

Using AADL to Model a Protocol Stack

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

In recent trends, the Architecture Analysis and Design Language (AADL) has received increasing attention from safety-critical software development industries. Specific about the AADL is its strong syntactic and semantic support for the description of both hardware and software architectures. Considering the existing range of software architectures, we study the support AADL offers for the description of software architecture. As a case study we use an implementation of a UDP/IP protocol stack. Based on our experiences, our position is that a number of abstract concepts, e.g. a generic component concept, are missing in the AADL to make it well-suited for the high-level description of software architecture.¹

1. Introduction

In our environment, it is not difficult to note how embedded systems are omnipresent. Embedded systems are used for a wide range of applications and have to address different requirements. In a typical development process for embedded systems, the design of a software architecture to meet these requirements is based on good practice, experience and estimates. After the software architecture has been implemented, the relation between original requirements and different parts of the implemented architecture is often unclear. This situation is particularly problematic in the case of safety-critical embedded systems, which have the additional requirement to be dependable, safe, reliable and available.

Architecture Description Languages (ADLs) [3] provide a means to describe architecture at high level. Abstracting away from lower-level details, an ADL allows to describe an architecture as a collection of interacting components. An ADL provides a way to formalize detailed requirements through the addition of properties, and reason about architectural design choices. In this way, the use of an appropriate ADL has the potential of decreasing the complexity of developing embedded

systems. With the increasing complexity of embedded systems, the potential benefit of using an ADL as part of the development process grows.

Recently, safety-critical system development industries have turned towards the Architecture Analysis and Design Language (AADL) [1] as a concrete example of an ADL. AADL has grown out of MetaH [2], an ADL specific to the avionics domain. Noticeable about the AADL is its strong syntactic and semantic support for architectures consisting of components of a limited number of functional categories. Along with this it allows to add non-functional properties to architectural components, such as timing, memory consumption and safety properties. In this way, the model of a system architecture allows specific tools to predict non-functional properties of the system in early design phases, which makes AADL a particularly interesting notation for embedded software development.

In this paper we examine the syntactic and semantic support that AADL offers to modeling software architecture. We concentrate on the structural properties of software architecture, described in terms of components, interfaces and connections, and behavioral properties. To assess this support, we use the AADL to model the layered software architecture of a UDP/IP protocol stack implementation.

The remainder of this paper is organized as follows. In section 2, we introduce the AADL. In section 3, we provide an overview of the protocol stack used as a case study. In section 4, we explain how the architecture of the implementation of this protocol stack can be modeled. In section 5, we discuss our modeling experiences. Finally, in section 6 we draw conclusions.

2. Introduction to AADL

Architecture description languages aim to describe architectures in an abstract but precise way. Numerous ADLs [3] have been developed that each provide complementary capabilities for architectural development and analysis. ACME [4] is an architecture description interchange language, introduced for the specific purpose of providing an interchange format for architectural descriptions developed in different ADLs. For this purpose it defines an architectural ontology, consisting of seven basic architectural design elements. To clarify the

¹ 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).

interpretation of the component-based paradigm used in AADL, we discuss how each of the elements in this ontology is used in AADL. For a more profound comparison of ADLs, we refer to Medvidovic [3].

The design elements identified in ACME [4] are the following.

- A *component* in ACME represents a computational element or data store of a system. AADL describes an architecture in terms of components, which are subdivided into ten functional component categories, each supported by specific semantics. There is no notion of a component with generic functionality in AADL.
- A *connector* in ACME represents an interaction among components. AADL provides several notions for modeling interactions. Components can have a connection, binding or call each other.
- A *system* in ACME represents a configuration of components and connectors. In AADL a system is represented by a specific component category. Systems can have subcomponents of other categories.
- A *port* in ACME represents a point of interaction between a component and its environment. AADL also provides the concept of a port.
- A *role* in ACME represents a participant of the interaction represented by a connector. Connections in AADL have two dedicated semantic roles, which are source and destination.
- A *representation* in ACME represents a detailed, lower-level description of a component. AADL provides the complementary concepts of component type and implementation, in which an implementation is the representation of a component type.
- A *rep-map* in ACME represents a correspondence between internal system representation and external interface of a component or connector. In AADL a rep-map is implicitly defined when instantiating components and choosing an implementation.

What noticeably distinguishes AADL from other ADLs, is its semantic support for component functionality by defining component categories. These categories can further be subdivided as belonging to one of three types of architectures. In the following list, we give an overview of the types of architectures AADL supports, and the component categories that are specific to these architectures.

- Software architecture
 - Process
 - Thread
 - Thread group
 - Subprogram
 - Data
- Hardware architecture
 - Processor
 - Memory
 - Device
 - Bus
- System architecture
 - System

In this list, system architecture is defined as the architecture of a system consisting of both hardware and software components. The AADL standard [1] defines support for functional categories through the definition of a number of semantic constraints. These constraints define restrictions on the composition of components of given categories, the annotation of non-functional properties to components of a given category, the semantic interpretation to values of these properties, etc. As an example, the AADL standard restricts the nesting of threads to processes and the nesting of processes to systems to obtain a correct architecture of software components. Component declarations can also be logically grouped in packages. AADL provides no semantic support for packages, i.e., it is not possible to declare input and output ports to packages.

The complete model of a software, hardware or system architecture in combination with specific analysis tools, enables the prediction of non-functional properties on the architecture. Strong semantic constraints are needed to allow these architectural analyses to be useful. As an example, consider the case where we want to execute a schedulability analysis on a system architecture. For schedulability analysis, a processor component and a number of thread components need to be present in the architectural model, and must have properties specific to schedulability, e.g., dispatch policy or priority. In AADL, these semantic constraints are defined using component categories.

Besides structural aspects of system architecture, the AADL allows to specify behavior of threads and subprograms. To this end, AADL provides three mechanisms. First is the use of modes for the definition of state-dependent architectural configurations. Second is the specification of the behavior of threads through sequential subprogram calls. Third is the specification of the behavior of subprograms through the AADL Behavior Annex [5]. While building our model of a protocol stack, we used only subprogram calls to specify behavior.

3. A Protocol Stack

To assess the software architecture modeling capabilities of AADL, we consider the example of a protocol stack implementation. We use this example because of its theoretical background [6] and its frequent occurrence in industrial settings [7]. A protocol stack follows a layered architecture. A layer [6] in this architecture is defined as a collection of services offered to higher layers, using services from lower layers. A service is a set of operations implemented using protocols, in which the latter define format and semantics of messages that are specific to one layer.

The protocol stack implementation we model in our example is the implementation of a UDP/IP protocol stack. Figure 1 displays a box-and-line diagram of the implemented protocol stack.

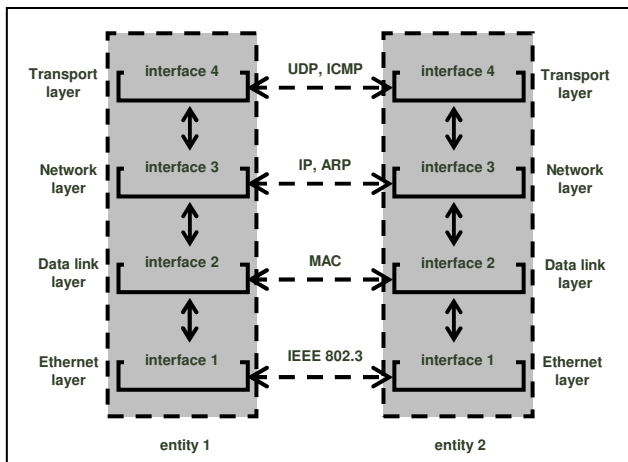


Figure 1: UDP/IP protocol stack.

In figure 1, the implementation is shown on two network entities. Each entity implements the protocol stack as a layered software architecture. A dashed line between the layers of two entities represents virtual communication [6] i.e. direct communication between two layers using a layer specific protocol. Full lines represent communication between layers, using the interfaces of these layers, which offer respective services. At the highest level we find the transport layer, which uses UDP [8] to communicate using sessions. At the lowest level we find the Ethernet layer, which uses the IEEE 802.3 [9] protocol. In between we find the network and data link layers.

The software architecture of an embedded system that implements these protocols through services, is the layered software architecture we want to model in AADL.

4. Modeling a Protocol Stack

To model the protocol stack architecture of section 3 in AADL, we consider what components the architecture consists of. Taking into account that each layer has a specific function and interface, and uses this interface to communicate to higher and lower layers, it seems reasonable to model each layer as a separate component. Unfortunately AADL does not provide a component category to model the functionality of a layer. Instead, we model each layer as a package of subprogram and data component declarations. Subprograms in this package represent the interface of the layer, data components represent formats defined in the protocols on that layer. Subprograms model the services each layer offers, i.e., allow to open sockets to network entities, send and receive packets from network entities etc. In a similar way, a package of subprograms is used to represent concepts used on several layers, such as a socket, a FIFO queue or a cache of addresses, since there is no component category for such concepts available in AADL with semantics that apply to this case. As an example, a cache could be modeled as a memory component but semantic constraints in AADL do not allow us to use it as a subcomponent of a process or a thread. A subcomponent in AADL is a component that is nested in another component.

```

package Transport_Layer
public
  -- data types
  data PDU_Length
  properties
    Language_Support::Data_Format=> Positive;
  end PDU_Length;

  data Receive_Fifo
  properties
    Language_Support::Data_Format=> Record;
  end Receive_Fifo;

  data implementation Receive_Fifo.impl
  subcomponents
    length: data PDU_Length;
    socket: data Network_Entity;
  end Receive_Fifo.impl;
  -- ...

  -- subprograms
  subprogram Init
  end Init;

  subprogram Bind
  features
    socket: in parameter Network_Entity;
  end Bind;
  -- ...
end Transport_Layer;

```

Figure 2: AADL model of transport layer.

In figure 2, the model we built of the transport layer is shown, leaving out a number of details for clarity. In the upper half of figure 2 we find data types that represent message length and a FIFO queue for receiving frames. In the implementation of the latter we see that it has data subcomponents, that represent length of the queue and the socket of which the frames were received from. In the lower half we find subprograms that represent an initialization routine and a routine to open a UDP connection to a given network address.

Because we started from an existing implementation in which there was only one thread, we modeled all subprograms as being executed in one thread, which is in turn part of one process. As a result, the model we obtain of the UDP/IP protocol stack implementation consists of one process, one thread, a number of data components and a large number of subprograms. Inside this thread, subprograms call each other to use services from lower layers and offer services to higher layers.

Along with the structural properties, we modeled behavioral properties of the protocol stack implementation. The behavior of the single thread is modeled by means of subprogram calls. In figure 3 we provide a model of the behavior of this thread. What we found missing in this mechanism, is the possibility to model conditional subprogram calls i.e. if-structures.

```

thread Application
  features
    address : in data port Network_Entity;
  properties
    Period => 5 ms;
    Compute_Execution_Time => 2 ms .. 3 ms;
    Language_Support::Priority => 100;
    SEI::RAMBudget => 2000.0 B;
end Application;

thread implementation Application.impl
  -- ...
  calls {
    Create_Socket : subprogram
      Transport_Layer::Create_Socket;
    Bind_Socket : subprogram
      Transport_Layer::Bind;
    Send_To : subprogram
      Transport_Layer::Send_To;
  };
  connections
  -- ...
end ApplicationTask.impl;

```

Figure 3: Behavioral specification of thread.

In figure 3 we show part of the behavioral specification of this thread, leaving out a number of details for reasons of clarity. Among these details are connections between subprogram calls. These allow to transfer results from one subprogram call to the other i.e. represent data flow from one layer to the other.

5. Discussion

At the end of this modeling process, we obtained the AADL model of a protocol stack with one process, one thread, many data components and numerous subprograms. Using only a single thread considerably limits capabilities to add structure to the model, but is our deliberate choice. The model we obtained has a full specification of structural properties and a partial specification of behavioral properties. We added only a limited number of non-functional properties to components, since our goal was to investigate the support AADL offers for modeling software architecture.

Our position is that AADL allows to model software architecture down to low level, but at the same time does not succeed in providing an abstraction level higher than that of an imperative programming language. As a result, the effort needed to model a software architecture in AADL is comparable to the effort needed to implement it in a language such as Ada. This conclusion is also reached by Vergnaud [10]. The use of a modeling language should bring higher efficiency to a development process, which is not the case with AADL at the current moment.

What we found most noticeable is the lack of high-level software architecture abstraction concepts. A generic component category would allow AADL users to model concepts such as ‘layer’ or ‘service’ in a software architecture. Since these concepts are present in many architectures, this problem also applies to domains other than protocol stacks. The lack of this type of concepts to describe software architecture is also what Shaw and Clements [11] mention as criticism to UML [12]. In section 4 we have shown that the most suitable way to model the concept of a layer in AADL is as a package of lower-level concepts. Unfortunately these packages provide no semantic support, e.g., it is not possible to model the interface of a layer as the interface of a package since AADL does not support package interfaces.

High-level software architecture abstraction concepts would allow the instantiation of architectural patterns, and therefore improve the efficiency of using AADL. This suggests the use of mechanisms for stepwise refinement in AADL. A first step that is needed to support this kind of mechanisms is the introduction of a well-defined set of high-level concepts, e.g., a generic component category. A second step is the definition of the syntax and semantics of refinement mechanisms.

In a working draft of the AADL v2 standard [13] an abstract component category is proposed. Abstract components have very few semantic restrictions and can be refined into concrete component categories. The abstract component category can be used to represent conceptual architectures. We know of no other work that addresses the issues presented in this paper.

6. Conclusion and Future Work

Our conclusion is that the AADL is a well-suited notation towards predicting non-functional properties of system and software architectures through its strong semantics. However, there is a lack of high-level software architecture abstraction concepts. The lack of this type of concepts means the effort needed to model software architecture in AADL is comparable to the effort needed to implement it. A generic component category complemented with refinement mechanisms would improve this situation. Modeling behavior in terms of subprograms adds significant detail, provided that refinement mechanisms are available to automatically add this behavior to an AADL model.

In our future work we will investigate whether abstract components introduced in AADL v2 can be used to represent high-level software architecture. In addition, we will investigate whether software architectures with abstract components remain semantically meaningful enough to allow analyses on the architecture.

References

- [1] SAE, "Architecture Analysis & Design Language (AADL)", AS-5506, 2004.
- [2] P. Binns, M. Englehart, M. Jackson, S. Vestal, "Domain-Specific Software Architectures for Guidance, Navigation and Control", *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, pp. 201-227, June 1996.
- [3] N. Medvidovic, R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93, 2000.
- [4] D. Garlan, R. Monroe, D. Wile, "ACME: An Architectural Description Interchange Language", *Proceedings of CASCON 97*, Toronto, Ontario, November 1997, pp. 169-183, 1997.
- [5] R.B. França, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, D. Thomas, "The AADL Behaviour Annex – Experiments and Roadmap", *Proceedings of ICECCS07*, pp. 377-382, Auckland, New-Zealand, July 2007.
- [6] A.S. Tanenbaum, "Computer Networks", Prentice-Hall PTR, ISBN 0130661023, pp. 1-950, 2002.
- [7] M. Felser, "Real-Time Ethernet - Industry Prospective", *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1118- 1129, June 2005.
- [8] J. Postel, "User Datagram Protocol", RFC (Request For Comment) 768, USC/Information Sciences Institute, August 1980.
- [9] IEEE, IEEE 802.3-2005 standard, "LAN/MAN CSMA/CD Access Method", available at <http://standards.ieee.org/getieee802/>, 2005.
- [10] T. Vergnaud, L. Pautet, F. Kordon, "Using the AADL to Describe Distributed Applications from Middleware to Software Components", *Proceedings of the 10th International Conference on Reliable Software Technologies Ada-Europe 2005 (RST'05)*, volume LNCS 3555, pages 67 - 78, York, UK, June 2005.
- [11] M. Shaw, P. Clements, "The Golden Age of Software Architecture: A Comprehensive Survey", tech. report CMU-ISRI-06-101, Institute for Software Research International, Carnegie Mellon University, February 2006.
- [12] Object Management Group, "Unified Modeling Language: Superstructure", OMG/2005-07-04, 2005.
- [13] SAE, "Architecture Analysis & Design Language (AADL)" v2, working draft. Unpublished.