A model-based Architecture for a small flexible Fault Protection System

Garth Watney

Jet Propulsion Laboratory,
California Institute of Technology
Flight Software and Data Systems Section

Copyright 2008 California Institute of Technology, Government Sponsorship Acknowledged
Agenda

- Introduction
  - Local Detection, Centralized Mitigation Fault Protection
  - A Small Flexible Software Architecture using Statechart models
- Fault Protection Architectures for Space Missions
  - Cassini AACS
  - Deep-Space One
  - Deep Impact
- A model-based Software Architecture
- Implementing Models into Flight Software
- Implementing State-chart models using the Quantum Framework
- Space-based Interferometry Mission (SIM) Fault Protection
- Light-weight Statechart Architecture
- Advantages of this Architecture
- Shortcomings of this Architecture
- Conclusion
Introduction

- Traditional Fault Protection implementation
  - Local Detection, Centralized Mitigation Fault Protection
  - This work proposes a small flexible software architecture that fits within this traditional paradigm (as opposed to an advanced goal-based system such as JPL’s MDS Architecture where fault protection is an integral part of the design and not a separate subsystem.

(LFP = Local Fault Protection)
Fault Protection Architectures for Space Missions

- **Cassini AACS (Attitude and Articulation Control Subsystem)**
  - Parallel Recovery Execution that in turn drive an immense V&V effort to identify contradicting or overlapping system interactions.
  - Rule-based system:
    - Symptom events $\rightarrow$ Diagnostic Rules $\rightarrow$ Activation Rules
  - Manual Ada Code
    - No software implementation models.
  - Advantage
    - Well-tested and extremely capable fault protection system capable of responding to all conceivable anomalies
  - Disadvantage
    - High implementation cost
    - “pile of code”
    - Implementation was tightly coupled to the Cassini mission – not malleable to change or reuse across missions
• Deep Space One (DS1)
  – 2 Layer Fault Protection architecture applied to single response execution.
    • Shorter (more urgent) responses could interrupt longer (less urgent) running responses
  – A Fault Protection Engine responsible for fault isolation and the orderly execution of responses was implemented manually in C
  – Statechart models were used to define all Monitors and Responses using the Matlab Stateflow tool.
    • Stateflow auto-generated C code from the models which were in-turn transformed into C flight code
  – Advantages
    • One system engineer could develop Monitor and Response models
  – Disadvantages
    • Statechart models did not conform to the UML statechart semantics
    • Stateflow tool saved models in a proprietary output
    • Stateflow tool generated non flight-like C code
    • Fault Protection Engine was tightly coupled to the DS1 mission – not malleable to change or reuse across missions
      – “pile of code”
Deep-Impact

- Legacy DS1 Design
  - 2 Layer Fault Protection architecture applied to single response execution.
- A refactored mission-agnostic Fault Protection Engine was coded in C++
- Statechart models were used to define all Monitors and Responses using the Matlab Stateflow tool.
  - Stateflow auto-generated C code from the models which were in-turn transformed into C++ flight code
- Advantages
  - One system engineer could develop Monitor and Response models
  - A “reusable” Fault Protection Engine software component
- Disadvantages
  - Statechart models did not conform to the UML statechart semantics
  - Stateflow tool saved models in a proprietary output
  - Stateflow tool generated non flight-like C code
  - Fault Protection Engine was still a fairly complex software component and not malleable to changes in the behavior.
    - “pile of code”
A model-based Software Architecture

- This Architecture provides an inheritance base from which we can build different software applications.

- Model-based Framework Layer provides:
  - A Framework for the development of an Application specified as a collection of models.
  - A level of abstraction higher than the real-time Operating System.
  - Common real-time software capabilities such as schedule control, commanding and telemetry.
Implementing Models into Flight Software

- **UML Modeling**
  - Explicitly capture the intent of the requirements
  - Formally capture the behavior in a model
  - Create a crisp notion of *state*

- **State-based Framework**
  - Supports the UML standard
  - Allows developers to think and work with higher constructs – states, events and transitions

- **Auto-coding**
  - Light-weight Java program
  - Reads in the Model which is stored in a non-proprietary data format (XML)
  - Converts the input model into an internal data structure
  - Has multiple back-ends to support different project requirements

- **Test harness**
  - Ability to run the model stand-alone – module test environment

- **Model checking**
  - Automatic generation of Verification models
  - Exhaustively explore the state-space of the model
  - Checks for various correctness properties within the model
Implementing State-chart models using the Quantum Framework

- Small lightweight framework intended for embedded real-time applications
- Supports active objects
  - Event-driven, concurrently executing objects
  - Each object embedding a Hierarchical State Machine
  - Objects do not share data – only communicate via an exchange of event instance
• Monitors, Responses and the Fault Protection Engine are all expressed as a collection of interacting UML Statechart models.
• Fault Protection strategy is encapsulated in the FP Engine model.
• Underlying software architecture is a model-based architecture which directly supports the instantiation, execution and communication of hierarchical state-machines.
• Advantages
  – All FP software components are explicitly modeled and flight code generated directly from these models
  – Our model-based software development process is not tied to a vendor’s auto-coding or drawing tools.
  – The auto-generated code is readable and understandable since the UML statechart syntax is directly mapped to tried and tested software design patterns.
  – The fault protection strategy was explicitly captured as a model enabling the specific strategy to be malleable to changes.
• Disadvantages
  – Abandoning the Matlab Stateflow resulted in system engineers not directly specifying the monitor/response model
  – Scalability problems – multiple instantiations of the same model and processing throughput.
Light-weight Statechart Architecture

- Based on the SIM model-based Architecture with the following modifications:
  - State-machines can be grouped into subsystems that communicate only at the local level
  - Monitors, which are multiple instantiations of the same state-machine are distributed amongst several subsystems and eliminates the problem of them all responding to the same global event.
  - FP Manager tracks and invokes only the active Responses which eliminates the throughput problem.

- Light-weight state-machines have the following restrictions:
  - Do not subscribe to events
  - Must be encapsulated by a global interface component
Advantages of the Architecture

- Modeling the FP Engine provides a blueprint template that can be customized for any future mission
- The FP Engine can also be extended into a general purpose Autonomy Engine that responds to non fault events
- Models in general provide a communication medium between system and software engineers
- Formal and explicit models reveal whether a software engineer has understood the intent of the imposed requirements
- Models help to nail down ambiguous or loosely worded requirements
- Executable models can be used as a prototype to demonstrate and test early behavior
- Models can be used to analyze and document the design
- Models can be used to automatically generate large portions of the flight software code
Shortcomings of the Architecture

- The Architecture explicitly supports capturing the dynamic behavior of a software component as a state-machine. This works very well at the local software component level but fails to capture the following crucial aspects:
  - Component interaction at the global level
  - Scheduling execution of components
  - Threads of execution

- Very few software defects arise from incorrect implementation at the local level, but defects are often found from unforeseen interactions at the global level.
  - Component publishes an event without a recipient
  - Component subscribes to an event without a publisher
  - Component executed in a high rate group publishing events to a component executed in a low rate group causes queue overflows

- Need to explicitly model the component communication and the execution threads.
  - Analyze the global interactions of software components and remove these type of defects at build-time or before
Conclusion

• What’s the goal:
  – Create a software architecture that is highly flexible in its particular FP philosophy.
  – Involve system engineers in the software development process
  – Explicitly and formally capture the FP Engine, Monitors and Responses as models.
    • Analyze the models
    • Auto-code the flight software
    • Malleable to change

• Identified short-comings
  – Global behavior is not being explicitly modeled which leads to defects in the communication between software components

• Future work
  – Explicitly model the high level architecture for component interaction and execution scheduling.