An Architecture-Based Verification Technique for AADL Specifications*

Andreas Johnsen, Paul Pettersson, and Kristina Lundqvist

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
{andreas.johnsen,paul.pettersson,kristina.lundqvist}@mdh.se

Abstract. Quality assurance processes of software-intensive systems are an increasing challenge as the complexity of these systems dramatically increases. The use of Architecture Description Languages (ADLs) provide an important basis for evaluation. The Architecture Analysis and Design Language (AADL) is an ADL developed for designing software-intensive systems. In this paper, we propose an architecture-based verification technique covering the entire development process by adapting a combination of model-checking and model-based testing approaches to AADL specifications. The technique reveals inconsistencies of early design decisions and ensures a system's conformity with its AADL specification. The objective and criteria (test-selection) of the verification technique is derived from traditional integration testing.

1 Introduction

The architecture design phase is one of the most critical phases in the development process of software-intensive systems. The architecture specification is the initial development artefact representing the earliest design decisions made on the intended system's structure, functional properties and quality attributes (also known as non-functional properties or extra-functional properties). Design decisions involve the allocation of functional properties – which are closely related to a system's behavior, capabilities and services – to certain structures to achieve certain quality attributes. Furthermore, the architecture specification is used as a mutual communication blueprint among stakeholders and guides the implementation phase of the system. Consequently, the developed system will heavily depend on the architecture specification, which it ideally should conform to.

The design decisions established in the architecture design phase, or the absence of some, may impose incorrect properties of the system and thereby creating challenges in quality assurance processes. These incorrect structural, functional as well as non-functional properties may go unnoticed until later phases of the development process where a correction is known to be significantly more

* This work was partially supported by the Swedish Research Council (VR), and Mälardalen Real-Time Research Centre (MRTC)/Mälardalen University.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 105–113, 2011.

costly compared to a correction in the architecture design phase. Hence, evaluating the architecture specification is crucial in order to detect possible faults and inconsistencies before the development process progresses, reducing a significant amount of cost and time. Furthermore, in order to preserve the valuable effort made at the architecture design phase, an implementation of the system must be implemented in conformance with the architecture specification. The verification techniques used to tackle these challenges, i.e. *1) to evaluate an architecture specification* and *2) to test the conformance of an implementation with respect to its architecture specification*, rely on what kind of properties and their relations that may be described with the Architecture Description Language (ADL) used to specify the system's architecture.

Software-intensive systems are systems where software interacts with sensors, actuators, devices, other systems and people. Examples of such systems are embedded systems for aerospace, automotive and telecommunications. What these systems have in common is that they often are operating in dynamic, time- and safety-critical environments. One ADL that has been developed for this kind of systems, is the Architecture Analysis and Design Language (AADL) [1], which is widely used both within industry and the research community. In this paper we propose an architecture-based verification technique, for software-intensive systems specified by AADL, addressing challenge *1)* and *2)* mentioned above. The technique is based on formal constructs enabling automation of the verification activities where challenge *1)* and *2)* are tackled by adapting model-checking and model-based testing approaches to an architectural perspective. The objective of the technique is to evaluate the integration of components at both the specification-level and the implementation-level. Automated simulation of AADL specifications is not feasible directly from the artefact since AADL lacks formal semantics and implemented semantics. Formal semantics of a subset of AADL and an implementation thereof can be found in [2].

The rest of this paper presents an overview of AADL in Section 2. The architecture-based verification technique is introduced in Section 3, defined verification criteria are presented in Section 4, followed by concluding remarks in Section 5.

2 Preliminaries

AADL [3] was released and published as a Society of Automotive Engineers (SAE) Standard AS5506 [1] in 2004. It is a textual and graphical language used to model, specify and analyze software- and hardware-architectures of real-time embedded systems. The AADL language is based on a component-connector paradigm that describes components, component interfaces and the interactions (connections) among components. Hence, the language captures functional properties of the system, such as input and output through component interfaces, as well as structural properties through configurations of components and connectors. Furthermore, means to describe quality attributes, such as timing and reliability, are also provided. AADL defines ten types of component abstractions which can be divided into three groups:

- **Application software:** process, thread, thread group, data and subprogram
- **Hardware/Execution platform:** processor, bus, memory and device
- **Composite:** system

3 The Architecture-Based Verification Technique

This section presents an overview of the automatable verification technique for AADL specifications. The technique comprises both evaluation of specifications and the systems' conformity to them. It is depicted as a flowchart in Figure 1, where initially a system's intended architecture is specified using AADL. Such an artefact is commonly specified through a translation from something cognitive, an idea, a need or an informal/semi-formal requirement specification, but since it is informal, it is not possible to formally prove that the AADL specification correctly conforms to the informal one it is derived from [4]. Consequently, making this type of evaluation far from possible to automate and thus is out of scope in this technique. What is possible though is to formally reason about a system solely through the AADL specification, to prove its consistency and completeness, and later use it as a test model to perform model-based testing on. The different steps of the verification technique are as follows: The first step is to use the mappings/transformation rules (described in [2]) to transform an AADL specification to a timed automata model upon which automated formal verification can be performed.

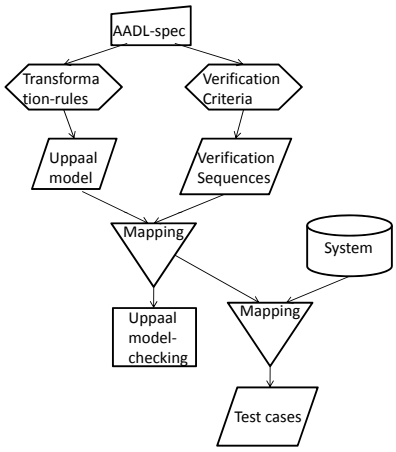

Fig. 1. Flowchart of the technique

The second step is to apply the architecture-based verification criteria (section 4) to the AADL specification. They define the test selection, i.e., what samples of the specification to evaluate and how they are extracted, and the coverage requirement, i.e., how many samples to evaluate. The samples generated from the criteria are sequences of component-integrations in terms of control-flows and data-flows.

Sequences are transformed, in the third step, to the corresponding timed automata paths through a structural mapping between them.

The outcome, a set of timed automata paths are required in the fourth step to be fully simulated by the Uppaal model-checker, by using temporal logics, in order to satisfy the criteria. The verdict from the simulations reveals the consistency and completeness of the AADL specification, where a correction of the specification should be made if it is shown inconsistent or incomplete.

The paths are later used in the fifth step to generate test cases to the implementation (model-based testing), to test the conformance of the implementation with respect to the architecture specification. Test paths are transformed to concrete test cases through a mapping between the architecture specification and its implementation (we assume identical name spaces between the AADL specification and the system).

4 AADL Verification Criteria

Due to features of an ADL, the primary focus of evaluation at this level is the integration of components as described by Eickelmann and Richardson [5] in their work about architecture-based defect prevention and detection. The idea of taking traditional data-flow and control-flow analysis criteria to the architectural-level has been proposed by Jin and Offut in [6], where explicit data-flow and control-flow properties through system architectures are defined. Based on these properties, they define general architecture-based testing criteria applicable to any ADL treating components and connectors as separate entities interconnected through their interfaces. Since AADL connectors do not have interfaces, and are dependent on the interfaces of the components they connect, the defined criteria are not applicable to AADL specifications. From the definition of the general criteria defined by Jin and Offut, we define architecture-based verification criteria specific to AADL based on the possible bindings of data-flow and control-flow properties (leading to sequences) described by AADL.

4.1 Verification Objectives

AADL specifications have explicit control-flows and data-flows through the architecture described by the informal semantics of AADL. These flows are dependent on how components transfer control and data through their interfaces (AADL component features). The possible interactions among components are represented by, and restricted to, four different types of connections: *port connections*, *data access connections*, *subprogram calls* and *parameter connections*.

Component abstractions within the software group, except data components, may have port interfaces for directional (in port, out port or inout port) interactions of typed data and events. A port can either be a data port (for transfer of data), a data event port (for transfer of data and associated control) or an event port (for transfer of control). Port interfaces can be connected through *Port connections*, which describe the transfer of control and data among/through concurrently executing thread components, or between a thread component and device component (threads and subprograms are used to represent functionality executed with or by device components). The flow of data or control through a port connection is determined by the directions of the connected ports.

Data components representing static data sharable among components are not accessible though ports, they are accessible through data access interfaces that may be declared with components of the software group. *Data access connections* describe the transfer of data where the data flow is determined by the value (read or write) of an *Access Right* property associated with the connection.

Subprogram components are not declared as subcomponents, instead they are called from thread or subprogram components through explicit *subprogram calls* declarations, expressing a flow of control from the calling component to the called subprogram. Call declarations may implicitly describe flows of data, where data can be provided to or received from a subprogram through parameter or data access connections. Parameters are interfaces of subprograms, similarly to data ports, for directional data interactions, where a parameter can be connected to a data port or another parameter through *parameter connections* describing the transfer of data. The data flow through a parameter connection is determined by the directions of the connected interfaces.

The runtime configuration of subcomponents and their interactions within a component may change if it is specified with *modes*. For each mode, it is possible to set the active components and connections, mode-specific subprogram calls and mode-specific properties. The transition from one mode to another is triggered by events derived from event ports, which is specified in a mode state machine. These modes can also be used to describe the internal logical execution (functional behavior) of thread and subprogram components. In addition to modes, a behavioral annex (BA) [7] extending the expressiveness of mode state machines has been developed to specify logical execution through automata syntactically similar to mode state machines. Thereby, it is possible to refine logical execution through state variables, states and transitions operating on a component's interfaces. Transitions can be specified with guards, such as boolean expressions and events, as well as actions, such as assignments and subprogram calls. Consequently, the control- and data-flows specified with the four different connections (described above) are refined if a behavioral model operates on the interfaces the connections connect.

The four different types of connections specify the architectural control-flows and data-flows of an AADL specification. Architectural control-flows are the different execution orders of architectural elements whereas architectural data-flows are the relations between definitions of data elements in a source component and uses of the corresponding data elements in a target component. These flows may be dependent on mode state machines, refined by the BA and constrained by associated properties where conflicts may occur between these constructs. The objective of the verification criteria is to ensure consistency and completeness of and between the flows, their refinements and their constraints through analysis of control-flow reachability, data-flow reachability and concurrency among flows:

Control-flow reachability: Every architectural element in an execution order should be able to reach the subsequent element to be executed in the order. The subsequent element should be reached without conflicting properties (constraints) of the execution order.

Data-flow reachability: Every data element should be able to reach its target component, where the data is used, from its source component, where the data is defined. The target component should be reached without conflicting properties of the data flow.

Concurrency among flows: Analysis of single interactions of data or control is not enough since there are implicit relations between them that may cause deadlocks in the system. The relations between the flows should not prevent control-flow reachability or data-flow reachability, and where the system should be free from deadlocks.

4.2 Verification Criteria

In order to extract control-flows and data-flows from an AADL specification, we define the atomic bindings of control and data that generates the flows, where we refer these atomic bindings to AADL relations. The relations are used to define integration verification sequences of control-flow and data-flow upon the verification criteria are defined.
In the definitions of AADL relations, an AADL Specification is represented as a 5-tuple:

$$AADLSPEC = \langle N, I, C, BAC, PAC \rangle$$

Where:
N is the set of Components $= \{n_1, n_2, ..., n_n\}$

I is the set of component interfaces $= \{n_x.i \mid n_x.i$ is a port, data acess, subprogram or parameter interface of n_x and $n_x \in N\}$

C is the set of Connections $= \{c(s,d) \mid c(s,d)$ is a port-, a data access-, a subprogram call- or a parameter-connection connecting the source interface $s \in I$ to the destination interface $d \in I\}$

BAC is the set of BA Connections $= \{bac(s,d) \mid bac(s,d)$ is an automaton path and the initial location in the path or a transition from the initial location is labeled with $s \in I$ and the last location in the path or a transition to the last location is labeled with $d \in I\}$

PAC is the set of Property Associated Constructs $= \{pac \mid pac \in I \cup C \cup BAC$ and is constrained by at least one associated property$\}$

Based on this representation of an AADL specification, the defined relations are:

1. **Connection Transfer Relation:** defines the data or control transfer that is generated between two interfaces connected trough a connection.
   CTR is the set of Connection Transfer Relations where $CTR \subseteq I \times I$ such that $\langle n_x.i_1, n_y.i_2 \rangle \in CTR$ iff $c(n_x.i_1, n_y.i_2) \in C$
2. **Connection Property Relation:** defines the constrained data or control transfer that is generated between two interfaces connected trough a connection.

CPR is the set of Connection Property Relations where $CPR \subseteq I \times I$ such that $\langle n_x.i_1, n_y.i_2\rangle \in CPR$ iff $c(n_x.i_1, n_y.i_2) \in C$ and $n_x.i_1$or $n_y.i_2$ or $c(n_x.i_1, n_y.i_2) \in PAC$

3. **Component Internal Relation:** defines the (possibly constrained) data or control transfer that is generated between two interfaces of a component that are connected through a connection or a BA.
   CIR is the set of Component Internal Relations where $CIR \subseteq I \times I$ such that $\langle n_1.i_1, n_1.i_2\rangle \in CIR$ iff $\langle n_1.i_1, n_1.i_2\rangle \in CTR \cup CPR$ or $\langle n_1.i_1, n_1.i_2\rangle \in BAC$
4. **Direct Component to Component Relation:** defines the (possibly constrained) data or control transfer that is generated between two components that are directly connected through a connection.
   $DCCR$ is the set of Direct Component to Component Relations where $DCCR \subseteq I \times I$ such that $\langle n_1.i_1, n_2.i_2\rangle \in DCCR$ iff $\langle n_1.i_1, n_2.i_2\rangle \in CTR \cup CPR$
5. **Indirect Component to Component Relation:** defines the (possibly constrained) data or control transfer that is generated between two components that are indirectly connected through one or several component(s). The relation is recursive in order to cover any number of interconnected components. The base case is:
   $ICCR$ is the set of Indirect Component to Component Relations where $ICCR \subseteq I \times I \times I^*$ such that $\langle n_1.i_1, n_3.i_4, t\rangle \in ICCR$ iff $\langle n_1.i_1, n_2.i_2\rangle \in DCCR$ and $\langle n_2.i_2, n_2.i_3\rangle \in CIR$ and $\langle n_2.i_3, n_3.i_4\rangle \in DCCR$ and $t = \langle\langle n_1.i_1, n_2.i_2\rangle, \langle n_2.i_2, n_2.i_3\rangle, \langle n_2.i_3, n_3.i_4\rangle\rangle$
   The recursive definition is:
   $ICCR$ is the set of Indirect Component to Component Relations where $ICCR \subseteq I \times I \times I^*$ such that $\langle n_1.i_1, n_x.i_y, t\rangle \in ICCR$ iff $\langle n_1.i_1, n_2.i_2\rangle \in DCCR$ and $\langle n_2.i_2, n_2.i_3\rangle \in CIR$ and $\langle n_2.i_3, n_x.i_y, t'\rangle \in ICCR$ and $t = \langle\langle n_1.i_1, n_2.i_2\rangle, \langle n_2.i_2, n_2.i_3\rangle, \langle t'\rangle\rangle$

From these AADL relations three verification sequences are derived, which are paths of the architecture specification:

1. **Component Internal Transfer Path:** If there exists a $\langle n_1.i_1, n_1.i_2\rangle \in CIR$, there exists a path from $n_1.i_1$ to $n_1.i_2$. The path is constrained if $\langle n_1.i_1, n_1.i_2\rangle$
   $\in CPR$.
2. **Direct Component to Component Path:** If there exists a $\langle n_1.i_1, n_2.i_2\rangle \in DCCR$, there exists a path from $n_1.i_1$ to $n_2.i_2$. The path is constrained if $\langle n_1.i_1, n_2.i_2\rangle \in CPR$.
3. **Indirect Component to Component Path:** If there exists a $\langle n_1.i_1, n_x.i_y, t\rangle \in ICCR$, there exist a path from $n_1.i_1$ to $n_x.i_y$ via t. The path is constrained if any pair in $t \in CPR$.

The AADL specification is consistent if each path is free from contradictory behavior, that is, each path does not contradict Control-flow reachability, Data-flow reachability and Concurrency among flows. The AADL specification is complete

if each path not yielding an end-to-end flow (typically a sensor-to-actuator flow) is subsumed in another path.

Upon the integration verification sequences, we define the three architecture-based verification criteria, which specifies requirements for a set of simulations or test cases to be adequate. Within following definitions, "S" is either a set of simulations of an AADL specification or a set of test cases for an implementation implemented to conform an AADL specification.

- **Component Internal Coverage:** requires that S covers all *Component Internal Transfer Paths*.
- **Direct Component to Component Coverage:** requires that S covers all *Direct Component to Component Paths*.
- **Indirect Component to Component Coverage:** requires that S covers all *Indirect Component to Component Paths*.

5 Conclusion

The AADL language is a formalism for development of safety-critical software-intensive systems. In this paper we have presented a verification technique covering the entire development process of a system specified with this formalism. The technique evaluates the consistency and completeness of an AADL specification and tests a systems' conformity to it. The entire development process is covered by adapting a combination of model-checking and model-based testing approaches to an architectural perspective. The adaption is performed through the definition of AADL-specific verification criteria. We are currently validating the technique against a system developed by a major vehicle manufacturer. The next step is to enrich the verification criteria with further details as well as formally define consistency and completeness of AADL specifications.

References

1. As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards no. AS5506 (November 2004)
2. Johnsen, A., Pettersson, P., Lundqvist, K.: An Architecture-based Verification Technique for AADL Specifications. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-253/2011-1-SE, Mälardalen University (May 2011)
3. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis and Design Language (AADL): An Introduction. Technical report, Technical report (2006)
4. Stocks, P., Carrington, D.: A framework for specification-based testing. IEEE Trans. Softw. Eng. 22(11), 777–793 (1996)
5. Eickelmann, N.S., Richardson, D.J.: What makes one software architecture more testable than another? In: ISAW 1996: Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints 1996) on SIGSOFT 1996 Workshops, pp. 65–67. ACM, New York (1996)

6. Jin, Z., Offutt, J.: Deriving Tests From Software Architectures. In: ISSRE 2001: Proceedings of the 12th International Symposium on Software Reliability Engineering, p. 308. IEEE Computer Society Press, Washington, DC, USA (2001)
7. Franca, R.B., Bodeveix, J.-P., Filali, M., Rolland, J.-F., Chemouil, D., Thomas, D.: The AADL behaviour annex – experiments and roadmap. In: ICECCS 2007: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 377–382. IEEE Computer Society Press, Washington, DC, USA (2007)