved this problem by annotating our models with enumeration definitions that specify the selection and order of elements in the document. (The name of the enumeration is a parameter to the generator, so that the same model, or sections of one, can appear in various documents.)

For example, the FSW Subsystem Architecture model has a root package called *FswSubsystemArchitecture*. In that package, it has sub-packages *ArchitecturalDrivers*, *Environment*, *FswArchitecture*, *Introduction*, etc., and a package diagram called *Architecture Overview*. When we generate the ADD, we want these elements to appear in this order:

- Architecture Overview
- Introduction
- ArchitecturalDrivers
- Environment
- FswArchitecture

To achieve this, we define an enumeration named *ArchDocOrder* in the top-level package (*FswSubsystemArchitecture*), and the enumeration literals have exactly the names and order of the elements we want to see.

We apply this technique recursively: each sub-package also

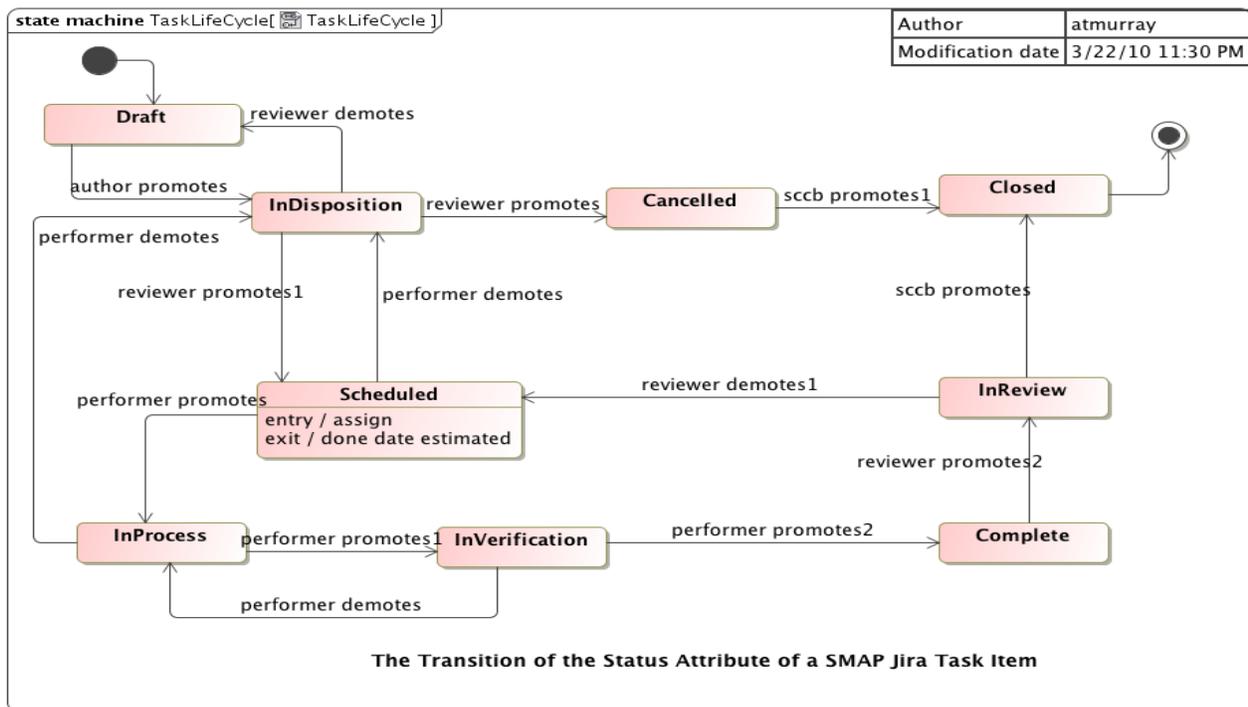


Figure 20 - The life cycle of an anomaly report or task issue

has an enumeration named ArchDocOrder. The modified Velocity template uses MagicDraw macros to recursively descend into the model. In each package it finds the enumeration and puts out the named elements (if they're not packages), and descends into any packages named in the list in the order they're encountered.

The modified velocity templates can be configured based on the settings of a group of variables that it reads. These include:

- (1) The root package at which to start generating
- (2) The document's name
- (3) The name of the enumeration to use to order the generation and output
- (4) The root package at which to start generating
- (5) The document's name
- (6) The name of the enumeration to use to order the generation and output

With this configurability, we can use the generator for a variety of documents, generated with overlapping sets of models. We will use this generator for our detailed verification plan document (which will include definitions of the level 4 verification scenarios) and for software design documents for all of the components.

There are actually two templates: one generates the navigation tree seen in the left frame of the web document, the other generates the content frame. Both have the same logic for navigating the model.

10. CONCLUSIONS AND ONGOING WORK

Every methodology has pros and cons. In this section we list and describe the principle advantages and disadvantages of our approach.

Problems and Disadvantages

We have encountered difficulties and disadvantages in our approach:

- (1) *Tool unreliability*: For the most part, we are quite happy with MagicDraw: it's a fine tool, it has a high-fidelity representation of the UML Metamodel that is available to the user, and in general it's fairly easy to use, considering the depth and power it puts at the user's disposal. But we have encountered difficulties with it: it is not the most reliable piece of software we've seen, and finding workarounds for some of its foibles has been time consuming.

- (2) *Modeling time overhead*: This may be related to the previous item, but we notice that time is consumed by details and idiosyncracies of the model itself, perhaps because the model and modeling tool are such rich and engrossing things. This is somewhat similar to the tendency to endlessly tweak the formatting of a Word document. We have to continually remind ourselves that the model is a means to an end.
- (3) *Communication outside of our team*: Audiences not familiar with modeling techniques and UML and SysML in particular can be put off by presentations featuring these diagrams.
- (4) *Requirements tool difficulties*: The interface for exchange between MagicDraw and DOORS is not as automatic and seamless as we would like. From our point of view, DOORS is the problem, though clearly it is a core tool for systems engineering at JPL and will not be replaced soon.

Advantages and Benefits

We have experienced and observed positive aspects and results of our model-based emphasis in FSW engineering as well:

- (1) *Architecture enabling*: The use of modeling seems to foster the early analysis of architectural issues and concepts. Making a model of something is kind of a forcing function that prods the modeler to obtain a better understanding of what is being modeled.
- (2) *Communication of designs*: The use of UML enables software engineers to visualize, understand, and communicate complex software design concepts quickly. We find the use of UML, frequently sketched on paper or white boards, facilitates and streamlines our architecting and design process.
- (3) *Reusability of design and architecture*: Storing our architectural and design information in a model allows us to re-use aspects of the information in multiple situations easily, e.g. using the same design elements in different kinds of diagrams for different purposes, e.g. for inclusion in a document or a presentation. If we used a simple drawing tool, each diagram would only serve the purpose it was originally intended for. This reusability also fosters consistency, a by-product of maintaining information in a single place.
- (4) *More useable traceability*: While there is utility in having a spreadsheet that maps, for example, requirement numbers to test case numbers, it is much more useful to have a trace relationship that can be easily followed from one element to another, e.g. seeing a requirement in context, following a *Verifies* relationship easily to a test, and being able to see the logic of the test

Ongoing and Future Work

There are some specific activities and techniques that we would like to delve into more. Given the demands of a project and the need to prioritize, we may not be able to accomplish these things.

- (1) *Design checking using automated model analysis and validation*: We described a few of our design constraints. If we could be certain that these constraints were followed in our design, our FSW product would likely be more reliable. The checking of these constraints could be automated, using MagicDraw's built in validation language (or using Java – it is possible to augment MagicDraw validation with Java programs).
- (2) We would like to arrive at a fuller evaluation of how well these methods are working. We plan to collect metrics for use in comparison with other projects, e.g. software defect rates, in order to try to get an objective assessment of these techniques for these kinds of projects.

Concluding Observations

This paper has described the application of a more consistently model-based technique than we've done before for flight software engineering at JPL.

It's still quite early in the process: we are coming up on our Preliminary Design Review, which will be the first external review of any kind of design detail. The ultimate effectiveness of this approach has to be judged, at least in part, on the success of the product: the timeliness of development, the quality of the software, the ease or difficulty of integration; these are the things that really matter. We think this approach will help us achieve these things.

We have had a FSW Architecture Review with a board of about a dozen reviewers, several external to JPL. Some experienced in flight systems, others not. We received a large number and variety of comments on the architecture and design. There were things about the architecture that they really liked, and some things they really disliked. There were many conflicting opinions among the board members.

But there was a consistent feeling among that board, highlighted in their report, that the review provided them with a more comprehensive and deeper insight into the software architecture than they had seen in flight software projects before. This seemed to us an indication of the value of this model-based approach from one point of view: the ability to communicate architecture, requirements, and software design effectively.

REFERENCES

- [1] Object Modeling Group (OMG), "OMG Unified Modeling Language (OMG UML), Superstructure", version 2.3, May 2010
- [2] OMG, "OMG Systems Modeling Language (OMG SysML)", version 1.2, June 2010.
- [3] IEEE Software Engineering Standards Committee, "IEEE Std 1471–2000, Recommended Practice for Architectural Description of Software-intensive Systems", September 2000.
- [4] Jet Propulsion Laboratory's public SMAP mission website: <http://smap.jpl.nasa.gov/>
- [5] Martin Fowler, "UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)"
- [6] Sanford Friedenthal, Alan Moore, Rick Steiner: "A Practical Guide to SysML: The Systems Modeling Language (The MK/OMG Press)"
- [7] Benowitz, E.; Clark, K.; Watney, G., "Auto-coding UML statecharts for flight software", Space Mission Challenges for Information Technology, 2006. SMC-IT 2006. Second IEEE International Conference on (0-7695-2644-6)
- [8] Holzmann, Gerard; "The Spin Model Checker — Primer and Reference Manual", Addison-Wesley, 2003. ISBN 0-321-22862-6.
- [9] The IBM Rational DOORS requirements tool: <http://www.ibm.com/developerworks/rational/products/doors>
- [10] Object Modeling Group (OMG), "A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2", June 2008
- [11] Todd Bayer, Matthew Bennett, Chris Delp, Daniel Dvorak, J. Steven Jenkins, Sanda Mandutianu: "Update - Operations Concept for Integrated Model-Centric Engineering at JPL", IEEE Aerospace Conference Proceedings, March 5, 2011

ACKNOWLEDGEMENTS

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

We wish to thank our colleagues Michael Sierchio and Jeff Levison for a careful review of this paper.

BIOGRAPHY



Alex Murray is a senior software engineer with the Jet Propulsion Laboratory, California Institute of Technology. He is currently technical lead for the development of the FSW for the SMAP project at JPL. He has led and done software development for flight, ground, and simulation software for missions and for technology development projects at JPL.

Previously he led and developed software for a variety of projects at TRW (now Northrop-Grumman), and he served as a system engineer for the European weather satellite agency, Eumetsat, as well as software engineer for the Dresdner Bank in Frankfurt, Germany. His experience includes embedded and flight software development, prototype and research development, OS development, AI, analysis and simulation tools, science and image processing applications, business and GUI applications, and databases. He holds BS and MS degrees in mathematics from The Ohio State University.



Dr. (Owen) Shang-Wen Cheng currently works as a Flight Software Engineer at Jet Propulsion Laboratory. He performed a year of post-doctoral research with Professor David Garlan at Carnegie Mellon University. Owen's research interests include self-adaptive

systems, software architectures, control systems, and software framework design, with an ultimate aim of simplifying and automating complex but routine human tasks. He received a Ph.D. in software engineering from Carnegie Mellon and a B.S. in computer science from Florida State University. On his off-time, he enjoys spending time with his wife on the finer things in life.



Leonard Reder is currently assigned to the Soil Moisture Active Passive mission Flight Software team at the Jet Propulsion Laboratory to promote the use of UML and model based engineering techniques for generating real-time software implementations. He was the lead developer of the automatic flight software interface code generation

tool set for the Mars Science Laboratory mission. In 2003 he deployed a science sequencer application, utilizing UML modeling techniques, for control of the Keck Interferometer, located a top Mauna Kea on the "Big Island" of Hawaii. Reder has extensive work experience in software development techniques and processes, software modeling, software design patterns, real-time image and DSP processing, autonomy, and media technologies. He holds MSEE from the University of Southern California and BSEE from Cal Poly University at San Luis Obispo



Chris Jones is a software engineer with the Jet Propulsion Laboratory, California Institute of Technology. He is currently a flight software developer for the SMAP mission at JPL and has provided integration and test support to a fractionated spacecraft project at JPL funded by DARPA. Chris has previously developed algorithms to analyze DNA

microarray data at the UCLA Semel Institute for Neuroscience and Human Behavior with the goal of elucidating the genetic basis of complex human disease. He has also contributed to a high-performance, power-aware computing research at the Scalable Performance Laboratory. This project aimed to reduce the overall energy consumption of large-scale compute clusters by dynamically throttling CPU voltage based on the execution characteristics of scientific applications. He has experience in the areas of embedded flight software development, bioinformatics algorithms, computational genetics, high performance computing, and microprocessor architecture. Chris holds a BS degree in computer science from the University of South Carolina Honors College and an MS degree in computer science from the University of California, Los Angeles.