Using AADL in Model Driven Development

Didier Delanote, Stefan Van Baelen, Wouter Joosen and Yolande Berbers
Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{didier.delanote,stefan.vanbaelen,wouter.joosen,yolande.berbers}@cs.kuleuven.be

Abstract

Software-intensive systems require the verification of functional and non-functional properties before the implementation and integration phases of the development process. In recent trends, the Architecture Analysis and Design Language (AADL) has proven a good candidate as a modeling language for software-intensive systems. At the same time, Model Driven Development (MDD) is gaining popularity as a development process. In this paper, we explore the use of AADL in a model driven development process from a usability point of view. Three issues regarding the usability of AADL as a modeling language for software-intensive systems are identified, namely system versus software level, complex component composition and property ambiguity. For resolving these issues, an approach is presented through integration of AADL models in a model driven development process with specifically designed model transformations. This approach enhances the usability of AADL for software developers.[*]

1. Introduction

As software-intensive systems are getting more complex, the verification of functional and non-functional properties of these systems is becoming even more complex. For example, mission-critical applications are often very complex and at the same time need a rigorous analysis and verification of its non-functional properties, also called quality attributes. Non-functional properties of interest for these systems are, amongst others, timeliness, performance, and safety properties. Verification of these properties is often delayed until testing phase, resulting in a slow and expensive development process, or, in the worst case, failed projects. Verification of non-functional properties in early design phases of the development process is therefore more important than ever.

First in the set of techniques that can enable this kind of verification, is an adequate modeling language. To describe the architecture of the system, a modeling language should allow describing both functional and non-functional properties of the architecture. To describe functional properties of a system, various modeling languages can be chosen. UML [1] is a widespread modeling language standardized by the OMG, which provides concepts to model the architecture, behavior and deployment of software systems in object-oriented or component-based paradigms. Moreover, the language can be extended through profiles, accommodating domain-specific modeling concepts. For example, SysML [2] is a profile to describe systems engineering applications. Non-functional properties are harder to describe using UML. The MARTE profile initiative [3] tries to extend UML for improving the Modeling and Analysis of Real-Time and Embedded systems. In addition, because the UML is an extensive standard, the language can be difficult to apply in tools and development processes such as Model Driven Development.

Architecture Description Languages (ADLs) aim specifically at modeling both functional and non-functional properties of system and software architectures. Making the description of these properties explicit enables analyses of the architectures. A number of ADLs have been proposed over the last decade [4]. From these proposals, the Architecture Analysis and Design Language (AADL) [5] has received increasing interest from mission-critical development industries. AADL has been standardized by the Society of Automotive Engineers (SAE) to support analysis and design of complex real-time safety-critical applications. AADL provides a standardized textual and graphical syntax for describing architectures with functional interfaces and for performing various analyses towards non-functional properties of the system being developed. In this paper, we provide an assessment of AADL as a modeling language from a usability point of view. We consider a representative modeling example and identify a number of issues in the usability of AADL for system developers.

Second in the set of techniques that can enable verification of non-functional properties on systems, is a

[*] The described work is part of the EUREKA-ITEA SPICES project, and partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders).

property-preserving MDD process. Model driven development allows using models as a primary artifact in the development process. In this way, models enriched with both functional and non-functional properties can be used to perform prediction of non-functional properties and validation of requirements, at a high level of abstraction. In this paper, we will assess the usage of AADL as a modeling language in a model driven development process, and propose an approach to improve the usability of AADL in such development context.

This paper is organized as follows. After the introduction, we present AADL and its most important modeling concepts in section 2. In section 3, we introduce an example to assess the usability of AADL and identify three *issues regarding its usability*. In section 4, we provide an overview of a model driven development process providing an approach for improving the usability issues identified in section 3. In section 5, we state related work. Finally, we draw conclusions in section 6.

2. Overview of AADL

AADL aims to describe the architecture of embedded real-time systems in an abstract but precise way. For this purpose, it provides a number of modeling concepts at a level of abstraction different than the level of implemented systems. AADL focuses on the description of systems using the component-based paradigm. Component specifications are made through modeling concepts in a number of dimensions. In this section, we list six dimensions AADL provides for specifying components.

2.1. Component analysis and design

As its name states, AADL can be used for both analysis and design of embedded systems. For this purpose, AADL uses components as modeling concepts. On the one hand, existing embedded systems can be analyzed in terms of components with a number of functional and non-functional properties. On the other hand, embedded systems can be newly designed in terms of components, which can be treated as black boxes. These are two separate usages of AADL as a modeling language, both using components as an appropriate modeling concept. Component analysis and design makes for a first dimension of modeling concepts, specified through properties. Component properties will be explained further on in section 2.4.

2.2. Component categories

A second dimension of modeling concepts is component categories. Components in AADL belong to one of ten predefined component categories. These ten component categories model the execution platform, the application software or composite components. AADL models hardware and software components in exactly the same way.

Execution platform components model the hardware part of the system, and belong to the *processor*, *memory*, *device,* or *bus* category. Each of these categories has specific semantics defined in the SAE AADL standard [5]. A *processor* component is an abstraction of hardware and possibly embedded software that schedules and executes threads. It may contain memory, and can access memory or device components through a bus component. Examples of a processor component are a hardware central processing unit or a PowerPC component with embedded Linux operating system. A *memory* component is an abstraction for a component that stores data and code, whether in randomly accessible physical storage or complex permanent storage. Because a memory component has a physical presence, it must have a specific number of properties, which will be explained further on in this paper. A *device* component is an abstraction for a component with complex behavior that interfaces with and represents a part of the external environment. Complex behavior represents the possibility of a device containing physical processor, memory and application software components that are not explicitly modeled. Again, because a device component has a physical presence it must also have a specific number of properties. A *bus* component is an abstraction for an execution platform component which provides communication of data and event messages between processor, memory and device components. Components requiring access to a bus must declare this access through specific syntactical constructs, which we will not further detail here.

Application software components belong to the *process*, *thread*, *thread group*, *subprogram,* or *data* category. A *process* component models a protected virtual address space. Contrary to the process concept in programming languages, such as Ada, a process in AADL is not necessarily executable. The process is considered executable if its implementation, which we will address further on, contains at least one *thread*. A *thread* component is an abstraction of a schedulable unit of concurrent execution. Properties of a thread are used to specify the processor on which the thread executes and the scheduling protocol used for execution. Threads can be logically grouped into *thread group* components. A thread

group component provides a single point of reference to multiple threads with the same properties. A *subprogram* component models a procedure call as in imperative programming languages. It allows modeling an entry point in a thread component. Subprograms can also be part of a data component, in which case this component models an abstract data type. A *data* component is used to model data structures stored in memory or exchanged between components.

Composite components model components consisting of both hardware and software. A *system* component models a component containing execution platform, application software and other composite components. Systems add hierarchy in the description and are at the highest level in this hierarchy. At lower levels of the hierarchy are subcomponents that belong to one of the ten possible component categories.

2.3. Component type and implementation

Besides a category, components in an AADL description have a *type* and *implementation(s)*. A *component type* describes the functional interface of the component and specifies what is visible on the outside of the component. A *component implementation* describes how a component realizes its interface and specifies the internal structure of the component. A component type can have several implementations. To describe both component type and implementation in more detail, AADL provides a number of specific modeling concepts.

A *component type* describes its interface in terms of *features*, *flows* and *properties*. *Features* specify how the component interfaces with other components in the system. A feature is a *port*, *subprogram*, *parameter,* or *subcomponent*.

A *port* feature models a logical connection point between components for directional transfer of data, events, or both. Port features model asynchronous communication. Depending on what is transferred over the port, a port can be either a *data port*, *event port* or *data event port*. Port features also have a direction, which is *in*, *out* or *in out*. Ports can be logically grouped into port groups. A *port group* component provides a single reference for multiple ports with the same properties. A *subprogram* feature models a procedure call, as explained in section 2.2. Subprogram features are used to model synchronous communication. Components accept *parameter* features. Parameters are directed *in*, *out* or *in out,* and are typed with a data component type, as explained in section 2.2. Parameters are comparable to port features, but are specific to subprograms. A *subcomponent* feature models the instantiation of a component type or implementation, similar to objects being instances of classes in object oriented programming languages. Subcomponent features enable nesting of components. Besides features, a component type can also be specified using flows. A *flow* through a component is an abstraction for a channel of information transfer through or between components. Flows are always from port to port, and have a source, path and sink. Finally, a component type can be annotated with *properties*. Properties provide information about the component and will be explained in more detail in section 2.4.

A *component implementation* specifies the internal structure of a component through connections, flows, modes, subcomponents and properties. A *connection* is a linkage between component features that represents the communication of data and control between components. AADL supports three types of connections. A port connection represents the transfer of data and control between two concurrently executing components, i.e., between two thread components, or between a thread and a processor or device. Parameter connections are an abstraction for the flow of data through the parameters of a sequence of subprogram calls. Access connections designate access to shared data components by concurrently executing threads, or by subprograms executing within a thread. Flows have been explained higher on, and, similar to components, can have a specification and an implementation. A flow specification, declared in a component type, models the existence of a flow path from source to sink. A flow implementation is specified in a component implementation and provides information about how this flow is realized. A *mode* represents an operational state of a component. A component can have mode-specific configurations of subcomponents and connections, and mode-specific property value associations. In this way, mode transitions model dynamic operational behavior that represents switching between configurations and changes in components internal characteristics. One of the component modes must be declared as being the component's initial mode. As for component types, subcomponents of a component implementation allow nesting of components, and properties allow providing information about the component.

2.4. Component properties

A property provides information about component types, component implementations, subcomponents, features, connections, flows, modes, and subprogram calls. A property has a name, a type, and a value. The property type specifies the set of acceptable values for a property and is one of Boolean, integer, float, string, enumeration, or component reference. Properties can also

have a measurement unit, such as time units. Property declarations can be grouped into property sets. AADL provides a standard set of properties applicable to all ten component categories. These properties are used to specify bindings between software and execution platform components, protocols for connections and scheduling threads, execution deadlines and periods for threads, transmission times for buses, etc. AADL also allows extending the language with user-defined properties. In this way, properties can annotate a model of the system or application with information about its quality attributes. Properties allow analyses on the model using external tools. Tools calculate a prediction of a quality attribute of the system, taking into account component properties and component composition, as will be explained in the next section.

2.5. Component composition

The SAE AADL standard provides a number of legality rules for component composition. Legality rules state for each component category what elements are legal for its component type and component implementation. In the example of a data component implementation, the standard defines that data subcomponents, access connections, modes, and properties are legal, whereas subprogram calls and flows are not. As was mentioned in section 2.4, properties are specific to component categories and are also mentioned in component composition rules. Composition rules are specific to each component category. Component declarations can also extend other components, which means composition rules need to take into account component extension.

2.6. Component connection and binding

Components can be connected and bound to each other in a number of manners. Component connections are used for communication of data and events, and are specified through connection, flow, and port features, as mentioned in section 2.3. Application software components can be bound to execution platform components through properties, as mentioned in section 2.4. The component connection and binding dimension is thus orthogonal to the component type and implementation dimension on the one hand, and the component properties dimension on the other hand.

3. Usability assessment of AADL

To assess the usability of AADL as a modeling language, and identify potential usability issues, we have used AADL to elaborate the example of an avionics autopilot system, which is based on Hamid [6].

3.1. An example

The autopilot system modeled in this example consists of a process for navigation control, a number of processing elements, and a number of devices providing position information and controlling aircraft movement. The process for navigation control in turn consists of three threads that periodically read GPS data, compute aircraft control data, and read external parameters respectively. A high-level architecture of the system is presented in Figure 1.

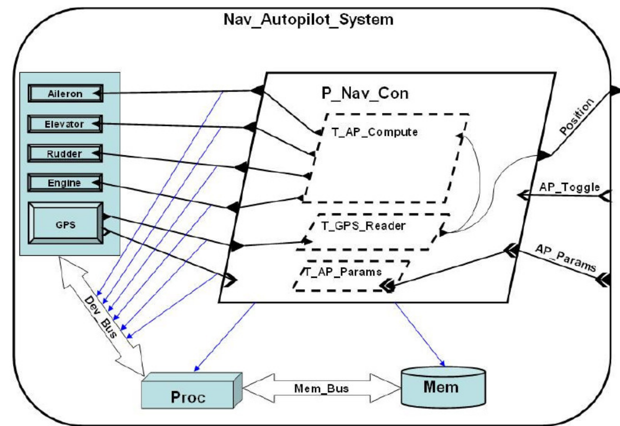

Figure 1: Autopilot system in AADL graphical notation

Figure 1 shows how devices, threads, and processing elements are connected in the system called `Nav_Autopilot_System`. Processor and memory elements, called `Proc` and `Mem` respectively, are connected through a bus called `Mem_Bus`. The devices controlling aircraft movement are called `Aileron`, `Elevator`, `Rudder`, and `Engine`. These are connected to the `P_Nav_Con` navigation control process through ports. A `GPS` device provides location data as input to the `P_Nav_Con` process. Connections between these components are bound to the `Dev_Bus` bus component. As was mentioned, the `P_Nav_Con` process consists of three threads called `T_AP_Compute`, `T_GPS_Reader`, and `T_AP_Params`. This process is bound to the `Proc` processor and `Mem` memory components. Further on, there are `Position`, `AP_Toggle` and `AP_Params` connections providing input and output between the systems interface and the `P_Nav_Con` process.

The graphical representation of Figure 1 can now be represented in textual syntax of AADL. We concentrate on the description of the system itself. In Figure 2, we provide an excerpt of the textual syntax of AADL for the system component. For reasons of clarity, a number of details have been left out of this textual specification.

We discuss how the modeling dimensions, mentioned in section 2, apply to this example. In Figure 2, the use of AADL as a modeling language is in the design extreme of the component analysis and design dimension explained in section 2.1. This can be asserted by the absence of analysis specific properties such as `Source_Language` and `Source_Text` on the system. The component categories dimension can clearly be identified through selection of the system category for the `Nav_Autopilot_System` component. The component type and implementation dimension is identified through separation of modeling concepts in `system` and `system implementation` component specifications. The component properties dimension is identified through use of a number of properties on the system component implementation. The component composition dimension is identified through instantiation of subcomponents in the system implementation. The component connection and binding dimension is identified through connections between subcomponents and bindings through properties of the system implementation.

```
-- Reads GPS data, sends control
-- signals to the control surfaces
system Nav_Autopilot_System
features
         AP_Toggle : in event port;
         AP_Position_Input : in event data
         port Nav_Types::Position.GPS;
         --…
end Nav_Autopilot_System;

system implementation
  Nav_Autopilot_System.PowerPC_G4
subcomponents
         -- Declare subcomponents
         Proc : processor PowerPC.G4;
         Mem : memory RAM.Generic;
         Dev_Bus : bus Device_Bus;
         Mem_Bus : bus Memory_Bus;

         -- create process
         P_Nav_Con : process
         Nav_Control_Process.PowerPC_G4;

         -- Declare four actuators
         Engine_RPM_Controller : device
                  Actuator;
         Aileron_Controller : device
                  Actuator;
         Rudder_Controller : device
                  Actuator;
         Elevator_Controller : device
                  Actuator;
connections
         -- bus access connections
         bus access Mem_Bus ->
                  Proc.Mem_Bus;
         --…

         -- connect bus to actuators
         bus access Dev_Bus ->
           Engine_RPM_Controller.
                  Connector_Bus;
         --…

         -- Connect GPS device to ports
         data port GPS_Loc.Position_Output ->
           P_Nav_Con.GPS_Position_Input
           {Actual_Connection_Binding =>
           reference Dev_Bus;};
properties
         -- Bind P_Nav_Con to processor
         -- Proc and memory Mem
         Actual_Processor_Binding =>
         reference Proc applies to P_Nav_Con;
         Actual_Memory_Binding =>
         reference Mem applies to P_Nav_Con;
end Nav_Autopilot_System.PowerPC_G4;
```

Figure 2: Autopilot system in AADL textual syntax

Using this example, we identify a number of issues in the use of AADL as a modeling language.

3.2. System versus software level

AADL uses the component-based paradigm, in which components are the primary modeling concepts. Szyperski [7] defines a component as follows: *"a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only."* Bachman et al. [8] extend this definition by stating *"in a component-based system (i) components and frameworks should have certified properties; and (ii) these certified properties should provide the basis for predicting properties relative to the whole system built out of those components."* From both these definitions, it is clear that AADL has succeeded in building the component-based paradigm into a modeling language for the description of both hardware and software systems. AADL forces system developers to describe a system in terms of components, which enables analyses on the system allowing the prediction of functional and non-functional properties.

AADL is therefore well suited for use by system developers. At the same time, it implicitly captures a modeling methodology, by further splitting up systems into components of a certain category. Categories map to functionality of components rather than concepts in which they are implemented in a specific implementation language. Software components may be implemented in an object-oriented programming language instead of a language using the component-based paradigm. In this case, components can be made up of several classes and, hence, are at a higher level of abstraction. Software developers interested in using AADL face a gap between modeling concepts and implementation concepts, caused by a gap between system and software level. This gap may increase the entry level to the use of AADL.

As an example, to software developers new to AADL, it may not be clear how the system component in Figure 2 is to be implemented. An implementation of this component could be a class in an object oriented language, it could be a hardware component specified in a hardware description language, or it could be a mix of software and hardware components.

3.3. Complex component composition

As stated in section 2.5, AADL allows composing components according to legality rules mentioned in the SAE AADL standard. Legality rules are specific to every component category. Component composition allows adding hierarchical structure to the description of systems. Because the legality rules stated in the SAE AADL standard are different for each component category, component composition is quite complex. Moreover, components can be extended, which adds complexity to composition. It may be unclear to developers what component can be composed with other components.

As an example, we consider the four device subcomponents of the `Nav_Autopilot_System` system component in Figure 2. Following the SAE AADL standard devices may have software accessing it executing on a processor, but it is not possible to model this software as a process or thread subcomponent of the device. From a software developers point of view, this may come as a high level of abstraction, whereas system developers may consider this close to system structure. As can be noted, the component composition issue is related to the system versus software issue from section 3.2.

3.4. Property ambiguity

Components in AADL can be annotated with properties, expressing non-functional properties of the component. As was explained in section 2.4, legality rules in the AADL standard define which component categories are allowed to have properties, and which properties apply to each component category. A complete AADL model annotated with a correct set of properties allows executing analysis of a non-functional property on the system model.

Unfortunately, there is no clearly defined relation between property sets and model analyses. The SAE AADL standard does not clearly state what properties need to be annotated in an AADL model to analyze the model for a given non-functional property. Unclear relations between component categories and properties on one hand, and between analyses and properties on the other hand, make for two causes of property ambiguity.

As an example, it is unclear what properties we need to add to the system and its subcomponents mentioned in Figure 2, to allow executing a schedulability analysis on the system. Several properties would need to be added to the thread and processor components concerning period of threads and processing capacity of the processor, but it is unclear from the standard exactly what properties need to be added.

3.5. AADL as a platform

Issues in the usability of AADL as a modeling language, identified in sections 3.2 to 3.4, increase the entry level for the use of AADL, and decrease feasibility of the use of AADL in a development process. If we consider AADL and its runtime environment as being a platform, this means the use of AADL requires a considerable amount of platform specific knowledge. In the OMG MDA guide [9], a platform is defined as *"a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented."* In the following, we propose a model driven approach that reduces the requirement of this knowledge, and increases feasibility of the use of AADL by considering AADL as a target platform in an MDD approach.

4. A general approach to improve usability of AADL using model driven development

In recent years, a lot of research has been done in the field of model driven development (MDD). The OMG has constituted to a high degree in the assessment of model driven development through the standardization of the Model Driven Architecture [9] initiative. In an MDD approach, models are used as primary artifacts throughout the software development process. Models of the software application are used at different levels of abstraction. In a typical MDD approach, three levels of abstraction can be identified. At the highest level, a platform independent model (PIM) of the application is built. Through model transformation, this PIM is transformed into a platform specific model (PSM). The PSM contains platform-specific details for the application towards the targeted execution platform and hence, is at a lower level of abstraction than the PIM. Platform-specific details are added by transformation rules that make up a model transformation from PIM to PSM. Finally, at the lowest level of abstraction is code in a platform specific implementation language. Again, implementation details are added through model transformation from PSM to

code. In this manner, platform specific knowledge is moved from the model to the model transformations.

After identifying in section 3 that the use of AADL requires a considerable amount of platform specific knowledge, we consider the combination of AADL and MDD as a particularly interesting approach for providing a partial solution. In the following, we consider the AADL runtime environment implemented in an analysis tool as the targeted platform in an MDD approach. At the highest level of abstraction is an incomplete AADL model of the software application being developed. We will refer to this model as the AADL PIM. Acquiring the AADL PIM is explained in section 4.1. The AADL PIM may not comply with the AADL semantic requirements as stated in the SAE AADL standard. One level down is an AADL model of the software application which fully complies to AADL semantic requirements stated as legality rules. In the following, we will refer to this model as the AADL PSM. The AADL PSM is acquired from the AADL PIM through model transformations, which are explained in section 4.2. Finally, at the lowest level of abstraction is an AADL model annotated with all necessary properties to analyze the application for a specific non-functional property. We will refer to this model as the AADL analysis model. The ultimate AADL analysis model is acquired from the AADL PSM through a number of model transformations, which are explained in section 4.3. Finally, a note on how to integrate these steps is given in section 4.4.

4.1. Obtaining an AADL PIM

As was mentioned in section 1, UML has become a widespread modeling language in software development. Although the UML may not be specific enough for some purposes, it offers the advantage of having a lower entry level. To use AADL with UML, the SAE AADL standard offers an annex describing the standardized UML 2.0 profile for AADL.

We propose the obtainment of an AADL PIM as a UML model annotated with AADL stereotypes according to the UML profile for AADL. The obtainment of this model is divided into two phases. First, a UML model is obtained that can be the result of an analysis or design phase early in the development process of the application, or it can be the result of reverse engineering an existing application. Second, this model is annotated with UML stereotypes identifying component categories for the application. A UML model annotated with a number of stereotypes makes for an AADL PIM which can be used as input for a functional transformation, as explained in section 4.2. This approach of obtaining the AADL PIM allows using AADL for both extremes of the component analysis and design dimension, as defined in section 2.1.

As a proof of concept, we have modeled the autopilot system from section 2 as a UML diagram, representing the AADL PIM of the application. This diagram can be seen in Figure 3. Components have been modeled as UML classes, after which stereotypes annotating the component category have been applied on each of the components. Besides classes and stereotypes, the model is annotated with UML compositions, directed associations and dependencies [1]. These three types of relations are also used in the functional transformations, as explained in section 4.2.

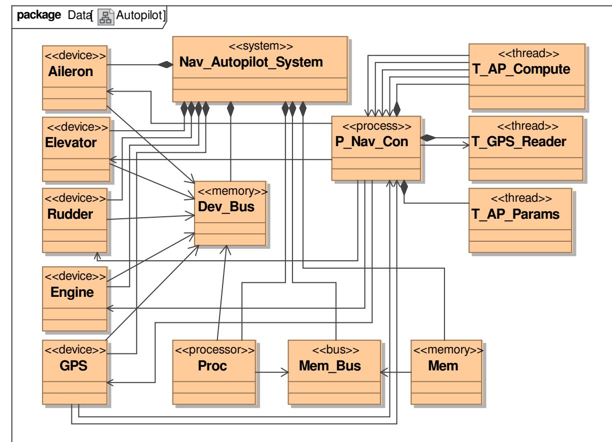

Figure 3: AADL PIM of the Autopilot system

4.2. Functional transformations

An AADL PIM annotated with component category stereotypes does not comply with SAE AADL standard legality rules yet. To comply with legality rules for component composition, as explained in section 2.5, the AADL PIM should be transformed using model transformations. Component composition rules are related to functional properties of the software application. In this paper, we use the term functional transformations for model transformations enforcing functional properties on the software application. The result of functional transformations is the AADL PSM of the application, which does comply with all SAE AADL standard legality rules. Platform specific knowledge, in this case enforcing SAE AADL standard legality rules, is built into functional transformations.

As an AADL PSM is more platform specific, instead of UML we use a domain-specific language to denote the PSM. Domain-specific languages allow adding more detail to modeling concepts. In our approach, we implemented this domain specific language using Ecore [10]. Models complying with an Ecore metamodel are

supported by a XMI-file format representation. In this way, AADL PSM models are interoperable with tools such as OSATE [11]. OSATE has a built-in semantic check, which allows checking compliance of the AADL PSM with AADL legality rules.

Functional transformations consist of a number of transformation rules. These transformation rules apply the following:

- Transform UML classes with a component category stereotype applied, to a component type of the same category in a domain-specific modeling concept.
- Provide a component implementation for the component type according to AADL legality rules, if necessary.
- Transform UML composition links between UML classes into subcomponent relations between component implementations.
- Transform directed associations between UML classes into *in*, *out* or *in out* port features, and provide connections between these features.
- Transform dependency relations between UML classes into required bus access features in the component type, and bus access connections in the component implementation.

Functional transformation rules as stated above could be modified or extended to enrich the transformation process. We consider extensions of these transformation rules as a future direction of research.

As a proof of concept, the AADL PIM shown in Figure 3 was transformed using functional transformations. An excerpt of the resulting AADL PSM is shown in Figure 4.

4.3. Non-functional transformations

The AADL PSM acquired through functional transformations is used as an input for a number of subsequent model transformation steps. Non-functional transformations add properties to the AADL PSM allowing the analysis of the model towards exactly one non-functional property of the software system. Platform specific knowledge concerning the relation between AADL properties and analysis of the AADL model towards non-functional properties, is built into non-functional transformations. The result of a non-functional transformation is an AADL analysis model. On this model, analysis can be performed using an AADL runtime environment. To perform analysis of several non-functional properties, the AADL analysis model can be used as an input for a subsequent non-functional transformation that adds additional sets of properties to the AADL analysis model.

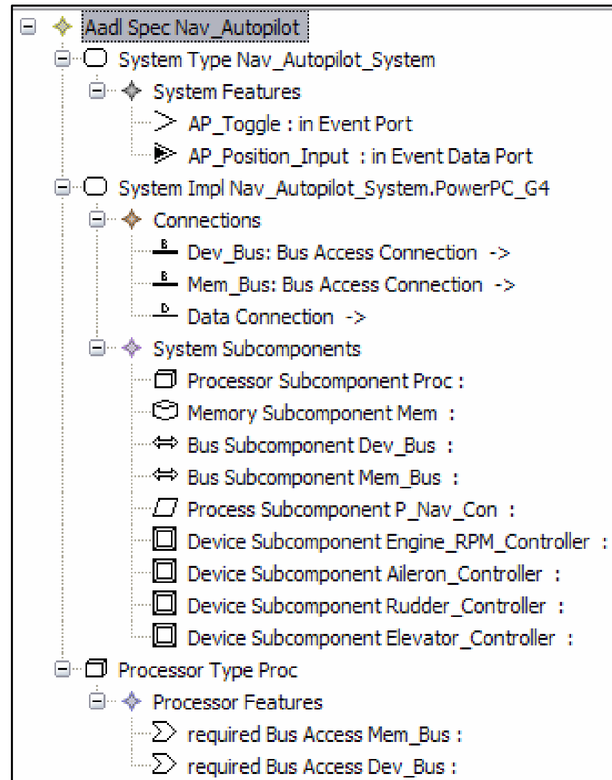

Figure 4: AADL PSM of the Autopilot system

As a proof of concept, we transformed the AADL PSM in Figure 4 through a non-functional transformation for schedulability analysis. To allow schedulability analysis of a system, a number of properties need to be present on certain components in the system. These are the following:

- There needs to be at least one thread and at least one processor component in the system.
- Processors need to specify a scheduling policy.
- Threads need to specify a dispatch policy, period, deadline and binding to a processor. If a thread is aperiodic, sporadic or background, in event ports and in event data ports need to have an incoming connection.

Therefore, the transformation rules which make up the non-functional transformation for schedulability analysis apply the following:

- If no processor component type is present in the system model, add a processor component type.
- If a processor component has no `Scheduling_Protocol` property, add this property to the processor type or implementation.
- If no thread component type is present in the system, add a thread component type.
- If a thread component misses one of `Dispatch_Protocol`, `Period`, `Deadline` or

`Actual_Processor_Binding` properties, add this property to the thread component type or implementation.
- If a thread component has the value of its property `Dispatch_Protocol` set to `aperiodic`, `sporadic` or `background`, add a connection for in event ports or in event data ports which do not have an incoming connection.

Properties added to component type or implementation through non-functional transformations, do not possess a value. Properties are added with the right name and type. Giving the newly added property a proper value, is left to the system developer.

As a proof of concept, a non-functional transformation for schedulability analysis was applied to the AADL PSM in Figure 4. The result of this transformation is an AADL analysis model for schedulability analysis. An excerpt of this model can be seen in Figure 5.

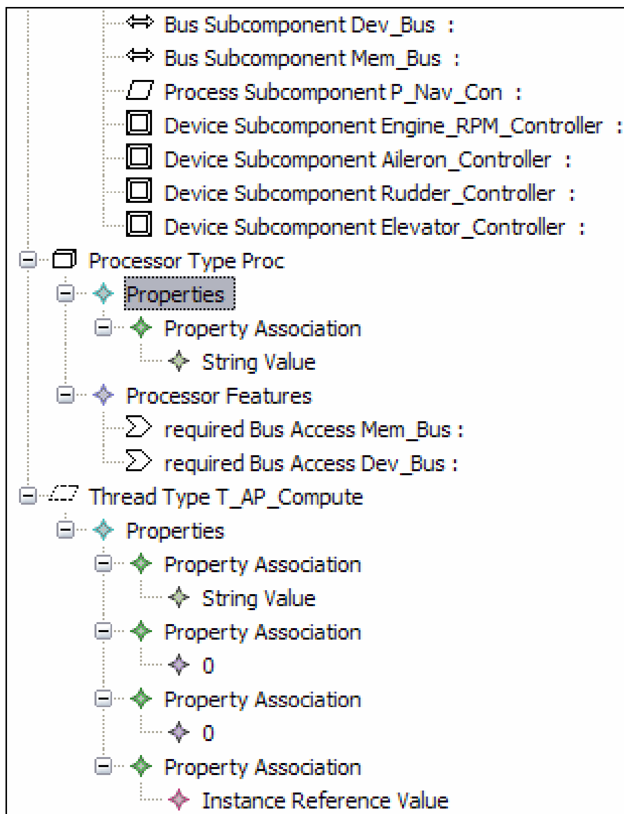

Figure 5: AADL Analysis model of the Autopilot system

4.4. Integration of the transformation process

Functional and non-functional transformations make up a logical sequence of model transformations. Functional and non-functional transformations can be integrated and chained. This chain of model transformations consists of functional and one or several non-functional transformations. Transformation chain modeling languages have been proposed as in [12]. A model transformation chain of functional and non-functional transformations make up a model transformation process targeted specifically towards model-based analysis of functional and non-functional properties, allowing an eased analysis process of both existing and newly designed software systems.

After a chain of model transformations has been applied to an AADL PIM, the AADL analysis model can be used with a tool that implements the AADL runtime environment. An example of such a tool is OSATE [11], which comes with a number of built-in analysis plug-ins. A schedulability analysis plug-in is one of the OSATE built-in plug-ins, as are semantic check, safety level and other analysis plug-ins.

5. Related work

A number of related approaches have been proposed. Dissaux [13] presents an approach to model transformation for AADL in combination with UML. In contrast to the approach we propose, Dissaux concentrates on the analysis of components from legacy code aimed specifically towards use with the HOOD Stood tool [13]. Whereas Dissaux's approach is comparable to the functional transformations we propose, our approach offers non-functional transformations as well. Also, we consider our approach more generally applicable towards use with other tools.

Bertolino and Mirandola [14] propose an approach for the specification and analysis of performance related properties of components using the RT-UML profile. Although the approach also uses a UML profile, it is not targeted towards model driven development like the approach presented in this paper.

Finally, a number of tools are available that address the issues discussed in this paper. Ocarina [15] allows model manipulation, generation of formal models, to perform scheduling analysis and generate distributed applications. Ocarina allows code generation from AADL descriptions to Ada. Cheddar [16] is a free real-time scheduling tool, providing a simulation engine and feasibility tests. Cheddar provides a number of features to ease the development of specific schedulers and task models.

6. Conclusion

Although the use of AADL as a modeling language offers a number of interesting advantages such as the prediction of non-functional properties, we have identified a number of usability issues for introducing AADL. First, software developers face an abstraction gap between system modeling concepts and software implementation concepts. Second, component composition rules in AADL are rather complex and hard to adopt. Third, the relation between the property mechanism used in AADL and prediction mechanisms used in AADL analysis tools is unclear. These issues entail an increased entry level and decreased feasibility for using AADL in software-intensive embedded system development.

To ease these issues and facilitate the use of AADL in the development of systems, we proposed a model driven development process using AADL models on three levels of abstraction. The AADL PIM of an application is at the highest level of abstraction, and does not necessarily fully comply with AADL semantic requirements. Through functional transformations, this model is transformed into an AADL PSM. Functional transformations enforce functional properties on the model, so that the AADL PSM does comply with all semantic AADL rules. Finally, using a number of non-functional transformations the AADL PSM is transformed into a full AADL analysis model. Non-functional transformations add non-functional properties to the AADL PSM, each specific for the model analysis of one specific non-functional property. Using transformation chains, functional and non-functional transformations can be chained and integrated into a model driven development process aimed specifically towards the early verification of non-functional properties of the system.

7. References

[1] Object Management Group, "Unified Modeling Language: Superstructure", OMG/2005-07-04, 2005.

[2] Systems Modeling Language open source specification project, http://www.sysml.org.

[3] Working group for the UML MARTE Profile, http://www.promarte.org.

[4] N. Medvidovic, R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Software Engineering 26, pages 70-93, 2000.

[5] SAE, "Architecture Analysis & Design Language (AADL)", AS-5506, 2004.

[6] I. Hamid, "Flight Control System", http://perso.enst.fr/%7Ehamid/aadl/fcs.pdf, ASSERT AADL Workshop, France, 2005.

[7] C. Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison Wesley, 1998.

[8] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, "Technical Concepts of Component-Based Engineering", Technical Report, CMU/SEI-2000-TR-008, 2000.

[9] Object Management Group, "MDA Guide Version 1.0.1", OMG/2003-06-01, 2003.

[10] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose, "Eclipse Modeling Framework", Addison-Wesley, 2003.

[11] SEI, "OSATE: An extensible Source AADL Tool Environment.", SEI AADL Team technical Report, 2004.

[12] B. Vanhooff, S. Van Baelen, A. Hovsepyan, W. Joosen, and Y. Berbers, "Towards a Transformation Chain Modeling Language", S. Vassiliadis, S. Wong, and T. Hämäläinen, editors, Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI), Lecture Notes in Computer Science (LNCS), Vol. 4017, pages 39-48, 2006.

[13] P. Dissaux, "AADL model transformations", Proceedings DASIA 2005 Conference in Edinburgh, UK, 2005.

[14] A. Bertolino, R. Mirandola, "Modeling and Analysis of Non-functional Properties in Component-Based Systems", Proceedings of International Workshop on Test and Analysis of Component Based Systems TACoS 2003, Electronic Notes in Theoretical Computer Science 82(6), 2003.

[15] J. Hugues, B. Zalila, and L. Pautet, "Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina", Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07), Brazil, 2007.

[16] F. Singhoff, J. Legrand, L. Nana, L. Marcé, "Cheddar: a Flexible Real Time Scheduling Framework", Proceedings of the ACM SIGAda International Conference, Atlanta, US, 2004.