Chapter Number 1

Modelling Embedded Systems with AADL: A Practical Study

Naeem Muhammad[1], Yves Vandewoude[1], Yolande Berbers[1], Sjir van Loo[2]
[1]*Departement of Computer Science, Katholieke Universiteit Leuven, Belgium*
[2]*Embedded Systems Institute, The Netherlands*

1. Introduction

In today's world, embedded systems can be seen everywhere around us. These systems range from consumer electronics such as mobile phones, cameras and portable music players to sophisticated devices such as planes and satellite systems. In either form embedded systems are designed to perform specific tasks with constraints on their qualities and available resources. These constraints can either be soft or hard depending on the nature of the system: a satellite system, for example, has hard safety constraints. Some of the major constraints for embedded systems are high reliability, performance, safety and dependability, small memory size, low power and low processing capabilities. Designing systems with such constraints is a challenge.

Developing system architectures during system development has gained importance as it helps in analyzing the system before its implementation. A system architecture is a formal description of a system that describes its building blocks, their properties and the interactions among them. System architectures can be used to analyze various properties of a system such as memory consumption and system safety. For embedded systems, this is of extreme importance since a well described system architecture allows us to predict whether any of the previously mentioned constraints can be met, without requiring the construction of an often expensive prototype implementation.

Description of system architectures can be achieved using the formal notations offered by Architecture Description Languages (ADLs). Such ADLs often also provide tool support for the modelling and analysis of the system architecture. Many ADLs for embedded systems are available in both academic and industrial communities, such as Rapide, MetaH, AADL and Wright. Among the available ADLs, the best known and most actively used language is the Architecture Analysis and Design Language (AADL). Standardized by the Society of Automotive Engineers, AADL was originally developed for modelling and analysis of systems in the domain of avionics. However, because of its rich modelling and analysis capabilities, it is widely used for embedded systems in other domains as well. AADL provides a modelling formalism accompanied by a toolset to support modelling activities and system analyses. AADL models can be used to perform various analyses such as flow latency, resource consumption, real-time schedulability, security and safety analysis. Because of its history in the avionics domain, AADL does not address each and every

modelling and analysis requirement of other embedded domains. However, during its design, it was foreseen that use of AADL in other domains could require additional modelling concepts and analyses. To meet potential needs AADL was designed as an extensible ADL.
This chapter is intended to provide insight into the design needs of embedded systems and the formalisms available to address those needs; we discuss them in section 2 and 3 respectively. We will explain the suitability of AADL and will present its architectural elements for modelling embedded systems in section 4. We will also highlight the shortcomings currently present in AADL and describe its extension mechanism by addressing one of the shortcomings with the help of an example, in section 5.

2. Embedded Systems

There is no standard, universally accepted definition of embedded systems. Some recognize embedded systems as a computer inside a product; some view them as computing systems embedded inside electronic devices; some refer them as electronic programmable devices integrated in a larger heterogeneous system. Despite the lack of a standard definition, a general consensus about embedded systems is that they are computer systems designed to perform dedicated functions. Today we can see many such systems ranging from very simple single-chip systems to complex and highly distributed embedded systems. They are present around us in various forms at various places such as digital cameras, home appliances, elevators, planes and medical devices. The role of embedded systems in human life has increased drastically over the last decade as human dependency on electronic systems has surged. In past, embedded systems were mainly used to address needs of the mission-critical systems. Recently however, they are used in devices such as mobile phones and PDAs as well.
One example of a very common embedded system is a digital camera, which is a composition of hardware and software components. The hardware is mainly responsible for capturing objects through sensors and storing images whereas the software facilitates image processing functions. Digital cameras usually have constraints such as low computation power, small size and low power consumption.
Physical constrains, such as size and weight, and cost are reasons why embedded systems, such as mobile phones or digital cameras are forced to perform their functions with considerably fewer resources than conventional systems. Other systems, such as flight-computers or transportation systems have high safety and reliability requirements. It is the presence of these non-functional constraints that makes the design and development of such systems a daunting task.
Embedded systems can be divided in to real-time and non-real time embedded systems. Real-time embedded systems have various timing constraints on their behavior, they are required to react and respond to such constraints. Correctness of their behavior depends on the ability to perform it in the given time frame or before a certain deadline. Whereas, non-real time embedded systems do not have time obligations.
Real-time embedded systems can further be categorized in hard and soft real-time embedded systems. Hard real-time systems have tight timing constrains with nearly zero-tolerance level. Failure to meet the deadlines causes the operations to stop and is considered an anomaly. Mission critical systems usually fall in hard real-time embedded systems

category. Soft real-time constrains can be found in video streaming applications for example. Failure to achieve required deadlines or throughput is annoying but does not necessarily invalidate the results. An embedded system may have both hard and soft real-time constraints for different functional and non-functional requirements.

The affordability of embedded systems is an indicator for a further increase of their use in the near future. Not only they are more widely used, embedded systems also provide more and more features and grow increasingly more complex each year. In a study, in 2008, about 30 embedded microprocessors were found per person in developed countries, with at least 2.5 million function points in the associated embedded software (Christof & Capers, 2009). The number of embedded devices in automobiles is an important growth rate indicator. Modern cars contain 20 – 70 electronic control units with up to 1Gigabyte total size of the accompanied software. A similar growth rate is predicted for embedded systems in other domains too. Embedded systems with complex specifications require sophisticated development methodologies in general and especially for modelling their architecture (Bouyssounouse & Sifakis, 2005).

Embedded Systems Specification and Modelling Needs

Embedded systems consist of both functional and non-functional properties. Although the nature of their functional properties differs between different domains, embedded systems often have similar non-functional properties known as Quality Attributes (QA). A camera for instance has different functional properties than a mobile phone but both of them are required to work with a small amount of memory and less computation power. QAs are system requirements used to describe quality aspects of the system. The quality aspects may include performance, usability, security, portability, availability, robustness and testability. There is no standard list of such QAs that define system quality, however a general criteria that distinguishes them from other requirements is that they only describe non functional aspects of system. The quality of a system is analyzed against these requirements; importantly they are used during architecture level analysis to find and fix any discrepancy to system requirements early in the development life cycle. Trudy Sherman in his research work identified QAs for embedded systems by examining eleven architectural designs of embedded systems developed by three different organizations. He produced a list of 30 QAs that are used to define various quality aspects of embedded systems (Sherman, 2007). Following are some commonly used QAs for embedded systems given in the list he produced:

Quality Attributes	Description
Reliability	The ability of a system to perform desired behavior under previously specified circumstances, and recover from undesired states if occurred.
Safety	The ability of a system to avoid potential hazards to itself, its users and the environment in which it is used.
Security	The ability of a system to resist any unauthorized usage.
Memory Usage	The capability to work with a limited amount of memory.
Performance	The degree to which a system or component accomplishes its designated functions within given constraints, such as speed and accuracy.
Usability	The system must be easy to use, operate and handle.

Table 1 Embedded System Quality Attributes

The magnitude of constraints on QAs defines the nature of a system as either a non-real time, soft or hard real-time embedded system. This magnitude must be explicitly defined and is based on the domain for which the system is developed. Furthermore, it serves as an important factor during QAs' tradeoff.

Because of the high pressure to produce embedded systems with previously mentioned characteristics in a low-cost and short time-to-market setting, embedded systems design methodologies are required to well address functional and non-functional constraints, and resolve potential issues before implementation. According to Talarico et al. they must provide support mainly to (Talarico et al., 2005)

1. Describe the interactions between the system and the external environment
2. Describe the system architecture
3. Model the behavior of hardware and software components that make up the system
4. Describe the system constraints and requirements
5. Describe the test scenarios used to simulate the system
6. Define a set of gauges to measure various performance metrics during simulation execution.

Although the criteria of Talarico et al. are valid for every software system a number of aspects are especially different for embedded systems. For example, the forth requirement support for a cost-effective method to formally specify system constrains is difficult for embedded systems, since the exact nature of all constraints is often not yet known in the early stages of development.

In addition, embedded systems are composed of software and hardware components which require that the modeling needs of both domains are well addressed. These needs include the specification of all types of software elements (such as processes, threads, communication among software elements and shared data), hardware elements (for instance processors, memory and physical communication channels), and most importantly the mapping of software elements to hardware elements. Additionally support is required for describing system dynamics. These dynamics may include system flows, system states and run-time interactions among system components.

It is also important that the support offered by the design methodology is not restricted to the modelling only but that it also facilitates system designers to analyze designs for issues and fix them early. Therefore, for modern system development methodologies the presence of design artifacts is extremely important. They not only serve as input to the next phase but are used for early system verification and validation (V&V). They can be analyzed to find potential performance deficiencies, security leaks and safety hazards. As previously discussed fixing these issues at early stage is cost effective. Any unaddressed performance deficiency or security leak that propagates to later stages of development may lead to a major refactoring and cost afterwards. With this additional perspective design techniques and tools are required to support V&V activities.
Moreover, it is important to assist design activities with rich set of toolsets to make these activities quick and correct. Although, many tools are available for modeling and analysis of functional and non-functional behaviours, some non-functional aspects remain unaddressed which are of primary focus for embedded systems. Tool support should be enhanced not only to deal with unaddressed areas but to tackle future design complexities.

3. Architecture Description Languages

Architecture Description Languages (ADLs) are modelling formalisms that provide support for describing system architectures through their formal notations. These are considered as modelling-language-plus as they can model more than conventional modelling languages. ADLs can model not only static but dynamic properties of systems as well. There is no consensus on a standard definition of ADL yet, for our discussions we will however use the one provided by Medvidovic and Taylor (Medvidovic & Taylor, 2000) "*an ADL must explicitly model components, connectors, and their configurations; furthermore, to be truly usable and useful, it must provide tool support for architecture-based development and evolution*".
The definition identifies three essential requirements an AADL must fulfil. Firstly, it should provide support for modelling a static structure in the form of components and connectors. Secondly, ADLs should provide support for modelling configurations of components and connectors, which usually define system's dynamic behaviour. The third pivotal requirement for an ADL is to provide tool support to assist modelling and analysis activities. There are several ADLs available to date; some of them are given in the following table.

ADL	Application Domain
AADL	Is used to model real time embedded systems particularly in the avionics domain.
Acme	Interchanges architecture description information between ADLs.
Aesop	Is used to model style-specific architecture descriptions, also provides support for designing custom architecture styles.
ArchC	Is a SystemC based language used to describe hardware elements.
ArchJava	Checks conformance of an architecture of a software system to its implementation, and keeps architecture and code consistent during their evolution.
ControlH	Is used to develop the architecture specification and code generation for control and navigation systems.
C2	Supports architecture description of highly-distributed, evolvable, and dynamic systems.
Darwin	Is used to describe architectures of dynamically changing highly-distributed systems.
EAST-ADL	Addresses modelling and analysis needs of automotive electronic systems.
MetaH	Supports modelling of real-time systems in the domain of guidance, navigation and control.
Modechart	Is used to describe architecture for hard real-time embedded systems.
Rapide	Provides support for developing event based simulations for distributed event-extensive systems.
SADL	Is designed to simulate real-time properties for hard real-time systems from the avionics domain.
Weaves	Is used for describing architecture for data-flow-extensive systems with real-time processing on a high volume of data.
Wright	Is used for describing communication behaviour of concurrent systems.

Table 2 Architecture Description Languages

The ADLs given in the table clearly show that most of them are domain specific description languages addressing needs of systems for particular domains. They vary widely in their supported abstractions and analysis capabilities. Their incapability to be applicable for every domain is resulting in new ADLs. With a large number of ADLs available and most of them domain specific, and considering the fact that a single ADL does not address every modelling requirement particularly in the case of multidisciplinary systems, it is difficult to choose the right language. Some work however is being done in establishing a contact point among ADLs where multiple languages may be used together. Acme is serving as such a contact point, it is an ADL whose core purpose is to support the mapping of an architecture description of one ADL to another (Medvidovic & Taylor, 2000).

The role of architecture for early system validation certainly is of great importance. However, the effectiveness of the validation activities can only be increased by a formal description of the architecture and assisting the activities with tool support. ADLs with their support for formal description and accompanied toolset for architecture modelling and analysis can serve for this purpose.
The QAs of an embedded system are difficult to model and analyze because they are associated with a system's dynamic behavior that is only available during the last phases of its development. It is nevertheless cost effective to resolve issues related to them as early as possible. ADLs' support for modelling dynamic behavior and tools for analyzing architecture for quality attributes can be utilized for embedded systems.
Although a large number of ADLs is available, most of those are present only in the research community, and are not applied in industry. In addition, some of them are no longer in use. In a survey, which we conducted to find a suitable ADL for modelling performance for an electron microscope embedded system we found that most of the candidate languages are no longer active. Among MetaH, Rapide, AADL and Wright, only AADL is active and being used in industry. In addition, the amount of literature published between 2000 and 2008 for AADL magnitudes greater than for others, suggesting a wide acceptance in the research community as well.
Besides its wide acceptance, AADL distinguishes itself from other ADLs by its extensible nature. Although initially designed for the avionics industry it can be applied to other domains by extending its core concepts where required (Feiler et al., 2006). This will help in generalizing an ADL for various domains, previously an issue with many ADLs.
In the following section we will discuss AADL in detail; we will highlight its modelling and analysis capabilities and discuss its toolset. Moreover, in section 5 we will explain how AADL can be extended to meet custom modelling needs using an example.

4. Architecture Analysis and Design Language (AADL)

In compliance to the definition of ADL, AADL provides a modelling formalism accompanied by a toolset to support modelling activities and analysis. Originally developed for modelling and analysis of systems in the domain of avionics, it has been standardized by the Society of Automotive Engineers. Because of its rich modelling and analysis capabilities it is widely used for embedded systems in other domains as well, especially suitable for model-based analysis and specification of complex real-time embed systems (Feiler et al., 2006). In this section a brief introduction is given on the AADL architectural notations, its analysis and its tool support.

4.1 Modelling

AADL consists of a rich set of architectural elements for modelling components, their interactions and their configurations. Architectural elements and the core concepts of the language are given in figure 1.
Components in AADL are used in terms of component types and component implementations. A component type defines the externally visible characteristics of a component usually by using features, flows and properties. Whereas, a component implementation models the internal structure of the component. An internal structure may consist of subcomponents, connection among them, flows across them and their operational

behaviour. AADL distinguishes between three types of components: software components, hardware components and composite (system) components:

Software Components:
Thread: Represents a unit of sequential execution through source code.
Thread group: Represents a logical grouping of threads.
Process: Represents a protected address space.
Data: Represents static data and data types.
Subprogram: Represents a callable part of a source code.

Hardware Components:
Processor: Hardware that is responsible for executing threads.
Device: Hardware that interacts with the external environment.
Bus: Hardware that provides access to the other execution platform components.
Memory: Hardware that stores digital data.

Composite (System) Component: A component composed of software and hardware or even system components.

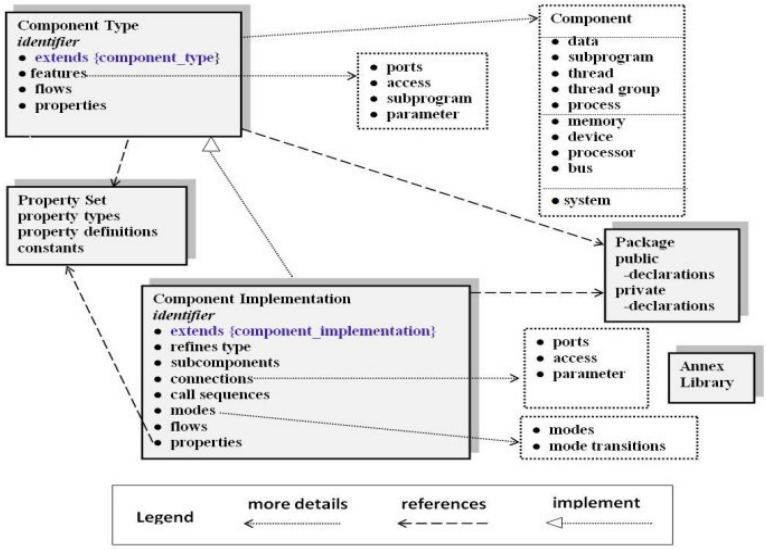

Figure 1 AADL Architectural Elements (Feiler et al., 2006)

Interactions between components can be realized by using *features* and *connections*. *Features* are interaction points of components, which are used for communication. *Features* are classified into *ports*, *component access*, *subprogram calls* and *parameters*. A *port* is a communication interface of components used to exchange data and events. AADL categorizes *ports* as *data ports*, *event ports* and *event data ports*. Moreover, multiple ports can be grouped together in a *port group*. *Component access* enables components to access shared data and bus. For access, components are required to explicitly use *provides access* and *requires access* declarations. *Subprogram calls* are used for synchronous calls to subprogram components, and *parameters* are used to represent data values passing in and out of a

subprogram. Connectors are used to connect interaction points of components. AADL provides *data*, *event*, *eventdata*, *dataaccess*, *bussaccess* and *portgroup* connectors.

4.2 Analysis and Tool Support

In its conformity to the ADL definition, AADL provides support for various kinds of analyses along with conventional modelling. A few of the supported analysis are:

Flow Latency Analysis
Understand the amount of time consumed for information flows within a system, particularly the end-to-end time consumed from a starting point to a destination.

Resource Consumption Analysis
Allows system architects to perform resource allocation for processors, memory, and network bandwidth and analyze the requirements against the available resources.

Real-Time Schedulability Analysis
AADL models bind software elements such as threads to hardware elements like processors. Schedulability analysis helps in examining such bindings and scheduling policies.

Safety Analysis
Checks the safety criticality level of system components and highlights potential safety hazards that may occur because of communication among components with different safety levels.

Security Analysis
Like safety levels, AADL components can be assigned various security levels. The analysis helps in identifying the security loopholes that may happen because of mismatches in security levels between a component and its subcomponents, and communication among components with different security levels.

Various tools are available to perform these analyses:
OSATE (Open Source AADL Tool Environment) developed by SEI is a set of Eclipse plug-ins for front-end processing and various analyses of AADL models (Feiler & Hansson, 2007).
ADeS (Architecture Description Simulation) by Axlog simulates various aspects of the system behaviour, such as scheduling of processes and threads by processors (AXLOG, 2009).
Cheddar (Singhoff & Plantec, 2007), developed by LISyC Team, is a real-time scheduling tool which provides support for quick prototyping of real-time schedulers and schedulability analysis. Like ADeS, Cheddar also supports simulation of scheduling properties of a system.
ANDES (ANalysis-based DEsign tool for wireless Sensor networks) (Prasad et al., 2007) was developed for modelling and analysis of wireless sensor network systems. It provides support for real-time communication schedulability, target tracking and real-time capacity analyses.

5. Extension to AADL

Initially designed as a language for modelling avionic systems, AADL includes core modelling concepts and certain analyses essential for real-time systems in the aerospace domain. However, during its design, it was foreseen that use of AADL in other domains could require additional modelling concepts and analyses. To meet potential needs AADL was designed as an extensible ADL.

It is possible to extend the AADL concepts either by introducing new properties to the modelling elements, by addition of new modelling notations, or by developing a sublanguage as annex to the AADL standard (Frana et al., 2007). The latter technique is mainly used for large-scale extensions and was considered out of scope for our own purpose. Since, for our research work we extended AADL by using the property based extension technique, the scope of the example we present here will be restricted to this technique only. In this ecample, we will extend AADL End-to-End (EtE) flows to provide support for Composite EtE flow modelling and latency analysis.

EtE flow latency is the amount of time consumed by the contributing components for a specific flow of information from a source to a destination. Currently, AADL requires that flow specifications of the contributing components are connected through the AADL connector element and does not provide support for composite EtE flows: flows that themselves consist of multiple discrete EtE flows. We will describe how this issue can be overcome with an extension to AADL, by introducing a new property for the AADL EtE flow element. Latter we will discuss results of the extended EtE flow analysis with the help of an analysis tool that we developed for this purpose.

We will apply this technique on an electron microscope embedded system. An electron microscope is a sophisticated microscope used to examine minute specimens by creating highly-magnified images.

5.1 AADL EtE Flow Extension

Flows in AADL describe the different sequences of an information flow through a set of contributing components. The description of this flow is subsequently used in certain analyses such as a flow latency analysis. In AADL, flows are defined with a flow specification and a flow implementation. A flow specification represents the externally visible flow of information in a component; it is specified within the component type declarations using flow sources, flow paths and flow sinks (Feiler & Hansson,2007). A flow source represents the originator of the flow, the flow sink represents the end consumer of the flow information, and the flow path embodies the link between incoming and outgoing ports involved in the flow. A flow implementation on the other hand represents the actual realization of a flow within a component; it is specified within the component implementation declarations.

```
system motion_client
        flows
        start_motion_flow: flow source C_start_motion;
        move_status_flow: flow sink C_motion_status;
end motion_client;

system motion_server
        flows
        start_motion_flow: flow sink S_start_motion;
        axis_move_flow: flow source S_move_axis;
        move_status_flow: flow path S_move_status ->
                                    S_motion_status;
    end motion_server;


device motion_controller
        flows
        axis_move_flow: flow sink Ctr_move_axis;
        move_status_flow: flow source Ctr_move_status;
end motion_controller;
```

Figure 2 AADL Flow Specification

An example of both a flow specification and a flow implementation is given in Figures 2 and 4. The excerpts are taken from the electron microscope's motion-subsystem that is responsible for moving the specimen holder. The system consists of three major components working in a client-server environment: the motion client, the motion server and the motion controller. The motion server receives its stage movement commands from a client application, processes it and moves the motion controller to the desired position. The externally visible flow of the move command is shown in Figure 3, which corresponds with the textual AADL representation in Figure 2.

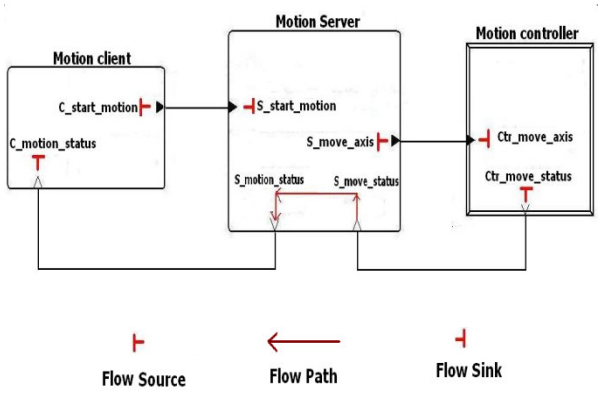

Figure 3 Motion-Subsystem Move Command Flow

```
system implementation motion_server.server_app

--following flow receives information at a sink port of the server and passes it to a flow specification
--of the MdlMotion subcomponent of the server through a connector.
start_motion_flow: flow sink S_start_motion -> ServertoMdlConnector
                    -> MdlMotion.start_motion_flow;

--following flow connects a flow specification of the MdlMotion subcomponent to a flow specification
--of the HalMotion subcomponent through a connector and continues through another connector to
--an out port of the Server.
axis_move_flow: flow source MdlMotion.axis_move_flow -> MdltoHalConnector
                  -> HalMotion.axis_move_flow -> HaltoServerConnector- > S_move_axis;

--following flow passes information received from an incoming port of the server to the HalMotion
--subcomponent through a connector, subsequently it connects a flow specification of the HalMotion
--to a flow specification of the MdlMotion by using another connector. Finally, it connects a
--MdlMotion flow specification to an out port of the server.
move_status_flow: flow path S_move_status -> ServertoHalConnector
                    -> HalMotion.move_status_flow  -> HaltoMdlConnector
                    -> MdlMotion.move_status_flow ->MdltoServerConnector
                    ->S_motion_status;

end motion_server.server_app;
```

Figure 4 AADL Flow Implementation

An EtE flow latency analysis requires the specifications of EtE flows. An EtE flow represents a logical flow of information from a source to destination passing through various system components. It is defined in the component implementation (typically in the top level component in the system hierarchy) and refers to the specifications of other flows in the system.
An EtE flow specification consists of the flow specifications of the contributing components connected through the AADL connector, Figure 5 contains the standard syntax for EtE flow specification.

```
end_to_end_flow_spec::=defining_end_to_flow_identifier:end to end flow
    start_subcomponent_flow_identifier
   { -> connection_identifier ->
     flow_path_subcomponent_flow_identifier
    }*-> connection_identifier-> end_subcomponent_flow_identifier
    [{(property_association)+}][in_modes_and_transitions];
```

Figure 5 Standard AADL EtE Flow Syntax

According to the standard specification, an EtE flow starts with a flow specification of the starting subcomponent, connects it to a flow specification of the subsequent subcomponent and so on, and finally connects to a flow specification of the last component in a flow. A notable point in the EtE flow modelling is that each contributing flow specification is connected to its adjacent flow specifications. The starting and ending flows are connected to one flow each, all intermediate flows are connected to both the predecessor and successor

flows. This concept restricts EtE flow analysis only to those flow specifications that are linked. AADL does not provide support for more abstract flows whose internal flows are not linked. For clarity, we distinguish in our discussion between discrete EtE flows (in which all sub-flows are connected) and composite EtE flows (that consists of unlinked sub-flows).
In case of our motion-subsystem example, *Move_Stage* is a composite EtE flow: the flow starts when a client application sends a move request to the server and ends when client gets acknowledgement that stage has been moved to a desire position. Internally, *Move_Stage* consists of three consecutive discrete (disconnected) EtE flows *Start_motion_flow*, *Axis_move_flow* and *Move_status_flow*.

Start_motion_flow:
The flow starts with a move request from the client application and ends with the server. Internally, the server simply places the command in a queue, therefore this flow ends here. A part of the AADL textual representation given in Figure 2 specifies this behaviour of the flow.

Axis_move_flow:
As soon as a move command is available in the queue, a component of the motion server processes it and generates a new instruction for the motion controller. Upon receipt of the instruction, the motion controller moves the stage to the desired position. The server component involved in this task is different from the component involved in previous EtE flow. More importantly, both are not connected with each other for this particular task. Since the flow specifications of both components are not linked with each other, according to the AADL specification they can not be a part of a single EtE flow. Hence, this results in another EtE flow starting from the motion server and ending with the motion controller. The specification of this flow can be seen in Figure 2.

Move_status_flow:
As soon as the stage is moved to its position, the motion controller sends a motion completion acknowledgement back to the server which subsequently dispatches it to the client application. Internally, two different subcomponents of the motion controller are responsible for stage movement and acknowledgement generation. As such, the flow specifications of both components are not connected with each other. Therefore, sending acknowledgement back to the server is the start of a separate EtE flow with the motion controller as its starting point and the client as its ending point. The specification of this flow is shown in Figure 2 as well.

AADL's incapability to model composite EtE flows exists at any level of abstraction in AADL models, although chances of having such flow specifications increase with higher level of abstraction. Therefore, need for modelling and analysis of such EtE flows is significant at higher abstraction (system architecture) levels. Providing modelling support for composite EtE flows will also enhance flow latency analysis. The capabilities will enable system architects to analyse system flows at higher abstraction.
The incapability, as we described earlier, exists because AADL does not provide any support for linking disconnected flow specifications. Bridging such flow specifications can

enable modelling and analysis of composite EtE flows. We will introduce a new property that will serve as a bridge between disconnected flow specifications.

Property Based Extenstion
An AADL property provides descriptive information about components, subcomponents, features, connections, modes, subprogram calls and flows (Feiler et al., 2006). A property consists of an associated value and type; the AADL standard consists of a set of predefined properties. However, new properties can be introduced in order to add additional information about the above mentioned architectural elements. The standard properties for EtE flows are:
Expected_Latency: Time
Actual_Latency: Time
Expected_Throughput: Data_Volume
Actual_Throughput: Data_Volume

We introduce a new property called *Link_Flow* to the existing properties. The new property holds a string value representing the identifier of the EtE flow that is to be linked. As new properties are defined in the AADL property sets, we declare the new property in the FLOWCONN property set. The declaration of the property is:

property set FLOWCONN is

Link_Flow: aadlstring applies to (flow);

end FLOWCONN;

Afterwards, the *Link_Flow* property can be used in the EtE flow declaration, in which it is assigned the identifier of the EtE flow to be linked.

While linking discrete EtE flows by using a property we assume that the delay between adjacent EtE flows is zero. Although this is the case in the motion subsystem (information is passed on through shared memory), many scenarios can be thought of in which this is not the case (such as the presence of a queue). Although not addressed in depth here, the solution remains applicable for those cases as well by including the delay in the model as a connector-like construct and applying a statistical model to them. The resulting latency of the EtE flow will also be a statistical distribution.
The introduction of properties allows us to attach additional information to the different elements of the AADL model. Subsequently this information can be inspected and/or manipulated by the accompanying tools that carry out the EtE flow latency analysis.

5.2 Applying Our Solution

In the previous section we defined a new property *Link_Flow* to connect discrete EtE flows. Now, we apply the proposed solution on the motion-subsystem, by using the property to connect internal discrete EtE flows of the *Move_Stage* flow. The use of our property within the standard AADL syntax is shown in Figure 6.

```
end_to_end_flow_spec::=defining_end_to_flow_identifier:end to end flow
    start_subcomponent_flow_identifier
      { -> connection_identifier ->
          flow_path_subcomponent_flow_identifier
       }* -> connection_identifier -> end_subcomponent_flow_identifier
[{(FLOWCON::Link_Flow=>"identifier_of_subsequent_EtE_flow") |
  (property_association)+}][in_modes_and_transitions];
```

Figure 6 EtE Flow Syntax with Link_Flow Property

As stated earlier the *Move_Stage* EtE flow is composed of *Start_motion_flow*, *Axis_move_flow* and *Move_status_flow* in sequence. By using the *Link_Flow* property we will link *Start_motion_flow* to *Axis_move_flow*, and *Axis_move_flow* to *Move_status_flow*. An AADL textual description of the linkage is given in subfigures 7(a) and 7(b).

```
Start_motion_flow:end to end flow
MotionClient.move_request_flow ->
  ClienttoServerMotionConnection ->
  MotionServer.move_request_flow
{ FLOWCONN::Link_Flow =>"Axis_move_flow";};
                    (a)

Axis_move_flow: end to end flow
  MotionServer.axis_request_flow ->
  ServertoControllerAxisConnection ->
  MotionController.axis_request_flow
{FLOWCONN::Link_Flow =>"Move_status_flow";};
                    (b)

Move_status_flow: end to end flow
 MotionController.status_flow ->
 ControllertoServerStatusConnection ->
 MotionServer.status_flow ->
 ServertoClientStatusConnection ->
 MotionClient.status_flow;
                    (c)
```

Figure 7 Link_Flow Property for *Move_Stage* Composite EtE flow

Since *Move_status_flow* is the last EtE flow in the composition it is not linked further, as shown in subfigure 7(c).

Using our technique, any number of EtE flows in a composition can be linked. Afterwards, the value of the defined property can be used during analysis to calculate the total flow latency of the *Move_stage* EtE flow. The calculation starts with the latency of the first flow,

subsequently adding the latency of the EtE flow given in the *Link_Flow* property value. The addition continues until an EtE flow without *Link_Flow* property (or with empty value) is found.

We developed an OSATE-plugin that analyzes AADL models for composite EtE flows. OSATE is built on top of the Eclipse platform and is very well suited as a basis for the development of tools that operate on AADL models. OSATE's extensible plug-in architecture provides a wide range of methods and services generated from the AADL meta-model that can be used by plug-ins to manipulate AADL models.

By using the proposed property *Link_Flow* the developed plug-in differentiates composite flows from distinct flows, counts them, identifies their compositions (list of the contributing discrete EtE flows) and calculates their latencies (total time consumed by the composite EtE flow).The results of the analysis on the motion-subsystem is shown in Figure 8.

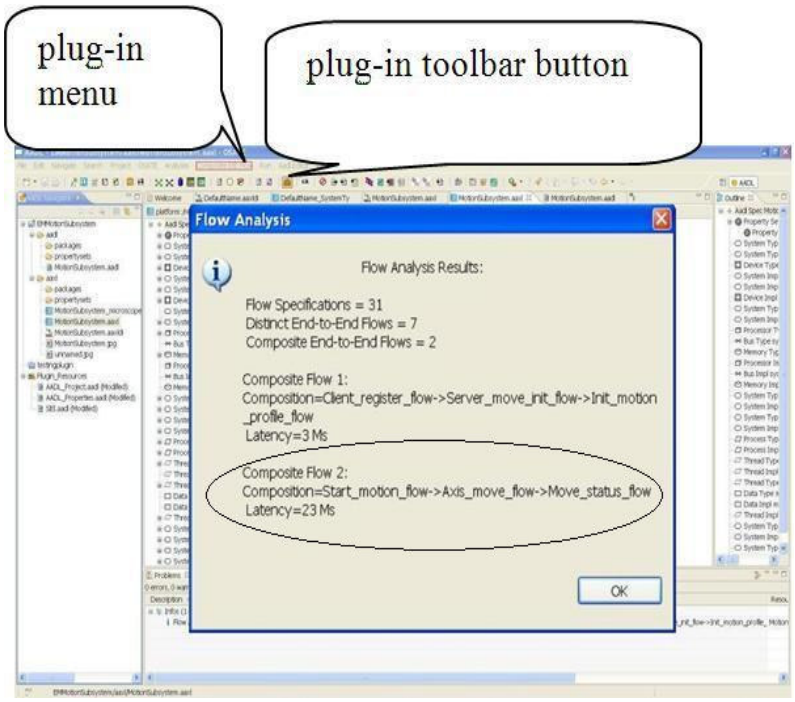

Figure 8 Composite End-to-End Flow Analysis

The analysis shows that the motions-subsystem contains 31 flow specifications, 7 discrete and 2 composite EtE flows. The *Move_stage* composite EtE flow that we are discussing in our example is among the two identified by the tool. The encircled part of the Figure contains its composition and latency.

6. Summary

In this chapter we highlighted the design needs for embedded systems and discussed the formalisms available to address those needs. We found that development of embedded systems is more complex than that of general IT systems because of the constraints associated to them. In spite of the development complexity their use is increasing significantly and their rapid growth imposes challenges to the current development methodologies. Sophisticated and formal approaches are required to tackle their growing complexity throughout their development life cycle in general and for their design in particular.

We also discussed how the magnitude of constraints on requirements of embedded systems categorizes them in non-real time, hard and soft real-time embedded systems.

Quality attributes are the driving force for embedded system design as they have great impact on design decisions. During our discussions we identified some quality attributes specific for embedded systems and highlighted the needs for modelling those attributes.

Architecture description languages with their formal notations and analysis capabilities can reduce design complexities for embedded systems. In this chapter we gave a brief overview of some of the existing ADLs with a brief description of their scope. Although a significant number of ADLs exists, many of those only operate in an academic setting or are no longer actively used and maintained. AADL is one of the best known and most actively used architecture description language for embedded systems. With a rich set of architectural elements for describing both software and hardware elements of embedded systems it is quite useful for designing their soft and hard real-time properties. Furthermore AADL provides support for analyzing and fixing those properties. Since most ADLs are domain specific, no single ADL can serve for all domains. AADL, although designed for the avionics domain, is an extensible language that makes it possible to enhance its applicability to other domains. In this chapter we presented an introduction to AADL and showed with an example how it can be extended for custom needs. We extended its EtE flows to include support for modelling and analysis of composite flows. We applied the proposed solution to an industrial case of the motion-control subsystem of an electron microscope. The application included modelling a composite EtE flow of the motion-subsystem, which is composed of three discrete EtE flows. With the help of a new AADL property we successfully linked the discrete EtE flows to model them as a single abstract EtE flow. In addition, we presented the results of the extended EtE flow analysis which we performed with the help of an analysis tool (an OSATE plug-in) that we developed for this purpose.

7. References

Bouyssounouse, B. and Sifakis, J. 2005 *Embedded Systems Design: the ARTIST Roadmap for Research and Development (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc.

Christof Ebert, Capers Jones, "Embedded Software: Facts, Figures, and Future," Computer, vol. 42, no. 4, pp. 42-52, Apr. 2009, doi:10.1109/MC.2009.118

Feiler, P.H.; J. Hansson, "Flow Latency Analysis with the Architecture Analysis and Design Language (AADL)," Technical Note CMU/SEI-2007-TN-010, Software Engineering Institute, 2007

Feiler, P.H.; Lewis, B.A.; Vestal, S., "The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems," *Computer-Aided Control Systems Design, 2006 IEEE International Symposium on*, pp.1206-1211, 4-6 Oct. 2006

Feiler, P.H., Gluch, D.P., and Hudak, J.J., "The architecture analysis & design language (AADL): An introduction," Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, 2006

Frana, R.B.; Bodeveix, J.-P.; Filali, M.; Rolland, J.-F., "The AADL behaviour annex -- experiments and roadmap," *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pp.377-382, 11-14 July 2007

Medvidovic, N.; Taylor, R.N., "A classification and comparison framework for software architecture description languages," *Software Engineering, IEEE Transactions on* , vol.26, no.1, pp.70-93, Jan 2000

Prasad, V.; Ting Yan; Jayachandran, P.; Zengzhong Li; Son, S.H.; Stankovic, J.A.; Hansson, J.; Abdelzaher, T., "ANDES: An ANalysis-Based DEsign Tool for Wireless Sensor Networks," *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International* , pp.203-213, 3-6 Dec. 2007

Singhoff, F. and Plantec, A., "AADL modeling and analysis of hierarchical schedulers," In Proceedings of the 2007 ACM international Conference on Sigada Annual international Conference, pp. 41-50, 4- 8 Nov. 2007

Sherman, Trudy: *Quality Attributes for Embedded Systems*. Springer (2007), S. 536-539.

Talarico, C.; Aseem Gupta; Peter, E.; Rozenblit, J.W., "Embedded system engineering using C/C++ based design methodologies," *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the* , vol., no., pp. 81-88, 4-7 April 2005

AXLOG. ADeS: A Simulator for AADL http://www.axlog.fr/aadl/ades_en.html, 2009

Acknowledgment

This work has been carried out as a part of the Condor project (http://www.esi.nl/ Projects->Condor) at FEI company under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program