# Model-Based Software Quality Assurance with the Architecture Analysis and Design Language

Peter Feiler[*] and David P. Gluch[†]
*Software Engineering Institute, Pittsburgh, PA, 15213-2612*

Kathryn Anne Weiss[‡]
*Jet Propulsion Laboratory, Pasadena, CA, 91109*

*and*

Kurt Woodham[§]
*L-3 Communications, Fairmont, WV, 26554*

**Model-based software quality assurance (MB-SQA) provides a rigorous framework for the verification and validation of software systems through the systematic modeling and analysis of formal architecture representations. This paper describes the results of applying an MB-SQA practice framework that utilizes the Architecture Analysis and Design Language (AADL) to JPL's Mission Data System (MDS) reference architecture. The MDS is a unified reference architecture for space mission flight, ground, and test systems. In the case study, the AADL assurance practice framework and several AADL-based analyses were applied to the evaluation of critical quality attributes of the MDS reference architecture as well as an MDS adaptation for the control of a heated camera. The results of the case study demonstrate the utility of the practice framework and the AADL-based analyses in addressing (1) the modeling of key MDS architectural themes and (2) quality assurance with respect to performance, particularly flow latency.**

## Nomenclature

| | | |
|---|---|---|
| AADL | = | Architecture Analysis and Design Language |
| FOM | = | Figure of Merit |
| IV&V | = | Independent Verification and Validation |
| MB-SQA | = | Model-Based Software Quality Assurance |
| MDS | = | Mission Data System |
| OSATE | = | Open Source AADL Tool Environment |
| SQA | = | Software Quality Assurance |
| V&V | = | Verification and Validation |

## I. Introduction

Throughout software engineering literature, software quality assurance (SQA) typically refers to monitoring the software development process to ensure that the project is adhering to established development standards and procedures. In other words, the high quality (read "quality" as an adjective) of a software system is assured through the rigorous enforcement of standards and procedures.

However, SQA can also refer to ensuring that the software system has certain qualities (here, read "quality" as a noun), or, as referred to in the software architecture community, quality attributes. These quality attributes are

[*] Senior Member of the Technical Staff, 4500 Fifth Avenue.
[†] Visiting Scientist from CSE Department at the Embry-Riddle Aeronautical University, 600 S. Clyde Morris Blvd., Daytona Beach, FL 32114-3900.
[‡] Flight Software Engineer, Systems and Software Division, 4800 Oak Grove Drive M/S 301-225, AIAA Member.
[§] Chief Engineer, Enterprise IT Services Division, 100 University Drive.

indirectly measurable software system properties, such as maintainability, performance, and safety, desired by system stakeholders. In order to ensure that a software system meets the quality attribute requirements set forth by the system stakeholders, software engineers must perform software quality assurance.

This type of SQA is a critical, yet difficult task for embedded software systems, because (1) quality attributes need to be built into a system from the beginning and (2) quality attributes are indirectly measurable.

First, quality attributes must be engineered into the software system through the employment of specific tactics, such as architecture styles, design patterns, or reference architectures.[**] These tactics are selected to address a particular or several quality attribute(s) requirements and contribute to the overall architectural concept. Second, quality attributes are indirectly measurable. For example, there is no single measurement for "performance," because performance is highly coupled to the functional requirements of the software system, the characteristics of the system within which the software is embedded, and the non-functional needs of the system stakeholders. Therefore, Figures of Merit, or FOMs, are usually defined, such as data transport latency, task execution time, and latency jitter, that quantify the quality attribute, in this case performance, with respect to this system context. These FOMs provide software engineers with an indication of performance and system stakeholders with assurance that their concerns are being addressed.

The architectural concept and quality attribute FOMs are documented as part of the software architecture and specified in an "architecture description document." Traditionally, the views that comprise an architecture description do not contain formal models that can be analyzed with respect to these FOMs. The lack of rigorously specified software architecture description documents makes it extremely difficult for quality assurance personnel to determine whether or not an architecture has been constructed such that stakeholder quality concerns are addressed.

These problems can be addressed through taking a model-based approach to SQA. Model-based software quality assurance (MB-SQA) is the application of model-based engineering techniques (i.e., the use of formal abstractions and analyzable representations to perform typical engineering tasks) to the verification and validation of software architectures with respect to quality attributes. Model-based engineering techniques utilize structured, graphical, implementation-independent notational systems for producing unambiguous documentation of system requirements and design, thereby providing the foundation for formal and effective software quality assurance. Using MB-SQA for both verification and validation (V&V) and independent verification and validation (IV&V), which are often employed at each phase of the embedded software system development lifecycle, provides software engineers with the capability to develop a thorough understanding of and insight into the critical characteristics of a system that are vital to its correct operation.

This paper proposes an approach to MB-SQA that leverages the Society of Automotive Engineers (SAE) Architecture Analysis and Design Language (AADL) and Open Source AADL Tool Environment (OSATE). It is proposed that these tools and associated practice framework can be employed in both the V&V and IV&V efforts of embedded software systems development to help ensure the achievement of stakeholder quality requirements. This formal modeling language and supporting toolset were applied to representing and analyzing an adaptation of the Jet Propulsion Laboratory (JPL) Mission Data System (MDS) reference architecture for real-time embedded control systems, to evaluate the effectiveness and ease of applying this approach to a real system.

Section 2 provides background on the AADL, the AADL assurance practice framework, and examples of the types of quality attribute analyses that can be performed by using AADL and the AADL assurance practice framework. Section 3 presents an overview of the MDS reference architecture and observations on how MDS architectural themes relate to both AADL modeling techniques and model-based engineering of software systems in general. Section 4 presents the case study used to illustrate the AADL assurance practice framework as well as the models that were developed for that case study. The results of performing various analyses on these models are also presented. Finally, Section 5 presents conclusions and future work.

[**] An architecture style is a form or pattern of design with a shared vocabulary of design idioms and rules for using them.[1] A design pattern is a commonly recurring structure of communicating components that solves a general design problem within a particular context. A reference architecture an architectural pattern, or set of patterns, partially or completely instantiated, designed and proven for use in particular business and technical contexts, together with supporting artifacts to enable their use.[2]

## II. BACKGROUND

This section provides necessary background information for understanding how the AADL is useful in performing MB-SQA and to understand the context for its application to the MDS reference architecture. First, a brief overview of the AADL is presented. Then, a summary of the AADL assurance practice framework is outlined. Finally, MDS and its architecture concepts are introduced.

### A. AADL

The SAE AADL standard provides formal modeling concepts for the description and analysis of software system architectures in terms of distinct components and their interactions.[3] The AADL includes software, hardware, and system component abstractions to (1) specify and analyze real-time embedded systems, complex systems-of-systems, and specialized performance capability systems and (2) map software onto computational hardware elements. The AADL is especially effective for model-based analysis and specification of complex real-time embedded systems.[4]

Within the AADL, a component is characterized by its identity (a unique name and runtime essence), possible interfaces with other components, distinguishing properties (critical characteristics of a component within its architectural context), and subcomponents and their interactions. In addition to interfaces and internal structural elements, other abstractions can be defined for component and system architectures. For example, abstract flows can be identified and associated with specific components and interconnections to perform flow analyses. These additional elements can be included through core AADL language capabilities and the specification of a supplemental annex language.

The component abstractions of the AADL are separated into the three categories listed below.[††] Figure 1 shows the AADL graphical representations of these component abstractions.

**Application Software**:

- Thread: active component that can execute concurrently and be organized into thread groups
- Thread Group: component abstraction for logically organizing thread, data, and thread group components within a process
- Process: protected address space whose boundaries are enforced at runtime
- Data: data types and static data in source text
- Subprogram: concepts such as call-return and calls-on methods (modeled using a subprogram component that represents a callable piece of source code)

**Execution Platform (Hardware)**:

- Processor: schedules and executes threads
- Memory: stores code and data
- Device: represents sensors, actuators, or other components that interface with or are part of the physical environment
- Bus: interconnects processors, memory, and devices

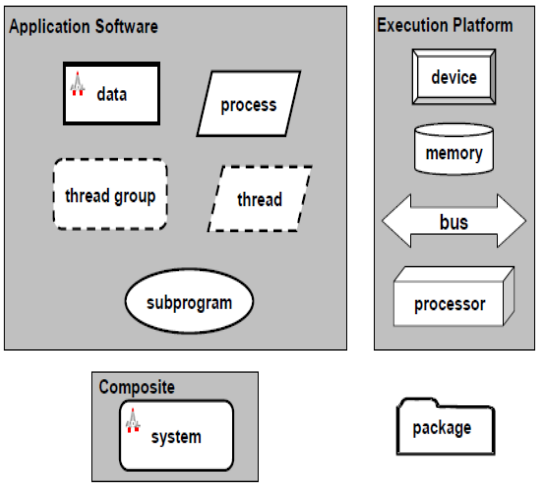



**Figure 1. AADL Component Abstractions**

**Composite**:

- System: design elements that enable the integration of other components into distinct units within the architecture; they can consist of other systems as well as of software or hardware components

AADL also includes several component interactions, which can be separated into the four categories listed below. Figure 2 shows the AADL graphical representations of these component interactions.

**Port Connections:**

- Data Port: interfaces for typed state data transmission among components without queuing
- Event Port: interfaces for the communication of events raised by subprograms, threads, processors, or devices that may be queued

[††] Throughout the remainder of this paper, AADL component abstractions are indicated by capitalization. AADL component instances in models are indicated by italics.

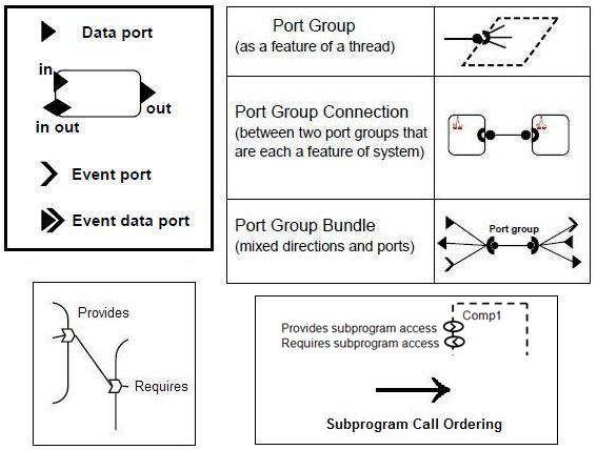



**Figure 2. AADL Component Interactions**

- Event Data Port: interfaces for message transmission with queuing
- Port Groups: a connection for a collection of ports or other port groups

**Component Access Connections:**

- Provides: indicates that a component provides access to a data or bus component contained within it
- Requires: indicates that a component requires access to a shared data component or bus component that is external to it

**Subprogram Call Connections:**

- indicates the control flow of a call from the caller to the callee – both local and remote

**Parameter Connections:**

- indicates the flow of data from ports to subprogram parameters and between parameters of different subprogram calls

The AADL standard includes runtime semantics for data exchange and control mechanisms including message passing, event passing, synchronized access to shared components, thread scheduling protocols, timing requirements, and remote procedure calls. In addition, dynamic reconfiguration of run-time architectures can be specified using operational modes and mode transitions within the context of both software and hardware components. For a full description of the AADL constructs please refer to Ref. 5.

### B. AADL Assurance Practice Framework

The AADL assurance practice framework provides a foundation of processes, artifacts, methods, and tools used to perform MB-SQA during both V&V and IV&V activities. Figure 3 presents an overview of the AADL assurance practice framework. For a more detailed description of the AADL assurance practice framework please refer to Ref. 6.

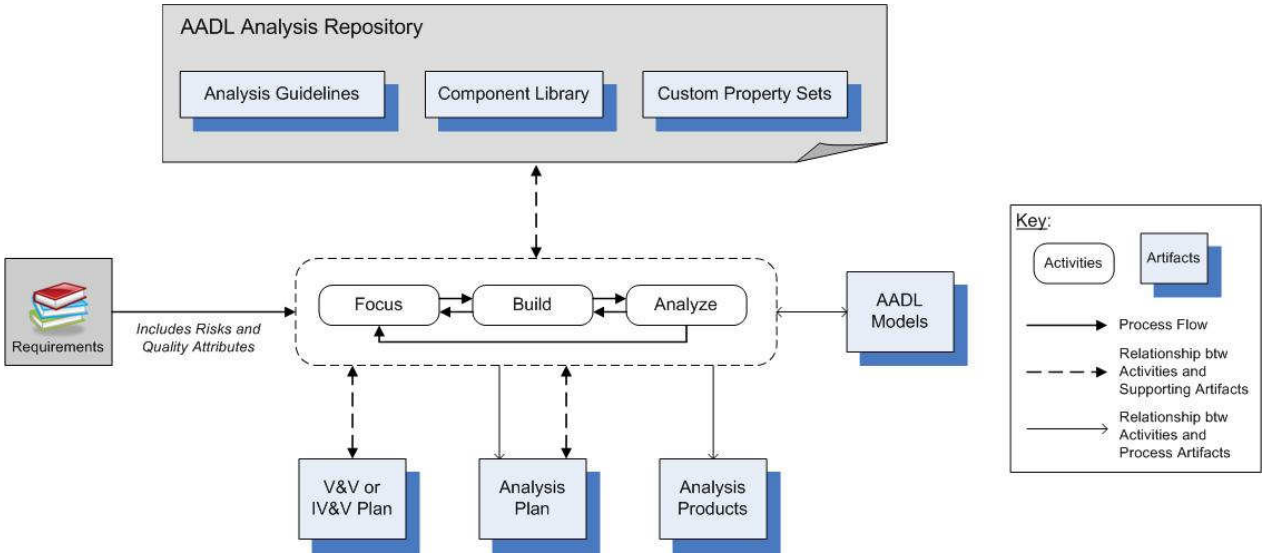



**Figure 3. AADL Assurance Practice Framework**

As illustrated in Figure 3, the AADL assurance practice framework has three main activities: Focus, Build, and Analyze. These activities form an iterative process with continuous feedback flowing from one activity to the next.

The Focus activity determines the components of the embedded software system that will be modeled and analyzed for quality assurance. In many situations, it may not be desirable or feasible to develop full, detailed

models of the entire software system architecture. Therefore, the Focus activity is driven by the critical issues identified by the system stakeholders (i.e., high risk quality requirements) and the analysis methods used to probe these issues. The outputs of the Focus activity are a well-defined Analysis Plan and any necessary modifications to the V&V or IV&V Plan.

The principal objective of the Build activity is to develop AADL models of the software system elements identified during the Focus activity. A model is a full or partial representation of a software system element sufficiently detailed to support one or more focused analyses. The initial steps in the Build activity involve referencing appropriate Analysis Guidelines (part of the AADL Analysis Repository) that define specific methods and identify tools for developing and analyzing models. The output of the Build activity is the set of AADL models.

The Analyze activity involves conducting detailed assessments of targeted aspects of the embedded software system using the models created during the Build activity.[7] Different assessments often require different models. For example, a reliability assessment requires the use of a stochastic process model, while a scheduling assessment requires the use of a timing model. Analysis Guidelines establish the requisite methods and steps for various types of analysis. A unified architecture analysis model for each software system is maintained as part of the Analyze activity to ensure that the various analysis models accurately represent the same architecture. Utilizing the extensibility of the AADL notation, this unified architecture analysis model is annotated with information relevant for different types of analysis, so that analysis models can be easily derived from one unified model of the software system, thereby reducing the analysis model validation step to the examination of the filters that generate the analysis models.

The Analyze activity leverages OSATE, the Open Source AADL Tool Environment.[8] OSATE is an extensible tool environment based on Eclipse that provides textual and graphical AADL editing support, semantic checking of AADL models, and translation of those models into XMI. Several analysis tools are available that interface with OSATE. Some of these, such as security and end-to-end latency analysis tools, operate directly on an instance of the AADL model. Others interface with AADL models through filters that map relevant information from the AADL model into representations specific to the analysis tool, supporting activities such as network loading or fault tree analyses.

Finally, it should be noted that the AADL assurance practice framework advocates the use of an AADL Analysis Repository, which consists of reference materials that support modeling and analysis for V&V and IV&V. As shown in Figure 3, the repository consists of Analysis Guidelines, a Component Library, and Custom Property Sets. Analysis Guidelines provide V&V and IV&V personnel with supporting materials for (1) formulating analysis strategies, (2) establishing key parameters that should be considered, and (3) identifying specific analysis processes, methods, and tools for their particular application. These guidelines are organized into viewpoints that address broad concerns associated with quality and other non-functional attributes of the target system as well as critical behavioral aspects. The Component Library is a collection of AADL component type and implementation declarations that can be used to create analysis models for a target system. These components are organized into hierarchical layers ranging from general components that can be used across an organization to specialized component variations required for a specific project. The repository's Custom Property Sets include specialized properties required for analyses. These properties are integrated into analysis models through a built-in AADL capability that allows a user to define new properties and property types. Initially, the content of the repository is general; however, during its use in a project or organizational context, the content of a repository evolves. Components and properties are added and / or modified, and new analysis techniques and tools are identified.

### C. The Mission Data System

The JPL MDS project was initiated in April 1998. The principal project objectives were "to define and develop an advanced multi-mission architecture for an end-to-end information system for deep-space missions" and "to address several institutional objectives: earlier collaboration of mission, system and software design; simpler, lower cost design, test, and operation; customer-controlled complexity; and evolvability to in situ exploration and other autonomous applications."[9] Figure 4 provides a conceptual diagram of the MDS reference architecture.

MDS is a goal-based system, which is defined as a system in which all actions are directed by goals instead of command sequences. A goal is a constraint on a state variable over a time interval.[10] Types of states include: dynamics, environment, device status, parameters, resources, data product collections, data management and transport policies, and externally controlled factors.

In MDS, the hardware adapter receives information about the environment and the hardware itself from the sensors. These measurements are used as evidence and are passed to a state estimator. The state estimators use the evidence provided by the hardware adapter and the history of the states to estimate the current state of the system including an estimate of the uncertainty. Operators express their intent in the form of goals declaring what should

happen, as opposed to low-level command sequences that dictate how the intent is to be achieved. A Mission Planning and Execution function then elaborates and schedules these goals based on the current state of the system as determined by the state estimator. The state estimates and elaborated goals are inputs to the state controllers, which issue appropriate commands to the hardware to achieve these goals. More detail on Goal Elaboration, Mission Planning, and Execution is provided in Ref. 10.

The following subsection provides an overview of the key MDS architectural themes that motivate the state- and goal-based control approach described above. The next section then provides a mapping between the MDS architectural themes and constructs and the approaches in AADL that will be used to model MDS.

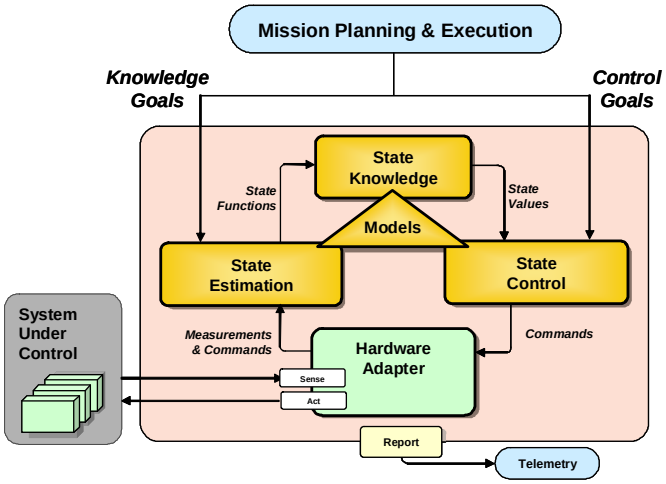



**Figure 4. The MDS Reference Architecture**

### D. MDS Architectural Concepts

The MDS architecture is based on a set of concepts that were developed to meet the needs of real-time embedded control systems given the unique characteristics of aerospace applications. The MDS architectural concepts include:

- *Take an Architectural Approach*: Construct subsystems from architectural elements, not the other way around.
- *Ground-to-Flight Migration*: Migrate capability from ground to flight, where appropriate, to simplify operations.
- *State and Models are Central*: System state and models form the foundation for information processing.
- *Explicit Use of Models*: Express domain knowledge explicitly in models rather than implicitly in program logic.
- *Goal-Directed Operations*: Operate missions via specifications of desired state rather than sequences of actions.
- *Closed-Loop Control*: Design for real-time reaction to changes in state rather than for open-loop commands or Earth-in-the-loop control.
- *Resource Management*: Resource state usage is projected with models and checked against constraints.
- *Separate State Estimation from State Control*: For consistency, simplicity and clarity, separate state estimation logic from control logic.
- *Integral Fault Protection*: Fault protection must be an integral part of the design, not an add-on.
- *Acknowledge State Uncertainty*: State estimation must be honest about the evidence; state estimates are not facts. State values are rarely known with certainty.
- *Separate Data Management from Data Transport*: Data management duties and structures should be separated from those of data transport.
- *Join Navigation with Attitude Control*: Navigation and attitude control must build from a common mathematical base.
- *Instrument the Software*: Instrument the software to gain visibility into its operation, not just during testing but also during operation.
- *Upward Compatibility*: Design interfaces to accommodate foreseeable advances in technology.

For a deeper description and exploration of these architectural themes, please refer to Ref. 9.

## III. Modeling MDS with AADL

In order to effectively model the Mission Data System with AADL, it was necessary first to create a mapping between the MDS architectural themes and constructs, and the approaches used in AADL. This mapping provides guidance in developing analysis strategies and approaches, identifying critical issues, and defining specific views and models for the MDS case study.

*Take an Architectural Approach*: AADL is an architecture description language for real-time embedded systems, and therefore enables modeling of component-based systems with fully defined interfaces and interactions between components.

*Ground-to-Flight Migration*: AADL supports modeling the migration of capability from ground to flight by clearly separating the software architecture at the application layer from its deployment on a physical and compute platform.

*State and Models are Central*: In MDS, state variable values (estimates) have a single producer and multiple consumers, and therefore state information flows from a single producer to one or more consumers. Consequently, state variables could be modeled in AADL as Data components that are accessed by producer and consumer tasks. In this representation, (1) information flow is reflected in the read and write access properties of the Data components, and (2) information transfer timing between a producer and consumer is implicit in the execution order of the producer and consumer tasks.

However, in this work, another modeling strategy was employed. AADL was specifically designed to model sampled state as signal streams in a closed-loop control system. Therefore, AADL provides Data Ports for representing the state estimates to be communicated. Furthermore, AADL provides Port connections that are used to express the flow of state information. Through the use of AADL Data Ports and Port connections, mid-frame (immediate) and phase-delayed communications can be expressed to ensure deterministic sampling. Deterministic sampling is important in order to maintain the stability of the control loops.

MDS state variables can thus be represented as the outgoing Data Port of the producer task that is generating the state estimate. Consumers (state estimators and state controllers) access state variables through Port connections to their incoming Data Port. Desired communication timing is specified through the connection semantics. By creating flow specifications of the individual components and the end-to-end system, V&V and IV&V personnel can annotate the flows with flow-related properties such as latency, data age, data accuracy and precision, and data miss rates for analysis purposes.

*Explicit Use of Models*: AADL is a formal language that supports rigorous modeling of systems as software and hardware components as well as their interactions.

*Goal-Directed Operations*: Goals in MDS are constraints on state over a time period. AADL supports the modeling of goals through the use of sublanguage annexes that enable the creation of domain-specific annotations to the AADL model.

*Closed-Loop Control*: AADL supports the modeling of closed-loop, flow-oriented architectures through the use of Data Ports that represent state and connections that represent flow. Deterministic flow is ensured through the use of mid-frame and phase-delayed connections. In other words, measurements are available through "out" Data Ports of sensor hardware adapters and are passed to estimators via Data Port connections. Data available to the estimator mid-frame is expressed by an immediate Data Port connection.

*Resource Management*: The execution platform components in AADL represent compute platform and physical resources. User-defined properties enable V&V and IV&V personnel to characterize resource capacities and resource budgets. The AADL concepts Processor, Memory, Bus, and Device define these resources as abstractions that include budget-based resource management.

System power consumption can also be modeled in AADL. With respect to MDS, power consumption can be addressed in two categories: power consumption of (1) the physical plant in the system under control and (2) the compute platform in the control system. The MDS Mission Planning and Execution function already focuses on power concerns of the physical plant. If desired, AADL can also capture power requirements of the physical plant elements through power-related properties on physical components modeled through the AADL Device concept. AADL can then be used to characterize variations in power requirements through Modes and Mode-specific Properties and different deployment configurations for performing trades.

The compute platform is represented in AADL as a resource on which the MDS application software is executed. Specifically, the binding of the hardware adapter software to the underlying compute platform is explicitly modeled in AADL to determine processor utilization.

*Separate State Estimation from State Control*: The AADL Package concept allows users to organize and compartmentalize the modeling space for representing multi-layer, componentized architectures. Consequently, state estimation can be modeled and packaged separately from the controllers that influence state.

*Integral Fault Protection*: AADL includes fault handling mechanisms as part of its execution semantics including Recovery Entry Points for threads, Error Event Ports for communicating with a health monitor, and Modes to represent various fault tolerant configurations. AADL also has an Error Model Annex extension that permits users to abstractly characterize fault behavior and fault propagation in support of fault impact and isolation analysis as well as reliability and fault tree analysis.[11]

*Acknowledge State Uncertainty*: AADL properties can be used to characterize the data represented in state variables including uncertainty characteristics.

*Separate Data Management from Data Transport*: AADL separates the logical flow of information through Data Ports from its deployment across both ground and flight compute platforms. Furthermore, the management of data history can be abstracted into the specification of desired data goals with respect to the state variable history logging and transport.

*Join Navigation with Attitude Control*: Joining navigation with attitude control is captured in AADL models through the use of a common set of state data accessed by both navigation and attitude control components.

*Instrument the Software*: Instrumentation of software can be modeled in AADL through Properties associated with software model elements. Alternatively, users can define AADL instrumentation patterns associated with model elements that are elaborated during model instantiation.

*Upward Compatibility*: AADL semantics allow the partial description of component interfaces so that they can be specialized within implementations or extensions. Furthermore, Properties on these interfaces can be used to explicitly capture upward compatibility requirements.

## IV. A Case Study

This section presents the results of using the AADL assurance practice framework to perform MB-SQA on the Mission Data System. First, the MDS reference architecture is modeled. Then, an adaptation of the MDS reference architecture, namely the control of a heated camera, is modeled. Finally, the MDS adaptation is analyzed with respect to flow latency, which provides an example of quality assurance with respect to performance in the V&V and IV&V context. The graphical AADL representations shown throughout this section were developed in OSATE and are equivalent to the textual AADL representations.

### A. MDS Reference Architecture Model

Figure 5 contains an AADL graphical description generated in OSATE of the MDS reference architecture. This diagram contains the same information as Figure 4 albeit a formal instead of informal representation and at a higher level of abstraction.

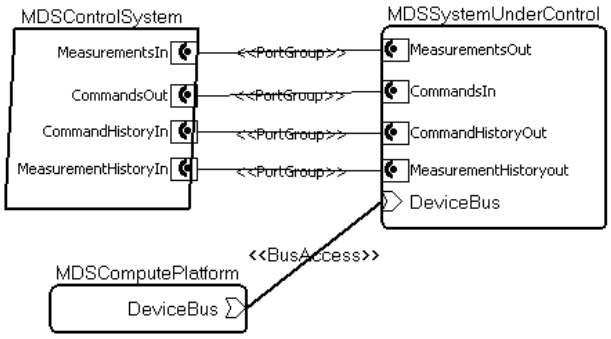



**Figure 5. MDS Reference Architecture AADL Model**

As seen in Figure 5, the AADL model is comprised of three top-level components, namely the *MDSControlSystem*, the *MDSSystemUnderControl*, and the *MDSComputePlatform*. The *MDSControlSystem* and the *MDSSystemUnderControl* interact with one another by passing sensor measurements and actuator commands as well as measurement and command histories. These interactions are depicted as connections between Port Groups to indicate that there may be a collection of connections between the two components. The *MDSComputePlatform* interacts with the *MDSSystemUnderControl* through the *DeviceBus* that provides physical access to the sensors and actuators in the system under control. In addition, the software components of *MDSControlSystem* are bound to hardware components of *MDSComputePlatform* via a binding Property (not shown in Figure 5).

*1. Modeling the Compute Platform*

Figure 6 contains a detailed AADL model of the *MDSComputePlatform*. The *MDSComputePlatform* may include the flight system and / or the ground system as well as the connectivity between the two. The *MDSComputePlatform* is connected to the *MDSSystemUnderControl* through the *DeviceBus* that provides physical access to the sensors and actuators in the system under control. The MDS hardware adapters are mapped to the compute platform in a deployment configuration through the use of AADL binding properties.

*2. Modeling the System Under Control*

The *MDSSystemUnderControl* consists of the devices and hardware adapters that comprise the system under control, which are defined when the MDS reference architecture is adapted for a specific system. Depending on the system being modeled, a single Device may represent the complete system under control. In that case, "out" Data Ports represent sensors, whose data content are measurements, and "in" Data Ports represent actuators, whose data

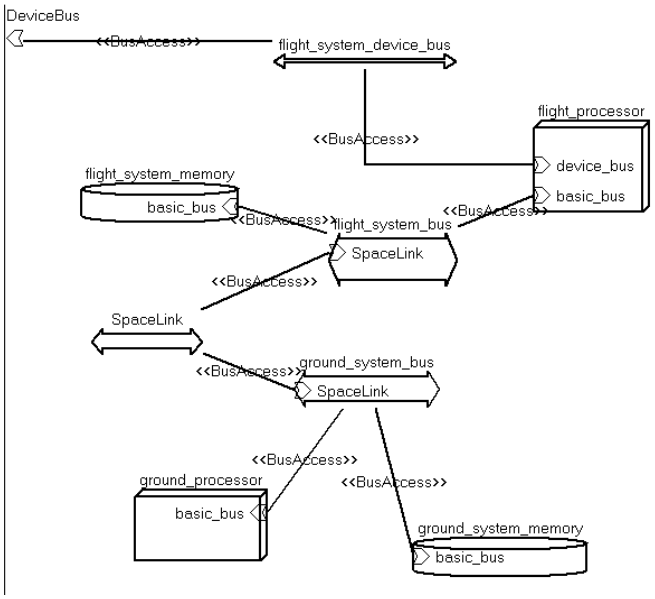



**Figure 6. MDS Compute Platform**

content are commands. Each sensor or actuator can be modeled as a separate Device, including one or more Data Ports for measurements and / or commands.

The AADL model of the MDS reference architecture consists of the hardware adapters, namely the *SensorHardwareAdapters*, which are responsible for converting sensor readings into measurements, and *ActuatorHardwareAdapters*, which are responsible for converting control commands into actuator commands. Sensor readings are passed from the physical system to the hardware adapters in the *MDSControlSystem* and control commands are converted by the hardware adapters and passed to actuators in the physical system.

Hardware adapters also maintain measurement and command histories and make them available to the *StateEstimation* component of the *MDSControlSystem.* For a specific MDS adaptation, the generic Port Groups shown in these figures are refined to represent specific sensor measurements and actuator commands, state estimates, xgoals, and histories.

*3. Modeling the Control System*

Figure 7 presents the AADL model of the MDS control system. As depicted in the conceptual view of the MDS reference architecture shown in Figure 4, the representation in Figure 7 depicts the major components of the *MDSControlSystem*, namely state estimation (*StateEstimation*), state control (*StateControl*), two components relating to execution (*GoalMonitor* and *GoalExecutive*), and two components related to planning (*GoalPlanner* and *OperatorConsole*).

As shown in Figure 7, state estimators are represented by the *StateEstimation* Thread Group and controllers are represented by the *StateControl* Thread Group. Bundling estimators and controllers as Thread Groups allows the refinement of each with a set of Threads that represent individual estimators and controllers.

The Port Group *StateEstimatesOut* represents the results of estimation, i.e., the observed state of the system under control. This Port Group is refined using Data Ports, each Data Port representing the current value of an estimated state variable. Estimated state variables are used by the Thread Group *StateControl*. Individual state estimators within the *StateEstimation* Thread Group may make use of each others' state values. The *StateEstimation* Thread Group is also responsible for maintaining a history log of the estimated states, which is made available through a separate Port Group *EstimateHistoryOut.*

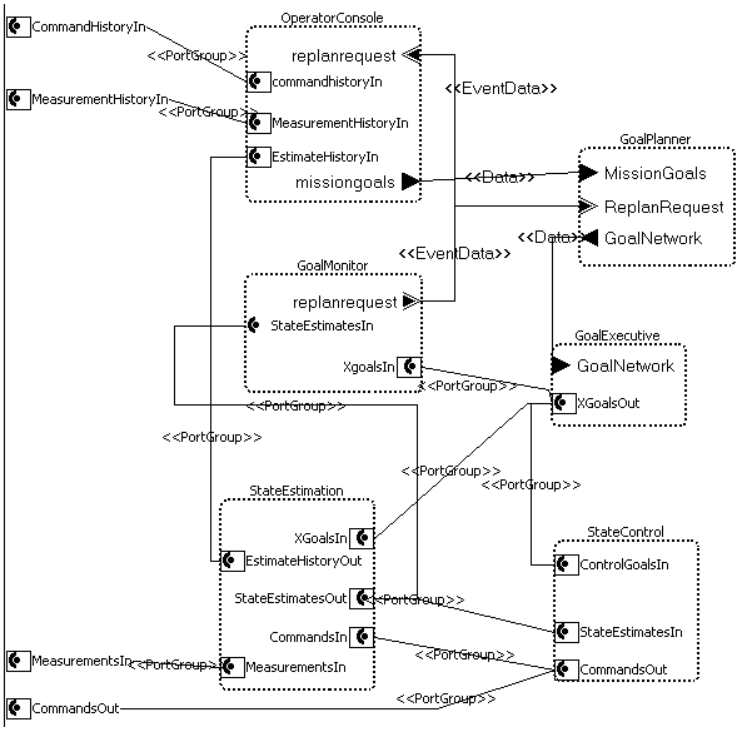



**Figure 7. MDS Control System**

Above the interface with the system under control is the execution layer of the MDS reference architecture, which consists of the *GoalMonitor* and the *GoalExecutive*. The *GoalExecutive* interprets a goal network, i.e., a mission plan, and passes executable goals to the controllers of the Thread Group *StateControl*. The *GoalMonitor* compares the state estimation history to the executable goals to determine whether the controllers are able to achieve the goals or if replanning should be initiated. The executable goals are represented by a Port Group that is refined in adaptations of the MDS reference architecture.

The *GoalPlanner* and the *OperatorConsole* address the planning aspects of the MDS reference architecture. The *GoalPlanner* is responsible for producing a goal network and replanning that goal network if the controllers are unable to achieve the goals within the goal network constraints. The *OperatorConsole* provides status including telemetry such as measurement, state estimate, and command histories, and allows for mission planning inputs by human operators.

*4. Model Organization*

The AADL Package concept is used to organize the modeling space as illustrated in Figure 8. All packages that comprise the MDS reference architecture model are included in one project (*MDS_Reference_Architecture*) in OSATE. The Package *MDSData* contains all declarations of Port Group types and Data component types. The Data component types are used in Data Port declarations to specify the type of data communicated through these Ports.

The Package *SystemUnderControl* contains the System declaration for the system under control. The *HardwareAdapters* Package contains the Systems representing the sensor adapters and the actuator adapters. The *MDSControlSystem* Package contains the MDS control system, while the components of the MDS control system, i.e., the state estimators, state controllers, goal executive, goal monitor, and goal planner, are declared in the *ControlSoftware* Package. The elements of the compute platform are declared in the *ExecutionHardware* Package. Finally, the top-level system is declared in the *CompleteMDSSystem* Package.

In addition to the packages, an MDS reference architecture Property Set is also defined for modeling rate groups. Other Property Sets can be similarly defined to analyze the MDS reference architecture with respect to other properties critical to MDS performance.

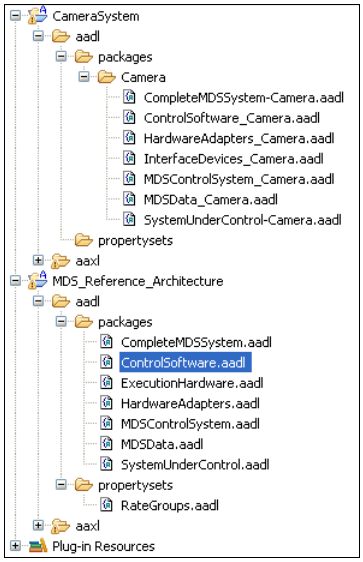



**Figure 8. Model Organization**

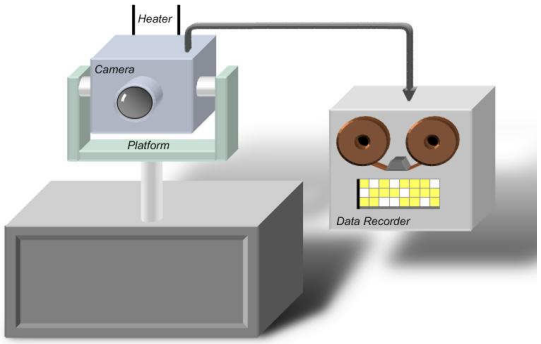
## B. MDS Adaptation Example

In this section, AADL is used to refine the MDS reference architecture model described in the previous section into an MDS adaptation model. AADL support for model refinement is used to accomplish this task. The MDS reference architecture is adapted to represent a specific MDS system in a separate OSATE project as seen in Figure 8, thereby enabling the independent development of multiple MDS adaptations. The AADL support for nested package names enables refinement of the original MDS reference architecture packages into system specific packages.

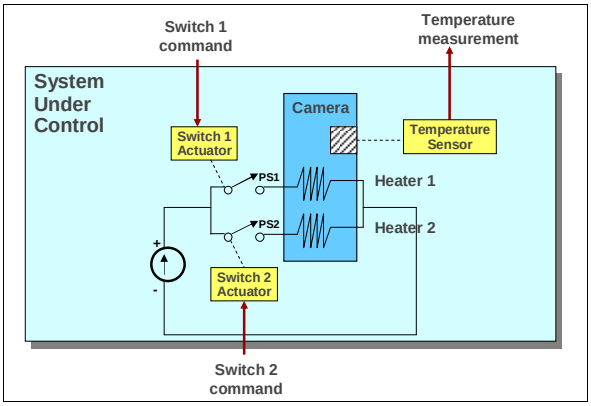
Individual components of the MDS reference architecture were refined by making use of the Extends construct. The Extends construct allows the declaration of a Port Group type, Component type, or Component implementation in terms of an existing type or implementation[‡‡]. These declarations refine previously declared features and subcomponents. The declarations can also be used to add subcomponents or features to the model. Port Group type extensions were declared to fill in the details of the Port Groups defined in the MDS reference architecture. Component type extensions were also declared to refine feature classifiers to the adaptation-specific Port Group and Component classifiers. Finally, component implementation extensions were declared that introduce specific instances of hardware adapters, estimators, controllers, goal executives, and goal monitors through subcomponent declarations.

In this case study, the MDS reference architecture is adapted to the temperature control of a camera mounted on a fixed platform, which is a typical control problem on board a spacecraft.[12] A diagram of the major components of the heated camera system is shown in Figure 9a. In this example, a temperature signal that originates in the temperature sensor (modeled as an AADL Device) flows through the control system, which controls the actuation of

[‡‡] For more information on the distinction between types, implementations, and instances please refer to Ref. 5.




**(a)**




**(b)**

**Figure 9. Fault-Tolerant Heated Camera Control System[12]**

the camera heater's power switch (also modeled as an AADL Device). A conceptual block diagram of the heater controller with sensors and actuators is shown in Figure 9b.

The camera hardware is modeled in AADL by refining the *MDSSystemUnderControl* component defined in the MDS reference architecture. Temperature sensor and heater switches (seen in Figure 9b) are represented as separate Devices. These Devices are physically connected to the Device Bus. The devices are connected to the hardware adapters, one for each sensor and actuator. These adapters provide a logical connection to the *MDSControlSystem* through the refined *MeasurementsOut* and *CommandsIn* Port Groups. These Port Groups have been refined to define the individual Data Ports used for communicating measurements and commands.

```
package ControlSoftware
public
  -- this type is refined for a MDS instance by refining the
  -- classifiers of the features to be instance specific
  thread group controller
    features
      StateEstimatesIn: port group MDSData::StateEstimatesIn;
      ControlGoalsIn: port group MDSData::XgoalsIn;
      CommandsOut: port group MDSData::CommandsOut;
  end controller;

  thread group implementation controller.basic
  end controller.basic;

  -- see comments regarding the controller
  thread group estimator
    features
      StateEstimatesOut: port group MDSData::StateEstimatesOut;
      MeasurementsIn: port group MDSData::MeasurementsIn;
      CommandsIn: port group MDSData::CommandsIn {
        StateValueHistory::ValueHistoryDepth => 2;
        };
      EstimateHistoryOut: port group ValueHistories::EstimateHistory;
      XGoalsIn: port group MDSData::XgoalsIn;
  end estimator;
```

**Figure 10. Example MDS Reference Architecture Package**

The components of the *MDSControlSystem* are also refined for this example. The MDS reference architecture has Thread Groups that represent collections of hardware adapters, estimators, controllers, goal executives, and goal monitors. Figure 10 shows the *Estimator* and *Controller* Thread Groups of the MDS reference architecture with Port Group features defining their interface to other MDS control system components.

In the heated camera example, these Thread Groups are refined by creating individual Threads for each component within the group. For example, in Figure 11 the *Estimator* Thread Group in the MDS reference architecture is refined into *TemperatureEstimator*, *TemperatureSensorHealthEstimator*, and *HeaterSwitchEstimator* threads through defining an *estimator.camera* Thread Group implementation that contains instance declarations for the three Threads.

```
package ControlSoftware::Camera
public
  thread group estimator
    extends ControlSoftware::estimator
    features
      StateEstimatesOut: refined to port group MDSData::Camera::StateEstimatesOut;
      MeasurementsIn: refined to port group MDSData::Camera::MeasurementsIn;
      CommandsIn: refined to port group MDSData::Camera::CommandsIn;
    --      EstimateHistoryOut: port group ValueHistories::EstimateHistory;
    --      XGoalsIn: port group MDSData::XgoalsIn;
    flows
      StateFlow: flow path MeasurementsIn -> StateEstimatesOut;
  end estimator;

  thread group implementation estimator.camera
    subcomponents
      TemperatureEstimator: thread TemperatureEstimator;
      TemperatureSensorHealthEstimator: thread TemperatureSensorHealthEstimator;
      HeaterSwitchEstimator: thread HeaterSwitchEstimator;
    connections
```

**Figure 11. Example MDS Adaptation Package**

For some system components in the MDS reference architecture, the refinement into the heated camera adaptation simply involves refining the classifiers from the generic classifiers of the reference architecture model to the heated camera system specific classifiers. This is illustrated in Figure 11 by the refinement of the estimator Port Group features to refer to the camera-specific Port Group classifiers. These Port Group classifiers are themselves extensions of the Port Group types in the MDS reference model that add Data Ports specific to the camera example.

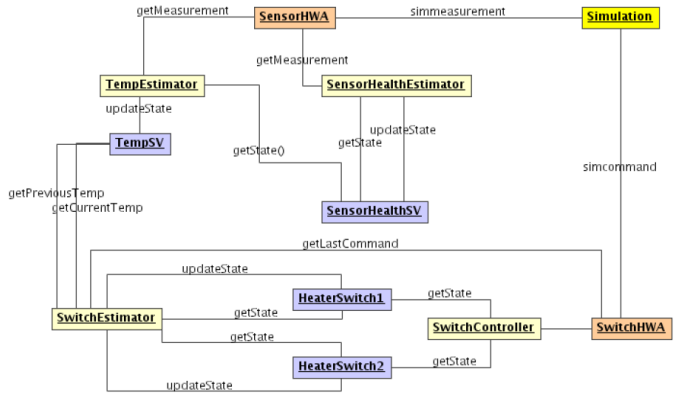



**Figure 12. Heated Camera Example Information Flow[12]**

Once the adaptation model structure was completed, it was possible to examine the information flow through the system. The information flow in the MDS control system is shown in Figure 12 as a collaboration diagram. State variables are read and updated through access methods. The heater switch controller takes heater goals as input and produces heater switch commands. This controller uses the estimated states of the heater switches to decide which command to issue. The switch estimator determines the state of the heater switches. Both the current and the previous state value may be used, as shown by the *SwitchEstimator* accessing both the current and previous temperature estimates.

The estimators make use of temperature measurements from the hardware adapters, temperature sensor health state, and heater switch state. The estimated state is available to other estimators and to controllers through "out" Data Ports, namely the *StateEstimatesOut* Port Group. This Port Group was refined for the heated camera example by creating Data Ports specific to this system. For example, the *StateEstimatesOut* Port Group is refined to include a Data Port called *Temperature_State*. Access to the current value of a state variable is represented by an immediate Data Port connection, while access to the previous value of a state variable is represented by a delayed Data Port connection.

### C. Flow Latency Analysis

This section presents the results of performing the Analyze activity of the AADL practice framework to the MDS adaptation example described in the previous section. As previously stated, one of the objectives of using a MB-SQA approach is to assure that certain quality attribute requirements are being achieved by the software system. In this example, the figure of merit "flow latency" is used to characterize the quality attribute "performance" for the heated camera system. This section presents the results of analyzing the heated camera system model with respect to flow latency, which gives an indication of whether or not performance requirements are being met.

Control systems process signal streams. Often a control algorithm is sensitive to the signal stream characteristics. These characteristics include the accuracy and precision of the sensor readings, bad or missing sensor readings, expected changes to successive values of the signal stream, the latency and age of the data in the

signal stream, as well as variation and jitter in latency and age. AADL provides the capability to model end-to-end flows and to utilize these end-to-end flow specifications to perform end-to-end flow latency analysis.

From a control engineer's perspective, end-to-end flow latency is comprised of (1) processing latency to perform the control computation, (2) sampling latency due to over- and under-sampling, and (3) transmission latency of the signal from the sensor and the signal to the actuator over physical connections. Furthermore, when a control system is implemented as software, several additional factors contribute to end-to-end flow latency including (1) sharing of processor and network resources, (2) preemptive scheduling, (3) blocking due to mutually exclusive access to shared logical resources such as shared data areas, (4) use of partitioned architectures, and (5) rate group optimization.

Processing latency refers to the amount of time it takes to perform a function. For example, the processing latency of a sensor is the amount of time between the detection of a signal and the corresponding response event or message from the sensor; in software, the processing latency of a software component refers to the amount of time it takes to compute the function. This time may be bounded by its worst-case execution time, which is a value often used in scheduling analysis to determine schedulability.

Sampling latency refers to the time delay that results from a task reading its input and then performing its computation at a specified rate. The maximum latency contribution due to sampling is the period of the recipient.

Control engineers are concerned with transmission latency over physical connections between the system under control and the control system. They often do not take into consideration any delays in communication between the software components in the control system. However, communication protocols contribute to latency as transfer requests from multiple sources are handled. In some protocols, latency takes the form of queuing delays as data from multiple producers is queued. In other protocols, latency takes the form of sampling delays as data ready for transfer must wait until its assigned slot in the protocol schedule is available.

In addition, the runtime architecture of the embedded software has a number of latency contributors. Preemption latency occurs when tasks share a resource. For example, multiple tasks may execute on the same processor, or tasks may require exclusive access to a shared data area. Typically, a deadline is specified for tasks to indicate the latest time since its dispatch by which it is expected to complete its execution. In essence, the deadline represents the worst-case sum of processing time and preemption time.

Partition latency occurs when different parts of the embedded application execute within different partitions, i.e., in different virtual machines on the same processor. Different partitions get different time slots to execute on the same processor. Communication between partitions is either non-deterministic, which results in latency jitter, or deterministic and therefore phase-delayed, which increases latency.

Finally, rate group optimization is used to reduce the number of separate threads and context switching between these threads by placing logical threads with the same rate in the same operating system thread. Consequently, the execution order of logical threads changes. If the logical threads communicate through shared variables, a change in the execution order may change what was intended to be mid-frame communication into phase-delayed communication thereby increasing the latency of the data being communicated.

The SEI has developed a flow latency analysis framework for AADL models that utilizes end-to-end flow specifications and knowledge about the control application execution.[13] This framework consists of a collection of application threads executing at a given rate and communicating their results via different communication mechanisms. This implementation of the flow latency analysis capability in OSATE was applied to the MDS adaptation models of the heated camera system.

An End-to-End Flow declaration *TemperatureResponse* was defined to represent a signal from the temperature sensor through the control system to the switch actuator device. This provides a measure of the time between the sensing of a switching threshold temperature and the reception of a command to turn the heater on or off by the switch actuator. The path is defined as an end-to-end flow through the *MDSControlSystem* of the heated camera, originating at the *TemperatureSensor* Device within the camera hardware (*MDSSystemUnderControl*) and ending in the *HeaterSwitch* Device within the camera

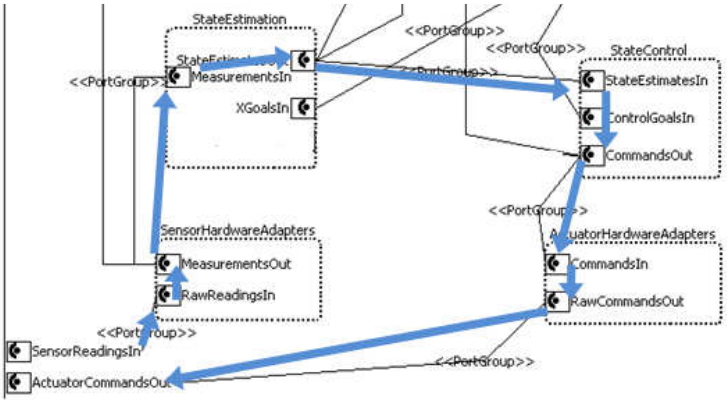



**Figure 13. The *TemperatureResponse* Flow through the Heated Camera Adaptation Model**

hardware (*MDSSystemUnderControl*). Figure 13 depicts the application of the *TemperatureResponse* flow specification to the MDS adaptation models of the heated camera system.

The flow through the *MDSControlSystem* has been specified using the "flows" keyword as shown in Figure 11. A flow specification indicates the flow from a component input to one of its outputs without having to expose or know its implementation. Flow specifications can have properties such as latency.

In the case of the *MDSControlSystem*, a component implementation with subcomponents was modeled and is shown in Figure 7. The component implementation declaration includes a flow implementation, which indicates how the flow specification is realized through the subcomponents. Figure 13 depicts the realization of the *TemperatureResponse* flow specification through the components of the *MDSControlSystem*. The flow starts with the sensor reading going to and through (1) the sensor hardware adapters, (2) the state estimators, (3) state control, (4) the actuator hardware adapters, and (5) finally ends with the actuator command output. The individual flow specifications have a latency Property associated with them that indicate the latency contributed by the component.

In the adaptation model of the heated camera system, the end-to-end flow declaration *TemperatureResponse* is elaborated by expanding the flow specification of the *MDSControlSystem* by its flow implementation. This expanded end-to-end flow is then interpreted by the flow latency analysis capability in OSATE. The latency analyzer calculates the end-to-end latency taking into account latency contributions by the runtime architecture as well as the computing hardware. It compares the results of this analysis with the expected latency property value of the end-to-end flow specification. Figure 14 shows how OSATE uses the Eclipse marker mechanism and problem view to report results to the user. This figure shows two error reports indicating that the calculated end-to-end latency has exceeded the expected latency.[§§]

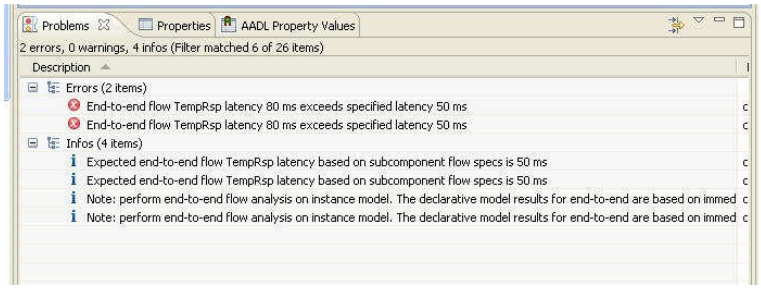



**Figure 14. Flow Latency Analysis Results for the *TemperatureResponse* Flow**

## V. Conclusion

Model-based software quality assurance (MB-SQA) provides a rigorous framework for the verification and validation of software systems through the systematic modeling and analyses of formal architecture representations. MB-SQA provides V&V and IV&V personnel with the capability of formally demonstrating that the architecture of their embedded software system meets quality attribute requirements. These quality attribute requirements are codified as Figures of Merit, or FOMs, which are measurable system properties that give an indication of how well the architecture addresses a particular quality attribute. This paper specifically addressed the application of the AADL quality assurance practice framework that utilizes the SAE Architecture Analysis and Design Language and supporting toolset OSATE for performing MB-SQA on embedded software systems.

This paper described the process of applying the AADL quality assurance practice framework to JPL's Mission Data System (MDS) reference architecture. The MDS is a unified reference architecture for space-mission flight, ground, and test systems. In the case study, the AADL assurance practice framework and several AADL-based analyses were applied to the evaluation of critical quality attributes of the MDS reference architecture as well as an MDS adaptation for the control of a heated camera.

As demonstrated in the paper, AADL can be used to effectively model key MDS architectural themes such as state-based control, the separation of estimation and control and the separation of data management and data transport, as well as top-level MDS constructs such as state estimators, state controllers, and hardware adapters. The ability to model these architectural themes and constructs provides the foundation for analyzing the MDS reference

[§§] Note that illustrative values were used for this model and the results are not indicative of the results for any existing MDS implementation.

architecture as well as adaptations of the MDS with respect to FOMs that give an indication of critical quality attributes.

The case study described in this paper involved modeling both the MDS reference architecture and an adaptation of the MDS as applied to the temperature control of a camera mounted on a fixed platform, which is a typical control problem on board a spacecraft. The quality attribute of interest was performance, and the specific FOM used to give an indication of performance was flow latency. The models of the reference architecture and adaptation were analyzed with respect to flow latency in order to ensure that the quality concern of performance was addressed.

It should be noted that these models can also be analyzed with respect to other figures of merit representative of performance such as scheduling and workloads as well as other quality attributes such as security and reliability. Future work in MB-SQA using the AADL assurance practice framework aims at exploring these other figures of merit and quality attributes.

In summary, the results of the case study demonstrate the utility of the practice framework and the AADL-based analyses in addressing (1) the modeling of key architectural themes for a reference architecture and (2) quality assurance with respect to performance, particularly flow latency.

## Acknowledgments

The work described in this report was funded by the NASA IV&V Facility Software Assurance Research Program (SARP) task titled "Model-Based Software Assurance with the SAE Architecture Analysis and Design Language (AADL)." Portions of this work were performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. This paper was also created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Special thanks go to the Mission Data System team at JPL, especially Matthew Bennett, Mitch Ingham, David Wagner, Kenny Meyer, and Bob Rasmussen.

## References

[1]Shaw, Mary and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, Upper Saddle River, NJ, 1997.

[2]IBM Corporation. IBM Rational Unified Process (RUP). 2003.

[3]Society of Automotive Engineers. "The Architecture Analysis and Design Language (AADL)." Standard AS-5506. November 2004. Revised Standard AS-5506A. January 2009.

[4]The Architecture Analysis and Design Language (AADL) Information Site. http://www.aadl.info.

[5]Feiler, P. H., D. P. Gluch, and J. J. Hudak. "The Architecture Analysis & Design Language (AADL): An Introduction." Carnegie Mellon University, Software Engineering Institute Technical Report CMU/SEI-2006-TN-011. Pittsburgh, PA. 2006.

[6]NASA IV&V Facility. "AADL Practice Framework for Model-Based Analysis: Beta Version." Project Report of the NASA Software Assurance Research Program (SARP) "Model-Based Software Assurance with the SAE Architecture Analysis and Design Language (AADL)." http://sarpresults.ivv.nasa.gov/ViewResearch/21/99.jsp. December 14, 2007.

[7]Feiler, Peter H., D. P. Gluch, J. J. Hudak, and B. A. Lewis. "Embedded System Architecture Analysis Using SAE AADL." Carnegie Mellon University, Software Engineering Institute Technical Report CMU/SEI-2004-TN-005. Pittsburgh, PA. 2004.

[8]The Open Source AADL Tool Environment (OSATE) for the SAE Architecture Analysis and Design Language (AADL). http://www.aadl.info.

[9]Dvorak, Daniel, Robert Rasmussen, Glenn Reeves, and Allan Sacks. "Software Architecture Themes in JPL's Mission Data System." Proceedings of IEEE Aerospace Conference, March 2000.

[10]Ingham, Michel, Robert Rasmussen, Matthew Bennett, and Alex Moncada. "Engineering Complex Embedded Systems with State Analysis and the Mission Data System." AIAA Journal of Aerospace Computing, Information and Communication, Vol. 2, No. 12, pp-507-536. December 2005.

[11]Society of Automotive Engineers. "Architecture Analysis and Design Language (AADL) Annex Volume 1." Standard AS-5506/1. June 2006.

[12]Bennett, Matthew, Daniel Dvorak, Greg Horvath, Michel Ingham, Richard Morris, Robert Rasmussen, and David Wagner. "State Analysis for Software Engineers: Model-Based Systems and Software Engineering." Jet Propulsion Laboratory, California Institute of Technology. http://mds.jpl.nasa.gov/public/index.shtml.

[13]Feiler, P. H. and J. Hansson. "Flow Latency Analysis with the Architecture Analysis and Design Language (AADL): An Introduction." Carnegie Mellon University, Software Engineering Institute Technical Report CMU/SEI-2007-TN-010. Pittsburgh, PA. 2007.