Evolution of an Avionics System

Peter H Feiler[1]
phf@sei.cmu.edu
October 29, 2007

This note summarizes the use of AADL in modeling and analyzing an Avionics system early and throughout the development life cycle. We will do this through a series of use scenarios. The example models can be found in IntegratedModel-1029007.zip. Unzip the file into a folder that represents an Eclipse workspace. Start up OSATE with this workspace. Invoke "Import.." from the file menu and "Existing projects into workspace…" and select the project. For further details see the installation note on "Installing the Example Models".

1 Organization as AADL Packages

First we describe the general organization of the AADL specification of the system as a set of AADL packages.

- AppTypes: This package contains a number of user defined data types from the application domain. In particular, the package contains the definition of port group types (PageRequest and PageReturn) that represent the interaction between subsystems for the purpose of requesting and displaying content information from various avionics subsystems. These port group types may initially be defined without detailing out the ports that they are composed of.
- HardwareParts: This package defines the types and implementations of hardware components such as processor, memory, and network (bus). The component classifiers have properties that specify resource capacities for the component as well as power requirements.
- HardwarePlatform: This package defines a number of hardware platform configurations that consist of one or more processors, memory, and network/bus connections. The platform provides access to a bus such that devices that are declared as part of the embedded application system can be connected physically to the processor(s). The hardware platform also includes an example of modeling a power supply.
- AppSubSystems: This package contain the specifications of the major subsystems of the avionics system. The subsystems are assumed to represent partitions. Therefore, they are represented as processes (protected address spaces for space partitioning) and have properties that specify its partition characteristics. AppSubSystem may initially provide a minimal parts specification (as shown for the WarningAnnunciationManager; the other subsystems have been expanded), that is later expanded to a full interface specification including ports and flow specifications. The subsystems also have resource requirements specified as resource budgets.
- FlightManager: This package contains several implementation declarations for the flight manager subsystem as a set of threads. One implementation represents a legacy

implementation where threads communicate through a common data area. Another implementation is a port and connection based representation of the interaction between threads within the flight manager. The threads have periods and worst-case execution time to support scheduling analysis.

- AppSystem: This package contains the application system with a collection of implementations that represent the system model at different levels of fidelity as well as variations of the system implementation. Interaction between the subsystems is specified through port group connections with bandwidth requirements for communication. Some implementations also include end-to-end flow specifications.
- SystemConfigurations: This top level package specifies a number of system configurations that consist of variants of the application system and variants of the hardware platform. Those system implementations are the root of system instances.

2 System Requirements and Parts Lists

In this scenario we illustrate a minimal AADL model of the avionics system that allows us to perform resource budget analysis. In this scenario we have defined the essential hardware parts (processor, memory, bus, device), and their composition into a hardware platform, as well as the application subsystems as part and their composition into an avionics system parts list.

2.1 The Model Specification

For this scenario we define the package *AppSubSystemsParts* with resource budget figures for processor resource requirements (MIPS) and for memory (RAM and ROM). We also define the configuration of these parts into an avionics system, which is reflected in the system implementation *EmbeddedApp.SubSystemParts* in package *AppSystems*.

For the hardware we define a processor type Xeon with two implementations (a solo core and a dual core implementation), a RAM and a ROM memory type, and an EtherSwitch as the high speed network type. By convention any memory type that represents ROM is expected to contain "ROM" in its name. The current implementation of the resource budget analysis plug-in utilizes this convention to identify ROM memory. The hardware platform as parts list is reflected in the system implementation *ComputingPlatform.ThreeProcessorParts* of the package *HardwarePlatform*.

The application system parts list and the hardware platform parts list are combined into a system implementation that is the root of a system instance. We have defined two system implementations in the package *SystemConfiguration*: *mysystem.parts* as a pure parts list configuration, and *mysystem.allocatedparts* as a system configuration in which application parts are allocated to hardware parts. The latter reflect a scenario where a requirements document may have specified an initial allocation of software components to processors or memory, or where a system architect may have made an initial set of allocation decisions.

Devices are AADL components that are logically part of the embedded application system and physically part of the hardware platform. We have placed their specification in the *HardwareParts* package, but have declared their instantiation as subcomponents in the application system. We have placed them in the application system because with a

focus on the embedded software system there tend to be more logical connections between the devices and the software components than between the devices and the processors that service them.

Note that we have chosen to represent the subsystems at this time as system components as we have not yet made a decision to map those subsystems into partitions.

2.2 The Budget Analyses

As part of the hardware component specification we have provided resource capacity of processors in terms of their ability to execute instructions per second (MIPS) and of memory in terms of their RAM and ROM memory size through *MIPSCapacity*, *RAMCapacity*, and *ROMCapacity* properties. Those properties have been defined as part of the *SEI* property set.

Similarly, for the application subsystems we have specified their processor and memory requirements as found in the requirements document, or their processor and memory budgets as assigned by the system architect to systems, processes, thread groups, and threads. They are specified through *MIPSBudget*, *RAMBudget*, and *ROMBudget* properties from the *SEI* property set.

MIPSCapacity and *MIPSBudget* take real values with units of KIPS, MIPS, GIPS. *RAMCapacity*, *ROMCapacity*, *RAMBudget*, and *ROMBudget* take real values using Size_Units as unit (B, KB, MB, GB).

2.2.1 Resource Budget Analysis

The intent of the resource budget analysis is to provide resource budgeting support early in the development life cycle. Users may have defined a system model that essentially is a high level parts list, i.e., defines the execution platform in terms of a collection of processors, memory, buses, and the application system in terms of major subsystems that may or may not be executed as separate partitions.

The components may not yet have features, nor do their implementations have connections. The application components may not have been elaborated beyond the first level of application subsystems, which may be represented as AADL systems, or as AADL processes if we know that they represent partitions with protected address space boundaries. The capacities and budgets can be associated with these components – with the type, implementation, or each subcomponent declaration.

The resource budget analysis works on system instance models. In other words you will first instantiate your model by selecting the root system implementation. Typically this system implementation contains two system components, one representing the execution platform, the other the application system.

In this scenario we instantiate the system implementation *mysystem.parts*. It can be found in the *SystemConfigurations.aaxl* file. Select *mysystem.parts*, and invoke the action *Instantiate System*. This action can be found in the toolbar, the context menu under *OSATE*, or in the *OSATE* menu.

The instance model represents a low-fidelity model of the system in terms of the system hierarchy, i.e., only one level of the system hierarchy for both the application and the hardware platform are shown.

We can perform a resource budget analysis on this model using the *Analyze Resource Budgets* action. This action is invoked through a toolbar command, or through the *Analysis* menu and the *Architecture* submenu.

This analysis totals up the resource capacities of all the processors and memories, totals up the resource budgets of all application components and devices, and checks whether the resource budget totals exceed the resource capacity totals. Note that these totals represent the total system capacity in terms of MIPS, memory, and bus bandwidth and the total system demand in terms of those measures.

The resource budget analysis also keeps track of the number of components for which no capacity or budget has been specified.

The results of the analysis are reported in two ways:

- As a summary in a dialog box
- As a collection of markers on the instance model resource.

In addition to recording the results shown in the dialog box as markers, each component without a capacity or budget is tagged with a marker. The results can be viewed through the Problem view. The Problem view can be filtered to show only the resource analysis results. This is done by selecting the *Resource Analysis Marker* in the Problem view filter. The marker results can also be turned into a report in HTML format. This is done through the *Generate Report* command in the OSATE menu.

2.2.2 Resource Allocation Analysis

The intent of the resource allocation analysis is to take into consideration allocation decisions of application components to processors and to memory. This allows the user to make allocation decisions by reflecting them in the actual processor binding property. Early in the development those decisions are validated in terms of MIPS and memory budget specifications.

In this case we instantiate the system implementation *mysystem.allocatedparts*. This system implementation is declared as an extension of *mysystem.parts* with binding information regarding allocation of software components to hardware components through a collection of binding properties (*Actual_Processor_Binding* and *Actual_Memory_Binding*). Those properties specify the processor or memory instance to which a component is bound to as property value and the component that the property value belongs to in an *applies to* clause. For example:

```
Actual_Processor_Binding => reference platform.MissionProcessor1
applies to App.FlightManager;
```

As in the previous scenario, the instance model is low fidelity in terms of the system hierarchy, i.e., it only represents subsystems of the application system.

We can perform a resource allocation analysis on this model using the *Analyze Resource Allocations* action, which is available through the toolbar or through the *Analysis* menu and the *Architecture* submenu.. This analysis takes into account the binding information.

It totals up the resource budgets of all application components that are bound to each processor instance and compares the total against the capacity of that processor. Similarly it totals the RAM and ROM budgets of components bound to a memory component and compares them against the memory capacity.

Note that in a system model the total resource budgets may be accommodated by the total resource capacity (*Resource Budget Analysis*), while specific allocations of components to processors and memories may cause some of the budget total to exceed the capacity of the hardware component they are allocated to.

If resource budgets exceed resource capacities we can carry out what-if scenarios. We can consider additional hardware resources, or negotiate less resource budget for specific subsystems. Through the resource budget analysis we know what the potentially smallest hardware configuration is that is necessary to support the resource budgets of the application system.

In case of the resource allocation scenario we can consider alternative allocations of application components to hardware components. These allocations may results in the need for a hardware platform with more capacity than the potentially smallest capacity identified under the resource budget analysis. These alternative allocations can be constructed manually and tested through resource allocation analysis.

3 Subsystem Interactions

In this scenario we elaborate the parts model from the previous scenario to specify the interactions between the subsystems. First, we add interaction points to each of the subsystems by declaring port groups as features. Second, we add port group connections to represent the fact that certain subsystems interact with each other without providing details of the interactions. Finally, we add bandwidth budgets to these connections to indicate the expected communication traffic between the subsystems. As in the previous scenario we consider a system model without allocation decisions and a system model in which allocation decisions have been made for application components.

We will then analyze the model as to whether the network load generated by the bandwidth requirements/budgets of the subsystem interactions can be accommodated by the network capacity.

3.1 The Model Specification

For this scenario we use as starting point the package *AppSubSystems* with minimal specification and elaborate it with port groups as interaction points. This allows us to model interactions between components without detailing out indivdual component interactions.

We make use of the package *AppTypes*, in particular the port group types. At this stage of the modeling process it is sufficient for the port group types to be defined without specifying the ports that make up the port group type.

For each of the subsystems of the avionics system, we add to the process type declaration the interaction points with other components as port group features. *Note: this is already*

reflected in the full process type specificaiton for most of the subsystems in the preloaded project.

We also define the configuration of these parts and their interactions through connection as an avionics system. This is intended to be a refinement of the parts configuration by the addition of connection declarations and by associating bandwidth budgets to these connections. These bandwidth budgets reflect the communication traffic between the respective subsystems. This system configuration reflected in the system implementation *EmbeddedApp.SubSystemTopology* in package *AppSystems*.

Similar to the previous case we also define top level system configurations. The first is *mysystem.communication*, which combines application configuration *EmbeddedApp.SubSystemTopology* with the same hardware platform configuration as in the previous scenario without allocation decisions. The second is *mysystem.allocatedcommunication*, which adds binding decisions to processors, memory, and one example of a binding decision of a connection to a bus.

3.2 The Bandwidth Analyses

In this case we are able to perform two analyses that focus on determining the workload that is generated by the connection bandwidth budgets and compares them against the bandwidth capacity of the network. The first totals the budgets and compares them to the capacity totals, while the second takes into account the binding of connections to buses to determine the workload.

Buses have bandwidth as a capacity. Connections have a bandwidth budget. We have introduced the properties *BandwidthCapacity* and *BandwidthBudget* for that purpose. These properties take real values using *Data_Volume_Units* as unit (bitsps, Bps, Kbps, Mbps, Gbps).

3.2.1 Analysis of Bandwidth Totals

In this case we instantiate the system implementation *mysystem.communication*. The analysis can also be run on the instance model for the workload analysis (see below).

The first analysis is similar to the resource budget analysis in that it totals up the bandwidth budgets of all connections and compares them to the bandwidth capacity totals of all buses. It does not take into account any allocation decisions of application components to hardware. For example, the bandwidth of a connection between two subsystems allocated to the same processor is included in the bandwidth budget total, although the connection may not generate any network traffic. This would correspond to all subsystems using a loopback protocol that utilizes an immediately connected bus even for communication within the same processor.

3.2.2 Analysis of Bus Workload

The workload of a particular bus is determined by the connection instances that are bound to a particular bus instance. This binding may be explicitly specified through the *Actual_Connection_Binding* property, or it may have to be inferred from the hardware component bindings of the connected components.

Each of the two variants supports taking into consideration loopback communication of connections within the same processor. In case of loopback communication, intra-processor communication is also routed through the network. This technique allows application components to be relocated to different processors without impacting the load on the network.

Inferred Binding Network Bandwidth Analysis

The inferred binding network bandwidth analysis algorithm works on instance models as follows. It identifies all connection instances whose endpoints (components) are bound to different processors. In other words, it expects application components with port connection instances to have an *Actual_Processor_Binding* value. If either component does not have such a property value the connection is not considered in the bandwidth demand calculation. If the endpoints are bound to different processors, then the connection is considered to be bound to any bus that physically connects the two processors, i.e., any bus that is accessed by both processors via *requires bus access*. The total of the bandwidth demand from connections is then compared against the capacity of the bus to which they are bound.

At this time the algorithm only considers buses that directly connect the two processors, i.e., a processor that is connected to another processor via two or more buses is not considered. If the two processors are connected directly by more than one bus, the analysis algorithm considers the connection to generate a workload on every such bus. This corresponds to the scenario where data is sent over all redundant network connections.

Explicit Binding Network Bandwidth Analysis

The explicit binding works on instance models for which port connection instances have an *Actual_Connection_Binding* property value. This value can be set in the model through a contained property association and must be a single bus instance. For example:

```
Actual_Connection_Binding => reference platform.Switch applies to
App.DisplayDM;
```

The analysis algorithm assumes that the connection binding is consistently specified, i.e., that the bus the connection is bound to actually connects the processors to which the connection endpoints are bound to. If this condition does not hold the bandwidth demand is still considered to contribute to the workload of the bound bus, but a warning marker informs the user of this inconsistency.

If either component that is the source or destination of a connection does not have an actual processor binding property value, the connection is not considered in the bandwidth demand calculation.

The Analysis

In this case we instantiate the system implementation *mysystem.allocatedcommunication*.

The bus workload analysis is invoked through the *Analyze Bandwidth Load for Bound Connections* command, available through the toolbar or through the *Analysis* menu and the *Architecture* submenu.

The bus workload analysis algorithm supports loopback analysis and a comparison of the workload with and without loopback. At this time the analysis algorithm considers all connections with the endpoints bound to the same processor as generating a workload on any bus that is accessed by the processor.

4 Power Consumption

In the third use scenario the modeler elaborates the hardware platform model by taking into consideration power consumption. This is illustrated in form of power draw from a powered bus by connected hardware components and in form of modeling a power supply.

We have introduced two properties, *PowerCapacity* and *PowerBudget*, for that purpose. Its values are real with *Power_Units* of mW, W, KW for wattage (an equivalent measure could have been impedance). The PowerCapacity can be associated with Bus components, while PowerBudget is associated with the *requires bus access* feature of the consuming component. As before, the property value can be specified with the component type, the component implementation, the subcomponent, or even component instances in the instance model.

4.1 The Model Specification

For that purpose we have elaborated the platform component description in package *HardwareParts* in two ways.

First, we have associated a power capacity (expressed in the *PowerCapacity* property) with the bus type *EtherSwitch*. Power budgets are associated with all *requires bus access* features of hardware components that connect to a bus of type *Etherswitch* and are expressed through the *PowerBudget* property. This represents a powered network or bus, and processors or devices that place impedance on the bus when connected to it.

Second, we introduce a bus type called *PowerSupply*. It represents a power supply with two implementations, one supporting 4.5V and the other 9.0V. The capacity of the power supply is expressed in Watts through the *PowerCapacity* property. Any hardware component that has power requirements will have a *requires bus access* declaration for bus type *PowerSupply*, indicating the voltage need by naming the appropriate bus implementation, and the power draw through the *PowerBudget* property. The component is then connected to the power supply through a bus access connection. This is illustrated in the hardware platform implementation *ComputingPlatform.TwoProcessor*, where a power supply is introduce and a processor connected to it.

4.2 The Power Analysis

The power consumption is reflected in the computing platform implementations for two and three processors, i.e., *ComputingPlatform.TwoProcessor*, and *ComputingPlatform.ThreeProcessor*. Those platforms are used in the system configurations *mysystem.communication*, *mysystem.allocatedcommunication*, and *mysystem.cmdflow*. Any of them can be used to run the power analysis.

The power consumption analysis (*Analyze Bus Power Draw*) is invoked in the instance model. It is invoked through a toolbar command, or through the *Analysis* menu and the *Architecture* submenu.

The power consumption analysis totals up the drawn power on each of the bus instances that have a power capacity and compares it to the power capacity of that bus. The results are reported via dialog and Resource Analysis Markers.

5 End-To-End Response Time in Partitioned Systems

In the fourth use scenario we add flow specifications and an end-to-end flow specification to the subsystem model. This end-to-end flow represents a critical information flow, a command issued by the pilot for information display from one of the major functional avionics subsystems. The modeler also annotates the subsystem model with information to reflect the decision to represent subsystems as separate partitions. This supports an end-to-end latency analysis to determine a lower bound for worst-case response time for such a pilot command for a system model at the level of fidelity of major subsystems.

5.1 The Model Specification

In this scenario we refine the *AppSubSystems* package by adding flow specifications to each of the subsystem type declarations. Each of the flow specifications indicates a logical flow through a subsystem from one of its incoming ports to one of its outgoing ports. Each of those flow specifications has a *Latency* property value to indicate the amount of time it takes for information to pass through this flow. *Note: The result is reflected in thepreloaded AppSubSystems package.*

In addition, we are indicating that the subsystems represent partitions. This is done in two ways. First, the category of the subsystem is changed to process in order to reflect the fact that partitions provide space partitioning through address space protection. Second, each subsystem is tagged with properties indicating whether they should be treated as a time partition (*IsPartition* property) and what the partition execution period is that results in phase delayed communication latency for cross-partition connections (*PartitionLatency* property).

The avionics system specification is then refined by adding an end-to-end flow declaration that specifies the end-to-end flow from the pilot display through various subsystems to the flight director and back to the display. This end-to-end flow specification has a *Latency* property value that specifies the expected *Latency* value. The system implementation with this end-to-end flow specification is shown in package *AppSystems* as system implementation *EmbeddedApp.CommandFlow*.

A top level system configuration called *mysystem.cmdflow* contains this application system implementation and a hardware platform.

5.2 The Flow Latency Analysis

The end-to-end flow latency analysis is performed in an instance model of the system representation. This instance model is created by opening the *SystemConfigurations.aaxl* file, selecting mysystem.cmdflow, and invoking the action *Instantiate System*. This

action can be found in the toolbar, the context menu under *OSATE*, or in the *OSATE* menu.

The analysis takes into account any latency value specified for flow specifications that are involved in the end-to-end-flow, as well as any latency value associated with connections involved in the end-to-end flow. In addition, the analysis considers the fact that cross partition communication results in latency due to phase-delayed communication across partition boundaries. The analysis assumes that all partitions execute on processors that operate with a common clock, i.e., are a globally synchronous system. Since partition execution is synchronized the partitions effectively can be viewed as sampling the data stream or pace the processing of event data at the maximum rate determined by the partition period. Incase of event or event data connections the algorithm also takes into account any delay due to time spend in the port queue.

The result is this end-to-end flow analysis is a worst-case end-to-end flow latency (or response time) figure that represents a lower bound on the worst-case response time. This lower bound is inherent in the application architecture, i.e., changes in the hardware will not reduce this figure. Such an analysis is useful early in the development cycle. If this figure already exceeds the requirements, we know that we have to change the application architecture itself. If the figure is close to the requirement for response time, we need to consider that allocation of the application to hardware may result in an increase in the latency.

Note: A full discussion of end-to-end flow analysis is found in a separate report.

6 Task Architectures for Subsystems

In the fifth use scenario one of the subsystems, the flight manager, is elaborated into a thread level model. In fact two variants of this thread level model are created: one that reflects the legacy implementation with thread communication through a common data area, and one that reflects signal and information flow through ports and connections.

Given the elaboration of one of the subsystems, we can perform a number of analysis. Some revisit the analyses performed earlier in the life cycle, such as budget rollup analysis and comparison of implementation actuals such as thread execution rates against budgets, or end-to-end flow latency analysis with a higher fidelity model to determine any impact of the flight manager implementation on the end-to-end latency. Other analyses focus on timing related engineering issues of the thread model such as priority inversion and schedulability of the thread with their given periods, deadlines, and worst-case execution times.

6.1 The Model Specification

We are expanding the AADL model of the avionics system by defining the details of the flight manager subsystem in a separate package *FlightManager*. By placing the details of this subsystem in a separate package, we can allow different people or teams work on different subsystems and their AADL specification separately at the same time.

The *FlightManager* package contains a collection of thread specifications. The package contains two sets of thread specifications: one in which all threads communicate through access to shared data; and a second set in which all threads communicate through data

ports. The first one represents a legacy implementation of the flight manager, while the second one is a modernized flow-based model that provides more precisely represents the actual system and provides more flexibility. The legacy threads include "Shared" in their name.

The legacy flight manger implementation is called *FlightManager.shared*. Its threads have priority assigned by hand that is represented by a *Priority* property. All threads have data access connections to the shared data area called *sharedData*.

The flow-based flight manager implementation comes in two variants. The first implementation called *FlightManager.PIO* includes a *PeriodicIO* thread, whose responsibility is to communicate data to and from other subsystems and do so at the beginning of every subsystem execution cycle. This is a legacy function. The second implementation called *FlightManager.NoPIO* accomplishes the same task without the PeriodicIO thread.

Both implementations have a thread *HandlePageRequest* that does not contribute to the flight management functionality, but services display requests from the pilot.

The threads have periods, and compute execution times. The thread execution times are set to values such that the collection of threads is not schedulable on a single slow or fast processor, but requires two processors to meet its deadlines.

Several top level system configurations make those application system variants available for instantiation. The legacy system is called *mysystem.legacy* and contains a three processor hardware platform. The other two system configurations are *mysystem.fmPIO* and *mysystem.FMNoPIO*. They can be reconfigured to use different hardware platforms with a single slow processor, a single faster processor, two processors, or three processors by changing the classifier of the *Platform* component.

6.2 The Analyses

This use scenario is elaborated into several sub-scenarios for analysis purposes. Several properties are expected to be associated with threads. First, the *Period* and *Compute_Execution_Time* properties are used in the to reflect execution timing information that can be compared against MIPS budgets. Second, the *RAMActuals* and *ROMActuals* properties can be used to record actual RAM and ROM requirements by the source code and data associated with a specific component. These properties can be assigned to threads as well as processes. The property values associated with a process reflect the actual RAM and ROM requirements of the process reflect the memory of that component only, i.e., are not cumulative. Each thread and data component in that process may have its own RAM and ROM actuals.

6.3 Resource Budget Analysis Revisited

The first sub-scenario revisits the resource budget analysis that was done earlier in the development process. In this case we have the resource budget of the flight manager subdivided to individual threads. During that design process the subsystem budget can be subdivided among the threads making up the subsystem.

During *resource budget analysis* those budgets are added up and their total is compared against the budget assigned to the subsystem. The cumulative figure is used in the rollup calculations of the total system budget.

During *resource allocation analysis* thread period and execution time, if specified for a thread are used to determine the resource demand for a thread. The result is compared against the thread budget and is used in the budget rollup. If a thread does not have a period and execution time the budget figure is used in the rollup.

During resource allocation analysis RAM and ROM actuals of threads are compared against their RAM and ROM budgets and are used in the budget rollup. If the actuals property value are not present the budget figures are used in the rollup calculations.

At this time the *Source_Code_Size* and *Source_Data_Size* properties are not used to determine the RAM and ROM actuals, because they do not distinguish between RAM and ROM requirements. However, their totals should be consistent with the sum of the RAM and ROM actuals.

6.4 Priority Inversion Due to Manual Priority Assignment

The legacy implementation of the flight manager uses a shared data area for communication and requires the threads to execute in a particular order. This order is specified through the *Priority* property.

The manual assignment of priority to preemptively scheduled threads introduce potential priority inversion. A *priority inversion checker* can be applied to an instance model of this system implementation that includes actual binding information of threads to processors. It will detect priority inversion caused by the assignment of a higher priority to a lower rate thread.

6.5 What-if Analysis to Meet Task Deadlines

The threads of the Periodic IO implementation *FlightManager.PIO* and the implementation without periodic IO (*FlightManager.NoPIO*) of the flight manager have been assigned execution times that make this subsystem not schedulable on a single processor.

The flight manager implementation with periodic IO has been configured in *mysystem.FMPIO* with a “slow” single processor computing platform. This is the configuration, whose instance model should be used for scheduling analysis.

Resource allocation decisions and scheduling analysis are performed through the *Bind and Schedule Threads* command, which is available in the toolbar and in the *Analyses*/*Scheduling* menu.

When the analysis is run, choose the first option in the dialog box. The result shows that the tasks overload the processor. The model contains comments that show how alternatives can be explored in a what-if analysis.

First, a faster processor can be considered. This is done by changing the choice of platform implementation to the platform implementation with the faster processor. This

configuration reduces the overload does still does not make the system schedulable. A change to a two processor system makes the system schedulable.

Note that the resource allocation and scheduling analysis takes into account that execution times may be different on processors of different type. The execution time is expressed in terms of a reference processor. The scaling factor is determined by the ratio of the *CycleTime* property that is specified for each processor implementation.

6.6 Response Time Revisited

The flight manager implementation has been elaborated with flow specification declarations that specify how the flow specification defined in the *FlightManager* type is realized in the *FlightManager* implementation.

We can run the flow latency analysis on the instance model of the system model with the elaborated flight manager with the periodic IO (*mysystem.fmPIO*) and without the periodic IO thread (*mysystem.fmNoPIO).*

The end-to-end latency figure of the *mysystem.fmPIO* analysis will be higher than that of the original end-to-end flow latency analysis, while the analysis of the *mysystem.fmPIO* system will result in the same latency figure. This demonstrates that the Periodic IO thread adds to the latency. The reason for this addition is that the partition communicaiton mechanism samples at the rate of partition execution. While executing within the partition as application code, the periodic IO thread does additional sampling, which increases the end-to-end latency.

6.7 Security Analysis

OSATE has a simple security analysis plug-in included. You assign a *SEI::SecurityLevel* property value to components and invoked the security analysis under the *Analyses/Security* menu. Two subsystems (DM and PCM) have already been assigned such a property.

There is also a prototype of a security analysis capability that supports security modeling concepts of Bell LaPadula and Chinese Wall. Please contact Jorgen Hansson (hannson@sei.cmu.edu) for further details.