Reducing Uncertainty in Architectural Decisions with AADL

Kenneth D. Evensen
California Institute of Technology
Jet Propulsion Laboratory
kenneth.evensen@jpl.nasa.gov

Abstract

A model-driven approach to real-time software systems development enables the conceptualization of software, fostering a more thorough understanding of its often complex architecture and behavior and promoting the documentation and analysis of concerns common to real-time embedded systems such as scheduling, resource allocation, and performance. Key architectural decisions can be made early in the development lifecycle by analyzing quantifiable quality attributes related to these concerns. Two modeling notations, the Architectural Analysis and Design Language (AADL) and the Unified Modeling Language Profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE), are domain specific notations with the capacity to analyze these concerns. However, MARTE is not mature, to the point where its existing formalisms have been adopted into wide tool support. Furthermore, the inherent ambiguity of UML makes analysis in MARTE difficult. As a declarative language, AADL provides an adopted, formal analysis framework that meets this need. AADL can be used to augment MARTE in modeling software systems and provide the formal mechanisms for conducting quality analyses, helping to reduce uncertainty in architectural decisions.

1. Introduction

It is paramount to correctly specify the software architecture and design for safety-critical real-time systems. The model-driven approach to real-time software development focuses on conceptualizing the software and hardware components, making it easier to address stakeholder concerns early in the development lifecycle. Models can assist in assessing feasibility of implementation of both functional and quality attribute requirements. The two modeling notations presented in this paper, the Architectural Analysis and Design Language (AADL) and the Modeling and Analysis of Real-Time Embedded Systems (MARTE) UML profile, aim to accomplish this but with varying similarities and differences. MARTE does not provide the formal constructs necessary to conduct early quantitative and qualitative analyses, which causes uncertainty in architectural decisions. AADL fills this gap by providing strong modeling semantics and analysis framework. This paper provides two examples of how AADL can augment a UML model to formally analyze architecture qualities. Formal analysis is important for reducing the uncertainty about these qualities for which critical architectural decisions are based. First, it is important to understand the general background and originating purpose of each of these modeling languages. Next, an example involving a quantitative analysis and another involving a qualitative analysis are presented. The quantitative analysis example presents a concern, performance (specifically latency jitter) upon which an architectural decision has to be made. The qualitative analysis example involves detecting the sufficient condition for software deadlock in a model. These two examples are loosely adapted from a flight software project in its preliminary design phase. Latency jitter is a concern to this project due to the uncertainty in the thread scheduling mechanism of the inherited software architecture. The concern for deadlock stems from the project’s choice of interprocess communication architecture.

2. Background

The Architecture Analysis and Design Language (AADL) is an Society of Automotive (SAE) published standard, AS5506. AADL was formerly known as the Avionics Analysis and Design Language and is based upon MetaH (from Honeywell)\(^1\). AADL is first a language, with strict semantics, defining the underlying metal model. It uses a combination of graphical and textual modeling. Many details, such as user-defined properties, do not appear in the graphical model, but
appear strictly in the textual model. The graphical representation of a real-time system in AADL is closely coupled to the textual specification. AADL is composed of a relatively small set of modeling components to abstract the software and hardware entities found in a real-time embedded system. The manner, in which these components interact, for example AADL port connections, is also strictly defined. AADL does not provide a mechanism to change the meaning of the components and connectors. However, AADL does provide mechanisms, such as properties, that enable the modeler to detail the model and address specific concerns.

AADL models are independent of particular architecture viewpoints; the modeler has control over how entity relationships and architectural concerns are addressed as long as the semantics of the language are observed. Specifically, the modeler can choose which type of AADL components are modeled in each diagram, based on the desired level of abstraction.

Succinctly, AADL captures the execution nature of real-time software and hardware systems. It is often in the execution architecture that the concerns of performance, safety, and reliability are addressed. However, AADL does not address the compile time or explicit deployment nature of the hardware or software. If such views are required, another modeling language, such as the Unified Modeling Language (UML), can be used as part of the design description.

MARTE is a UML profile designed to handle real-time embedded systems and software concepts. UML was simply not designed to address concerns such as scheduling, performance, and time. MARTE’s underlying meta-model, based on AADL’s meta-model, provides the capability to address these concerns using UML notation. Various components from SysML (Systems Modeling Language), such as blocks and ports, are carried over into MARTE. MARTE does not introduce any new diagrams, as part of it’s meta-model. It does, however, include guidelines for constructing models at various levels of abstraction. The MARTE specification is in beta as of June 2009.

3. Quantitative analysis problem – latency jitter

The following example is a loose adaptation of an architectural issue found on a real flight software project. The project opted to use varying constructs in MARTE but mostly UML. This example relies mostly on the MRATE constructs. This example will further show how AADL can be leveraged to conduct a quantitative analysis.

Suppose a simple, multithreaded, embedded application such that performance is a concern. Specifically, the requirement states that, “The end to end latency jitter of the avionics system, in the primary configuration mode, shall not exceed 25 Ns.” The avionics system is to have two configuration modes, healthy and redundant. One mode will utilize a user-level thread to dispatch other concurrent process in the software space, by taking and releasing a semaphore. The other configuration mode will use a real-time interrupt to trigger the thread dispatch. The architectural decision in question is which configuration mode, the hardware or software dispatcher, should be used to trigger the concurrent processes. The software system will contain four threads:

1.) The user level scheduler
2.) A thread to poll a sensor device
3.) A thread to actuate a motor
4.) A health monitor to monitor the user scheduler and trigger the transition between modes

Figure 1 presents a basic context diagram for this software system.

![Figure 1. Context Diagram](image)

3.1 Problem explored

Figure 2 illustrates the component relationship of the same example, presented in MARTE. The MARTE profile in UML contains the constructs necessary to model a high-level architecture as well as the detailed design of both hardware and software components in an embedded real-time system. MARTE has the facilities to model both the software and avionics hardware platform components. Relationships between executing entities are described by ports. MARTE also employs connections to specify the implementation (Figure 2) and compile time relationships between software components.
Figure 2. MARTE Software Organization

Figure 2 is not an explicitly an implementation diagram, in that it does not depict software classes that would be used to implement each of the execution components illustrated. However, there is an implied implementation that can be modeled with additional diagrams. Furthermore, the hardware entities are modeled, but are omitted from the above package diagram for simplicity.

The <<swSchedulableResource>> stereotype is defined by the MARTE specification, and is applied to objects to model concurrently executing threads. In Figure 2, the <<swSchedulableResource>> is applied to the “myHealthMonitor” and “myScheduler” components, for example. This stereotype can be applied to a UML class or UML component, neither of which is explicitly indicated in Figure 2. In fact, the pieces represented in Figure 2 are UML components. Certainly, this point can be made in an associated narrative, or with the appropriate UML annotations. But taken out of context, this diagram leaves much to be interpreted.

Figure 3 presents the example from Figure 1 in AADL’s graphical notation. The graphical notation is secondary to the textual model. With the exception of Figure 4, the textual notation will be used throughout the rest of this paper. Taking this diagram “at face value” and assuming one understands the semantics of AADL, there is only one way this diagram can be interpreted. There is a single partitioned software space, “myProcess” in memory, represented by the solid parallelogram, an AADL process component. There are three “black-box” devices, “Sensor,” “Actuator,” and “RTI.” These are physically connected over a bus, “pciBus” to a processor, “myProcessor,” and memory “myEEPROM.” The software is logically connected to the hardware devices over AADL data and event data ports.

Clearly what are missing from this diagram are the four threads presented in the initial problem. Figure 4 depicts the implementation of “myProcess” from Figure 3. The solid lined parallelogram, “myProcess” in Figure 3, is a static partitioned software space in memory, and AADL process component. This is akin to the memory partition that an operating system allocates for an executable process. Inside this partitioned software space in memory (Figure 4) there are several software entities to note. The four dotted parallelograms, AADL thread components, denote concurrent executable processes, the four threads. The solid square, “mySemaphore”, is an AADL data component, which here is used to represent a shared object in memory. The two hexagons, “redundant” and “healthy,” illustrate the two configuration modes in which the software exists.

In Figure 4, the notion of configuration modes is captured in the AADL model. Capturing the same concept in MARTE (or UML for that matter) takes multiple diagrams.

Figure 5 illustrates the both the hardware and software concepts of this example in an internal block diagram, a SysML concept.
MARTE’s mechanism for specifying modes is a UML state-machine diagram, as shown in Figure 6. The events in the state machine are tied to the ports on “myHealthMonitor” in Figure 5. The implication here, though not as explicit as AADL, that activity on each port causes the appropriate transition.

The flow specification is described either by a sequence diagram, as shown in Figure 7 and Figure 8, or an activity diagram. Figure 6, Figure 7, and Figure 8 help to describe the behavioral nature of the system.

There are two issues here. The first being that there is no formal notation for tying the state machine diagram in Figure 6 to the ports, as triggers, on “myHealthMonitor” in Figure 5. There is the similar issue of connecting the states in Figure 9 to the flow specification in Figure 8. The second point of concern is the increasing number of diagrams necessary to represent multiple configurations. Due to the lack of formal relation between diagrams, this can cause an ambiguity in component relationships and behavioral interaction in the system, causing uncertainty in implementation.

AADL addresses these issues by embedding multiple configuration modes, as well as explicit flow specifications in the AADL text model. Figure 9 illustrates the implementation of modes in this example. Here, architectural dynamics refer to a modal configuration of components tied to an AADL event port (“myHealthMonitor.switch” and “myHealthMonitor.repaired”). The state of a system may be inferred from which components are active based on the modal configuration.

In this example, AADL flows are the mechanism used to assess execution jitter. In AADL, flow specifications provide for scalable end-to-end flows across component hierarchy. Figure 10 illustrates the end-to-end flow in the model. AADL flow specifications define data flow across sibling components. This doesn’t explicitly capture the configuration modes on the flows.
Since the modal configuration, in this example, is captured in “myProcess” as shown in Figure 4, the internal flow specification of “myProcess” contains the explicit connection between multiple modes and multiple flow configurations. This is shown in Figure 11.

![Figure 11. AADL Flows with Configuration](image)

In the interest of fairness, one key area that generic AADL fails to address is the behavioral aspect of the real-time software system. MARTE uses UML diagrams such as a state machine diagram and system sequence diagram, (e.g. Figure 6 and Figure 8 respectively). Once again, the number of diagrams needed to describe behavior increases with software system complexity. There is the additional problem of not having a standardized relationship mechanism between diagrams.

AADL addresses this shortcoming with the AADL Behavior Annex Library (A standardized notation by SAE). An annex library extends the AADL language to enable the modeling of entities not represented by the core language. As previously stated, AADL does not provide a mechanism to detail state-machine-like behavior by default; the Behavior Annex embeds textual notation of state-machine behavior in the AADL textual model. While not necessary for the latency jitter analysis, it may be useful to specify the conditions in which the mode change takes place in the example. Figure 12 illustrates a local variable, “x”, being evaluated such that if it is equal to zero, a signal is sent over the “switch” event port, causing a modal change.

![Figure 12. AADL Behavior Annex](image)

### 3.2 Analysis

Revisiting the underlying concern, performance, and indirectly measuring the requirement, “The end to end latency jitter of the avionics system, in the primary configuration mode, shall not exceed 25 Ns,” the model’s fidelity to the requirements can be assessed. An initial decision to use the software scheduler as the primary thread dispatcher is made, but may be changed based upon the outcome of the analysis. First, it must be determined whether or not the concurrent threads are schedulable on the processor, given its clock rate.

With AADL, OSATE provides a suite of automated tools to perform a variety of analyses. These tools perform the analyses based upon the embedded properties in AADL models. The first analysis that must be performed is to assess the schedulability of the four threads in the system on the one processor with a clock rate of 1 GHz.

Table 1 lists the properties used to conduct the automated analysis. The properties are assigned to each component in the AADL “properties” clause. Using the worst-case execution times, the load on the processor is 22%. Obviously the assumption is that if the system is schedulable in the worst-case scenario, then it is also schedulable in the best-case scenario. This indicates that the threads can meet their deadlines. Even though “myPeriodic” and “myAperiodic” threads are sporadic, they are assigned a period to indicate the mean time between executions.

<table>
<thead>
<tr>
<th>Property</th>
<th>Component</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch Protocol</td>
<td>myScheduler</td>
<td>Periodic</td>
</tr>
<tr>
<td></td>
<td>myHealthMonitor</td>
<td>Periodic</td>
</tr>
<tr>
<td></td>
<td>myPeriodic</td>
<td>Sporadic</td>
</tr>
<tr>
<td></td>
<td>myAperiodic</td>
<td>Sporadic</td>
</tr>
<tr>
<td>Period</td>
<td>myScheduler</td>
<td>125 Ns</td>
</tr>
<tr>
<td></td>
<td>myHealthMonitor</td>
<td>50 Ns</td>
</tr>
<tr>
<td></td>
<td>myPeriodic</td>
<td>125 Ns</td>
</tr>
<tr>
<td></td>
<td>myAperiodic</td>
<td>125 Ns</td>
</tr>
<tr>
<td>Compute Execution Time</td>
<td>myScheduler</td>
<td>2 Ns...5 Ns</td>
</tr>
<tr>
<td></td>
<td>myHealthMonitor</td>
<td>2 Ns...5 Ns</td>
</tr>
<tr>
<td></td>
<td>myPeriodic</td>
<td>2 Ns...5 Ns</td>
</tr>
<tr>
<td></td>
<td>myAperiodic</td>
<td>2 Ns...5 Ns</td>
</tr>
<tr>
<td>Deadline</td>
<td>myScheduler</td>
<td>5 Ns</td>
</tr>
<tr>
<td></td>
<td>myHealthMonitor</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>myPeriodic</td>
<td>15 Ns</td>
</tr>
<tr>
<td></td>
<td>myAperiodic</td>
<td>10 Ns</td>
</tr>
<tr>
<td>Actual Processor Binding</td>
<td>myScheduler</td>
<td>myProcessor</td>
</tr>
<tr>
<td></td>
<td>myHealthMonitor</td>
<td>myProcessor</td>
</tr>
<tr>
<td></td>
<td>myPeriodic</td>
<td>myProcessor</td>
</tr>
<tr>
<td></td>
<td>myAperiodic</td>
<td>myProcessor</td>
</tr>
</tbody>
</table>
If the schedulability analysis indicated the threads where not schedulable on the processor, decisions would have to be made to perhaps use a faster processor, or assess the thread attributes in Table 1.

Operating under the guarantee that the deadlines are met, a best and worst case flow latency analysis is conducted. The difference between the two results will be the latency jitter. This particular model follows a “data-driven” pattern as suggested by the AADL flow latency analysis framework [5]. This means that the execution of the flow is initiated by some periodic entity, either the “RTI” or “myScheduler.” The remaining data processing is done by aperiodic entities, whose execution begins on the receipt of data at the specified port. The AADL event data port is used to indicate a message queue mechanism is used to communicate. If the size of the queue were larger than zero, processing delay would be included in the calculation.

The worst-case latency is calculated as the latency of the initial source plus the deadline of each thread across the flow, added to the final latency of the flow sink. In the healthy flow, the worst-case latency is 0.03 microseconds, and in the redundant flow 0.045 microseconds. For the best-case latency, the deadline is substituted for the best-case execution time of each thread. Therefore, the best-case latency of the healthy flow is 0.006 microseconds and the redundant flow 0.019 microseconds. The jitter of the healthy flow is 0.024 microseconds and the jitter of the redundant flow is calculated to be 0.026 microseconds. The decision to use the software dispatch, as the primary scheduler was correct as its associated jitter is less than the requirement of 0.025 microseconds. Had this not been the case, the decision could be made to make the hardware real-time interrupt be the primary scheduler.

AADL has a very mature framework for conducting schedualability and latency analyses. The suite of available of automated analyses and supporting framework goes beyond schedualability and latency.

Even thought the MARTE specification does outline the framework to conduct analyses, as MARTE’s formalism is not mature enough, especially in the properties, where extensive tool support exists for conducting automated analysis. This particular example is not prohibitively complicated such that a hand-calculation could be conducted. In larger, more complicated models, certainly hand-calculations would be increasingly difficult. MARTE does provide the means to conduct qualitative analyses.

3.3 Reduction in uncertainty

At the presentation of the problem, the decision to be made was whether the primary thread dispatcher should be of software or hardware. The consequence of implementing the wrong decision is the sensor thread will be dispatched late, with the end result being the actuator will not be activated on schedule. Without the AADL model, suppose a decision was made to design and implement the primary thread dispatcher to be the hardware implementation. The next venue in which the software performance is evaluated is an end-to-end test-bed. The software will likely have gone through implementation and various testing (unit, integration, etc.) quite possibly without the performance issue manifesting itself. In the test-bed, the wrong decision would likely result in the faulty behavior of the actuator. The time it takes to diagnose and solve the problem is costly. The cost increases if an incorrect diagnosis is made.

The use of AADL, in this example, provides an opportunity during the architecture phase, to address the concern of jitter. Both data flow’s (software and hardware) latency characteristics are analyzed and it is determined, before implementation, that the software dispatch is the appropriate choice for the primary scheduler. Here, the use of a formal model notation, such as AADL, increases confidence in architectural decisions, which cannot be made with the same confidence exclusively using a notation like the MARTE profile.

4. Qualitative analysis problem – deadlock

Some quality measures in architecture are not quantifiable. It is simply important to assess if a particular condition is present that would cause a fault in execution. An example of this is deadlock. The interdependency of resources exists such that one resource cannot execute until it has access to a service provided by another resource.

The same project team, as mentioned above, chose to employ a publisher-subscriber architectural style. A thread, typically a periodic, would poll a device and publish a request for further processing on a queue. An aperiodic, subscriber thread waits for a notification that there is something on the queue to process. If this thread can handle the request it will do so and respond to the original periodic thread. If it cannot, it will become locked and pass the request onto the next aperiodic thread to further process the request.

The necessary condition for deadlock arises when there becomes a circular dependency of
servicing, aperiodic threads. If a request comes in to a blocked thread, this results in deadlock as it is waiting for a response to a request. Figure 13 illustrates a very simple example, but doesn’t explicitly describe the necessary for a deadlock. “T” is a periodic thread, illustrated in blue. “S,” “A,” and “R” are aperiodic threads illustrated in red.

![Figure 13 - Deadlock Block Diagram](image)

### 4.1 Problem explored

Figure 14 illustrates the same organization of components as Figure 13 with a more explicit depiction of executing entities. Four threads are presented (hardware entities are abstracted out in this example), a periodic thread “T”, and three aperiodic threads “S,” “A,” and “R.” Thread “T” is depicted to share some data with “S” over a port with the MARTE <<FlowPort>> stereotype. Similarly “S” shares some data with thread “T” over a separate port. The notion of a message queue can be applied to the connections using the MARTE <<MessageComResource>>.

![Figure 14 - Deadlock MARTE](image)

Unfortunately, the necessary condition for deadlock cannot be discovered just by examining Figure 14 as it does not speak to the actual order of execution of the software system. The actual flow of service requests needs to be analyzed. As stated earlier, a flow specification in MARTE requires the use of a system sequence diagram.

When visually assessing the four threads in Figure 15, it is possible to detect the necessary condition for thread deadlock. It is unlikely, and impractical to use a sequence diagram to model the dozens of possible threads in a system. Even if the flow specifications were broken across multiple diagrams, the ability to detect deadlock is hindered. As stated before, no automated analysis exists to check for deadlock in a MARTE model, so detecting the condition for deadlock would require some level of review.

In actuality no automated deadlock analysis existed for AADL either. However, AADL is supported by the Open Source AADL Tool Environment (OSATE), which is built in the Eclipse Rich Client Platform. This enables the development of custom plug-ins to address specific analysis needs of the modeler, in this case, the need to detect the necessary condition for deadlock.

Simply specifying a flow of data across sibling components isn’t sufficient either. AADL, by default, does not provide a way to indicate that a resource is blocked and waiting for a service from the next resource. However, custom properties can be created and embedded in the model to address these concerns.

Figure 16 illustrates the two custom properties that are used in this example. The event data port is used in this example to model the queue, and the flow of data is modeled using an AADL flow specification. The first property “BlockedBy” is applied to a thread specifying that when a message leaves the port, the thread becomes “blocked”. Similarly, the “ReleasedBy” property specifies that the thread becomes “released” on receipt of a message into queue in which it has subscribed (the in event data port). In Figure 16, thread “myStarter” is blocked by port “DC01” on transmission of a message. The thread is released on receipt of a message into port “DC02”.

![Figure 15 - Deadlock Flow Specification](image)
The analysis tool traverses the flow checking to see if any of threads have already been locked due to a request. If a thread is locked, the tool notifies the user. The project, being concerned about deadlock, now has the ability to assess if it could exist and detect where a fault might occur. Upon examination of the analysis results, if the necessary condition for deadlock is detected, architectural decisions can be revisited.

Again this is a relatively simple example. But the scenario becomes far more complicated when multiple flows utilize the same components in other, independent flows. A single UML system sequence diagram cannot model multiple orthogonal flows that use the same components between them. These independent flow specifications can be embedded in an AADL model. This custom analysis tool detects this condition as well. In execution, there might not be the sufficient condition depending on the timing of execution of the threads. Even detecting the necessary condition is important for alleviating uncertainty.

4.2 Reduction in uncertainty

The question here is, “Does the condition exist where a circular dependency among concurrently executing threads could create deadlock?” With three or four threads, one can possibly visually determine if a circular dependency exists. Consider a software system comprised of a dozen or more concurrent threads each with multiple data flows. It may not be as simple to ascertain whether or not the necessary condition for deadlock exists. Granted, in a nominal execution of the avionics system, even if the necessary condition for deadlock exists, a failure may never occur depending on which execution paths are exercised at particular point in time. Assuming the defect does exist, and AADL is levied to conduct the deadlock analysis, the software architects can be alerted to the condition. An architectural decision can be made to alter the data flows. Even being aware of possibly where the latent defect exists, and assuming the decision was made not to remove the defect, the cost to diagnose and repair a failure is reduced.

AADL provides the framework for creating new analyses, such as the deadlock analysis presented here, that assist in understanding the execution aspects of the architecture. From an assurance standpoint, identifying the conditions for deadlock help illuminate the riskier parts of the architecture.

This can guide early re-architecture avoiding costly redevelopment. MARTE’s basic analysis model is complex making standard formal analyses difficult. Creating new, formal analyses with MARTE is also difficult.

5. Conclusions

The two examples presented show how AADL can be used to augment a model centric approach conducted in UML, to reduce uncertainty in architectural decisions. If we believe a model centric architecture approach can facilitate early analysis of safety-critical, real-time software systems, the ability to analyze quality concerns, such as performance and timing, early in the software development lifecycle can drive more effective architectural decisions.

Due to the early development phase of the flight project in question, no results can be presented answering, “How much did AADL reduce the uncertainty?” As the project progresses, and the architecture matures, further research to answer this question will be conducted.

In order to accomplish this, a standardized, formal notation needs to be used to model the software system. AADL and MARTE two such notations that provide the capability to describe real-time embedded software and avionics hardware.

AADL’s strength is strong semantics coupled with tool support for automated analyses. This provides a foundation for a strongly architectured software system (reducing ambiguity) and rapid early quantitative analyses of quality attribute requirements. By architecting the execution entities, AADL can provide facilities to conduct a quantitative and qualitative assessment for feasibility of deployment, given avionics hardware. Assessing the feasibility of deployment during the architecture phase, and correcting decisions if necessary, is less expensive than doing so in the implementation or deployment phase of development. However, architecting the execution entities do not speak to the nature of the implementation of the software system.

MARTE does provide mechanisms for modeling implementation, execution and deployment of the real-time software system. There is the potential that a broad range of architecture decisions can be made early. MARTE’s underlying meta-model provides the semantics to adapt UML and SysML to the real-time software domain. Unfortunately the specification is extensive and complicated. This does have the advantage of being able to model a variety of real-time concepts such as operating system components (AADL doesn’t have the capability to do this), but assuring the model is
correctly specified according to the MARTE model is difficult.

Reducing uncertainty in architecture is essential for increasing quality in both the development and the product. A formal notation is required for modeling and analysis activities in order to accomplish this. AADL and MARTE are both poised to address this need. However, AADL is stronger in maturity and analysis capability.

There is a larger project level implication to making the correct architecture decisions early in the development lifecycle. Often, architectural decisions are made with uncertainty and assumptions such as, “We can test our way out of this if necessary,” or “We can fix it later.” Unfortunately, if the wrong decision is made in the architecture phase and re-architecture is later required to correct the decision, the ultimate impact at the project level is expensive. The consequences of architectural decisions do not affect a single software task, or software alone, but ripple across multiple interfaces. Therefore, it is important, from a software assurance standpoint, to reduce uncertainty in architectural decisions in the appropriate phase and increase the likelihood of making the correct architecture decisions early. AADL reduces uncertainty by correctly specifying the software architecture and providing analysis capabilities for early, confident decisions.

6. Acknowledgements

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. This research was funded by the NASA OSMA Software Assurance Research Program.

7. References