Developing Dependable Software-Intensive Systems: AADL vs. EAST-ADL[*]

Andreas Johnsen and Kristina Lundqvist

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

Abstract. Dependable software-intensive systems, such as embedded systems for avionics and vehicles are often developed under severe quality, schedule and budget constraints. As the size and complexity of these systems dramatically increases, the architecture design phase becomes more and more significant in order to meet these constraints. The use of Architecture Description Languages (ADLs) provides an important basis for mutual communication, analysis and evaluation activities. Hence, selecting an ADL suitable for such activities is of great importance. In this paper we compare and investigate the two ADLs – AADL and EAST-ADL. The level of support provided to developers of dependable software-intensive systems is compared, and several critical areas of the ADLs are highlighted. Results of using an extended comparison framework showed many similarities, but also one clear distinction between the languages regarding the perspectives and the levels of abstraction in which systems are modeled.

Keywords: Dependable systems, Software-intensive systems, AADL, EAST-ADL, Architecture description languages.

1 Introduction

One of the most critical phases in the development process of software-intensive systems is the architecture design phase. The architecture specification represents a set of design-decisions, which are analyzed and evaluated to ensure conformance with the system requirements. The efficiency and effectiveness of the evaluation method is largely dependent on the type of artifact being evaluated. Hence, the means used to design architectures of dependable software-intensive systems are critical to ensure quality of the system. Architecture Description Languages (ADLs) have been developed as means for designing systems' architecture.

Software-intensive systems are systems where software interacts with sensors, actuators, devices, other systems and people [1]. Examples of such systems are

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

embedded systems for vehicles, medical equipment and avionics. What these systems have in common is that they often operate in dynamic, time- and safety-critical environments where the components embedded within the systems are heterogeneous and have to meet real-time constraints. Two widely used ADLs within both industry and the research community are the Architecture Analysis and Design Language (AADL) [2], developed by the Society of Automotive Engineers (SAE), and the Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL) [3], initially developed by the Embedded Architecture and Software Tools - Embedded Electronic Architecture (EAST-EEA) project in the Information Technology for European Advancement (ITEA) programme and further refined by the Advancing Traffic Efficiency and Safety through Software Technology (ATESST and ATESST2) projects[4]. EAST-ADL was developed specifically for automotive systems, and AADL was initially developed for Avionics but now targets all large-scale software-intensive embedded systems and systems of systems, such as, aircraft, motorized vehicles, autonomous systems, and medical devices.

In this paper, we investigate these two ADLs and compare the level of support they provide developers to ensure correctness of software-intensive systems. An ADL should support activities – or tools performing activities – such as analysis, V&V, model checking (formal verification), code generation/synthesis, etc., by providing multiple perspectives with well defined semantics. At the same time, an ADL should support understandability and communication among stakeholders, by providing multiple levels of abstraction [5]. Generally, ADLs do not support both parts [6], which is critical for dependable systems since both parts contribute to systems' correctness.

The comparison is performed by applying an extension of Medvidovic and Taylor's "classification and comparison framework for software architecture description languages" [6]. In order to be able to compare AADL and EAST-ADL, Medvidovic and Taylor's framework is expanded with aspects of hardware architectures and typical quality attributes of software-intensive systems, which are timing and dependability.

The extended framework will be presented in Section 2, before overviews of the languages under comparison is given in Section 3. The results of applying the ADLs to the extended framework are presented in Section 4, which is followed by conclusions in Section 5.

2 The Comparison Framework

Medvidovic and Taylor developed a framework [6] for classification and comparison of software ADLs. In this paper, we extend their framework with hardware architecture aspects and vital quality-attributes of software-intensive systems. The new framework consists of, as in the original framework, a set of building blocks and their features (depicted in Table 1) that an ADL should provide. The main building blocks are **components**, **connectors** and **configurations**, where these components, in order to interchange architectural information, must have

Table 1. ADL Building Blocks (bold), their Features (italic) and vital quality-attributes

ADL: Building Blocks and Features
Components
Interface, Types, Semantics, Requirements, Evolution, Non-functional properties
Connectors
Interface, Types, Semantics, Requirements, Evolution, Non-functional properties
Configurations
Understandability, Compositionality, Refinement and traceability, Heterogeneity, Scalability, Evolution, Requirements, Non-functional properties
ADL: Vital Quality Attributes
Dependability
Timing

interfaces. Connectors are the interactions within the architecture whereas configurations define how each building block is combined to form an architecture description.

Architectures of software-intensive systems can be represented by these building blocks, which are abstractions of architectural elements. The framework developed by Medvidovic and Taylor restricts these building blocks to be abstractions of architectural elements of software. The extensions are reflected in the defined restrictions (given in section 2.1, 2.2, 2.3) of what the building blocks are abstractions of, which are: architectural elements of software, architectural elements of hardware and architectural elements of software mapped on hardware.

Within following subsections, an overview of each architecture building block, their features and the vital quality attributes is given.

2.1 Building Block: Component

Components are abstractions of main hardware/execution platform-units, computational software-units or composition of software and hardware-units. Computational software units refer to procedures/functions as well as entire applications. Main hardware/execution platform-units refer to complex hardware that may be associated with software to complete its functionality. Examples of such units are: sensors, actuators, processors, memories and communication links such as buses. Composition of software and hardware units refer to systems where computational software units are mapped to main hardware/execution platform units (e.g. flight control system, GPS system, electronic cruise control system, etc.). Components interact through their *interfaces* which are logical points of interactions between a component and its environment. An interface of a component describes the services a component provides and requires. The behavior model of a component, which here is referred to as component *semantics*, is an important feature of a component since it describes requirements and provides information for analysis and V&V activities. Components that are encapsulated within a certain subset of semantics and properties are here referred to as a component *type*, which can be instantiated several times within an architecture. Component types facilitates the ability to understand and analyze architectures since instances of a component type have common properties. Types are most often created by extensible type systems within ADLs, but built-in component types should also be provided. Components should be able to be modeled with external and internal properties specifying unacceptable borders, which we here refer to as component *requirements*. Furthermore, an ADL should provide modeling of *non-functional properties* (e.g. reliability, safety, performance, etc.) associated with components for V&V, simulation and analysis purposes. In order to control evolution of components within a system, i.e., modifications of component properties, the language should be able to support the *evolution* of the system. An ADL can support the evolution by supporting subtyping of component types as well as refinement of component features.

2.2 Building Block: Connector

Connectors are abstractions of interactions, where the method to interact may be of simple or highly complex nature. The nature may exclusively consist of software (e.g. data flows, control flows, function calls and access to data), hardware (e.g. wires) or a combination of the two (e.g. bus system). Connectors may have *interfaces*, specifying interaction points which components or connectors can be connected to. The behavior models of connectors which specify interaction protocols, are here referred to as connectors *semantics*. Similar to component semantics, connector semantics provide information for analysis and V&V activities, where the information is based on interconnection and communication requirements/properties. Connectors that are encapsulated within a subset of connector semantics and properties are here referred to as a connector *type*. These types are provided, similar to component types, by ADLs to facilitate modeling and understandability by reusable building blocks. Connector *requirements* assert interaction protocol properties by describing unacceptable borders. Connectors should also be able to be modeled with *non-functional properties*, which can not be derived from the connector semantics. As these interaction protocol properties are modified according to the *evolution*, ADLs should be able to support this evolution through subtyping and refinement of connector features.

2.3 Building Block: Configuration

ADL configurations define how each building block (components and connectors) is combined to form an architecture describing correct component connections, component communications, interface compatibility and that the combined semantics of the whole system result in correct system behavior. Since a system architecture partly serves as a mutual communication blueprint among stakeholders, the *understandability* of specifications is of great importance. An ADL specification should describe the topological system with understandable syntax or/and graphical notions, where an architecture configuration can be understandable without knowing components' and connectors' architectural details. Closely related to the understandability of an architecture configuration is the architecture *compositionality.* In order to provide an understandable architecture configuration, it is important to be able to describe the system at different abstraction levels, by abstracting away uninteresting details when concerning specific perspectives of the system. Such views can be provided by ADLs that have the capability to model a system hierarchically, where an architecture configuration may be contained within a higher abstracted component. As ADLs provide means for architectural description at different levels of abstraction, it is important to have *traceability* throughout the *refinement* of properties and relationships, from high levels of abstraction to the concrete system, in order to bridge the gaps between them. Since ADLs partly are used to facilitate development of large, complex and often highly heterogeneous systems, it is important that ADLs can meet these *heterogeneity* and *scalability* problems by providing possibilities to specify components and connectors described by external formal languages, and to be able to handle large and growing systems. *Evolvability*, which is closely related to scalability, does not only concern ADLs ability to accommodate to new architectural building-blocks to be added, but does also concern how ADLs can accommodate to incomplete architectural specifications, since it is unfeasible to make all design decisions at once. *Requirements* and *non-functional properties* of architectural configurations are not specific to individual components or connectors, but may be extracted from or are depended upon component- or connector-specific requirements and non-functional properties.

2.4 Vital Quality Attributes

Software-intensive systems are of highly complex nature with numerous critical quality-attributes. What software-intensive systems have in common is that they often are operating in safety-critical and time-critical environments. Consequently, two of the most important quality-attributes are dependability and timing. Even though one of the fundamental results of architecture-based development is increased dependability, as a result of abstracting complex systems to understandable and manageable blueprints, an ADL for software-intensive systems should explicitly provide means for dependability modeling. An ADL should facilitate safety- and reliability-analysis, such as for example, provide means for error modeling, reliability modeling, hazard analysis, risk analysis, and structures of requirements. Another critical aspect of software-intensive systems is timing since these systems often have to meet real-time constraints. An ADL should provide means to support modeling and analysis of timing requirements and properties, such as for example, end-to-end timing (sensor to actuator timing), latency, task execution time and deadlines.

3 ADLs Under Comparison

We present an overview of both ADLs in order to provide a basis for the comparison in section 4.

3.1 Overview of AADL

AADL (1.0) [7] [8] was released and published as a Society of Automotive Engineers (SAE) Standard AS5506 [2] 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 interaction (connections) between 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 are also provided. A system is modeled as a hierarchy of components where components that represent the application software are mapped onto the components that represent the hardware platform. A component is modeled by a component type and a component implementation. The component type specifies the external interfaces of the component in which other components can interact through, while the component implementation specifies the internal view of a component, such as subcomponents and their connections, and must be coupled to a component type.

Although a new version of AADL (AADLv2) [9] was published in 2009, the survey is restricted to the version of the language released in 2004.

3.2 Overview of EAST-ADL

The EAST-ADL [3] [10] is a domain-specific ADL for modeling and development of automotive electronic systems, where the language has modeling possibilities to specify software components, hardware components, features, requirements, variability and annotations to support analysis of the system. The language supports modeling of electronic systems at four different conceptual abstraction levels, namely: Vehicle level, Analysis level, Design level and the Implementation level. These abstraction levels reflect the amount of details in the architecture where abstract features and functions modeled in higher abstraction levels are realized to software and hardware components modeled in lower abstraction levels. The language provides a complete traceability through the different abstraction levels. The basic vehicle features (e.g. wipers and breaks) of the electronic systems are captured at the Vehicle level, the highest level of abstraction. These features are refined in related functions at the Analysis level by abstract elements representing software functions and devices interacting with the vehicle environment. The Design level represents a realization of the functionalities depicted at the analysis level, where the level allows further decomposition or restructuring of software functions and preliminary allocation of software elements. Specified devices are realized at this level into hardware architectures, such as sensors and actuators, including software for signal transformations. The lowest level of abstraction, the Implementation level is defined by using the Automotive Open System Architecture (AUTOSAR) standard[11].

4 AADL vs. EAST-ADL

AADL and EAST-ADL are compared according to the comparison framework given in section 2, where each architectural building block, their features and vital quality-attributes are analyzed and discussed based on the AADL standard specification [2] and the EAST-ADL standard specification [3].

4.1 Modeling of Components

Both AADL and EAST-ADL support modeling of all three component categories (i.e. computational software, main hardware/execution platform and composition of software and hardware). EAST-ADL refer these components to *features*, *functions* or *components*, depending on which conceptual abstraction level is considered whereas AADL exclusively refer to *components.*

Interface. AADL support modeling of five different types of component interfaces, or *component features* as referred to in the AADL standard. The different types of component interfaces are: *ports*, *data access*, *bus access*, *subprogram* or *parameter.* Ports are interaction points of software components for transfer of typed data and events. Data access interfaces are used to connect software components to static data whereas bus access interfaces are used to interconnect hardware components through bus components (built-in component types are depicted in the "types" section). Subprogram components may be used as interfaces of data components, representing methods that can be called by thread components. Parameters are interaction points of subprogram components for transfer of data. EAST-ADL on the other hand provides modeling of different interfaces, depending on which conceptual abstraction layer is being modeled. At the functional analysis level and the functional design level it is possible to model interfaces such as *client-server ports* and *flow ports.* Client and server ports are interaction points for communication between clients and servers, i.e. operations are required or provided by *client ports* and *server ports. Flow ports* are directional interaction points for exchange of data which is specified by associated data-types. The hardware design architecture, modeled at the design level, provides *pin* interfaces in which hardware elements can be connected to electrical sources, sinks and ground.

Types. The AADL language provides ten types of built-in component abstractions: *process*, *thread*, *thread group*, *data*, *subprogram*, *processor*, *memory*, *bus*, *device* and *system.* Note that a bus component represents an entity that interconnects hardware components (processor, memory, device and bus components) for exchange of data and control according to some communication protocol, and thus, it could be argued to be a connector type. Families of related components may also be modeled in the AADL language by an extension system where a component extending an antecedent component will inherit its antecedent characteristics, which can be refined or modified. EAST-ADL has built-in component types which encapsulate semantics and properties in relation to a certain abstraction level, in contrast to AADL which types encapsulate semantics and properties in relation to the concrete component that is abstracted by the language. For example, at the vehicle level, it is only possible to model *feature* components, and at the analysis level, it is only possible to model *function* and *device* components, where the encapsulated semantics and properties of these types are abstract. As the abstraction level decreases, the types are getting more concrete. For example, at the design level, it is possible to model hardware components of *sensor* or *actuator* type, and at the implementation level it is possible to realize (by using AUTOSAR) design level functions into software components types. The EAST-ADL language provides modeling of component types where occurrences of such instances, in a modeling artifact, are called typed *prototypes*. Modeling by these typing systems is provided at every abstraction level, except at the vehicle level. The EAST-ADL language does also provide modeling of *variability models*, which has similarities with modeling of component types but with a difference of the conceptual usage. The main conceptual usage of variability models is to facilitate controllability of product lines, and not mainly to facilitate understandability and analyzability. The variability management is provided at all the different conceptual abstraction levels, where related components can be merged to a component (which can be seen as a component type) with variability properties, meaning that the aspect of such a component can vary to another closely related aspect.

Semantics. Both AADL and EAST-ADL provide specification of components' behavior, but with some limitations which can be exceeded by language annexes and integrated tools. For example, the AADL language is extended with a *behavioral annex* [12], which provides modeling of components' behavior by using automata theory whereas the EAST-ADL language has traceability to behavior models based on external notations such as Simulink [4]. Both core languages provide sufficient modeling of behavior and functionality through modeling of component modes and triggers based on data, events or timing, for exchange of modes.

Requirements. The AADL language provides modeling of requirements through the generic *property* annotation, which does not only provide modeling of requirements, but also modeling of a component's functional properties (component semantics) as well as non-functional properties. Component properties can be specified with either the *component types* or the *component implementations*, to distinguish internal and external requirements of a component. The AADL language provides built-in properties (requirements) and possibilities to define new properties. EAST-ADL, on the other hand, treats requirements as separate entities that are associated to the target EAST-ADL element with a specific association, according to principles of SysML [13]. The concept of the requirement modeling is to provide an interface between OEMs (original equipment manufacturer) and suppliers.

Evolution. AADL provides means for structural evolution through its component extension system, where an instance of a component type can be used to type other components. Since AADL is built on a paradigm where a system is modeled as a hierarchy of components, its nature provides means for refinements of component features across different levels of abstraction. EAST-ADL does not allow modeling of component subtypes, because the EAST-ADL domain model (metamodel) only describes component types and their prototypes (type instances). However, EAST-ADL provides means for refinement across different level of abstraction, but with a hierarchical difference compared to AADL. Even though starting from a high abstraction level, AADL specifies components that are abstractions of concrete implementation components (e.g. a system component with sensors, processes and actuators as subcomponents), which then can be refined with other abstracted components (e.g. thread components), modeled inside components. EAST-ADL, on the other hand, starts with specification of components that are abstractions of features and functions (which themselves are abstractions), which can be decomposed in a lower abstraction by specifying these features and functions by using more concrete building blocks (components). EAST-ADL's terminology defines this as each abstraction layer realizes its antecedent layer.

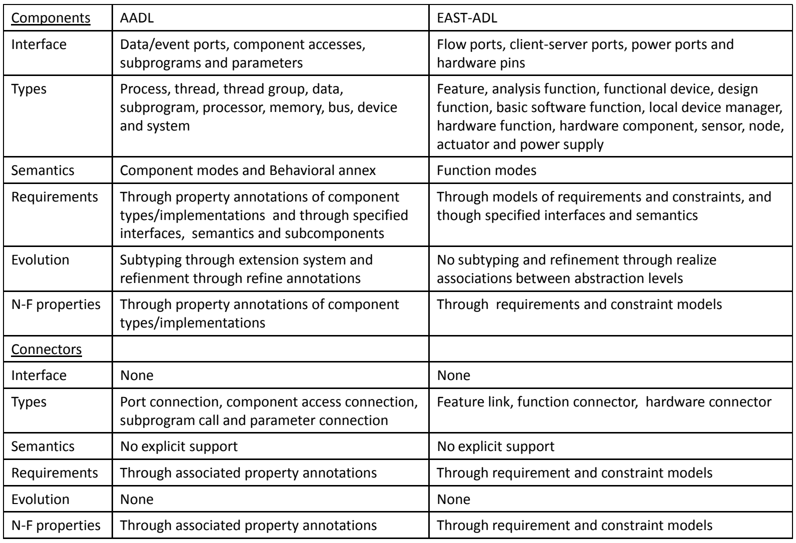
Non-functional properties. Both languages provide modeling of built-in non-functional properties of components, as well as means for specifying new non-functional properties. For example, for AADL components, there are built-in non-functional properties such as execution time, latency, throughput, startup deadline and write-time. For EAST-ADL components, there are properties such as safety, timing (e.g. execution time and latency), development cost, cable length and power consumption in addition to low-level properties represented through AUTOSAR elements. As can be seen by the presented built-in non-functional properties, EAST-ADL has properties of importance to higher levels of organizations compared to AADL.

4.2 Modeling of Connectors

Neither of EAST-ADL or AADL model connectors explicitly, instead connections are modeled "in-line" with the components, i.e. connectors are not first-class entities. Modeling of connectors within AADL and EAST-ADL basically consist of describing which component interfaces are connected. Connectors between software components are left out completely in the AUTOSAR language since the modeling concept is built on standardized component interfaces interacting through an abstract component called the Virtual Functional Bus (VFB).

Interface. EAST-ADL and AADL connectors do not have interfaces.

Types. EAST-ADL and AADL provides built-in connector types which encapsulates properties and semantics of a connector. Each connector type can be used to connect one or several types of component interfaces. For example, in AADL there is a *data access connection* connector type which can be used to connect *data access* interfaces, and in EAST-ADL there is a *FunctionConnector* connector type which can be used to connect *FunctionFlowPorts* or *ClientServerPorts*. AADL does also provide modeling of abstract information paths through



Components	AADL	EAST-ADL
Interface	Data/event ports, component accesses, subprograms and parameters	Flow ports, client-server ports, power ports and hardware pins
Types	Process, thread, thread group, data, subprogram, processor, memory, bus, device and system	Feature, analysis function, functional device, design function, basic software function, local device manager, hardware function, hardware component, sensor, node, actuator and power supply
Semantics	Component modes and Behavioral annex	Function modes
Requirements	Through property annotations of component types/implementations and through specified interfaces, semantics and subcomponents	Through models of requirements and constraints, and though specified interfaces and semantics
Evolution	Subtyping through extension system and refienment through refine annotations	No subtyping and refinement through realize associations between abstraction levels
N-F properties	Through property annotations of component types/implementations	Through requirements and constraint models
Connectors		
Interface	None	None
Types	Port connection, component access connection, subprogram call and parameter connection	Feature link, function connector, hardware connector
Semantics	No explicit support	No explicit support
Requirements	Through associated property annotations	Through requirement and constraint models
Evolution	None	None
N-F properties	Through associated property annotations	Through requirement and constraint models

Fig. 1. Modeling of Components and Connectors

a system, called AADL *flows*, to support control- and data-flow analysis such as end-to-end timing, reliability, resource management and latency.

Semantics. Semantics of AADL connections are defined by the type of the connection, types of components involved, as well as properties specified with the connections where the properties can be used to specify communication protocols. EAST-ADL connector types have predefined semantics, where means to specify additional semantics is not provided by the language.

Requirements. Modeling of requirements on connections is feasible in AADL through property statements, which is conceptually similar as with modeling requirements of AADL components. The same conclusion goes for EAST-ADL, where modeling of requirements on connectors is similar as on components.

Evolution. As both languages do not treat connectors as first-class entities, which can not be typed or reused, they do not provide means for controlling their evolution.

Non-functional properties. Modeling of non-functional properties of connectors is supported by both languages, similarly as with modeling of non-functional properties of components.

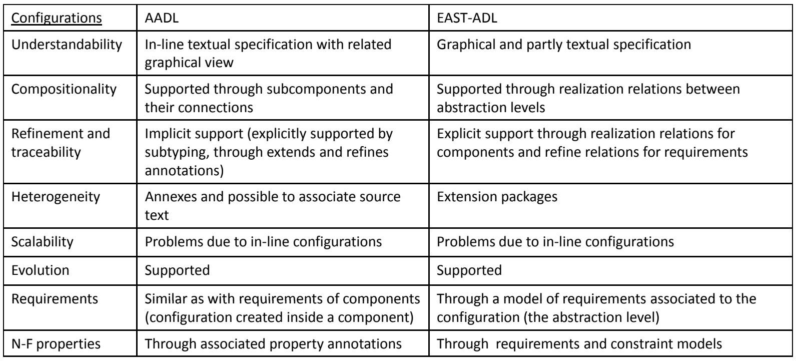
4.3 Modeling of Configurations

Architectural configurations can be modeled and expressed syntactically and/or graphically by the AADL language whereas in EAST-ADL configurations are modeled and expressed according to the UML-based metamodel. Modeling of a system configuration that may vary to another system configuration is provided by both languages, through their modes modeling features.

Understandability. Understandability of an AADL system configuration depends on which way it is expressed (syntactically or graphically). The graphical perspective provides a view of the system configuration that is easily understood. If a more detailed view is preferred, the syntactical model offers this, consequently with difficulties to perceive the whole system at once. Since there are precise relationships between a graphical and a syntactical configuration, both can be used simultaneous to enhance understandability. The understandability of EAST-ADL configurations depends upon which abstraction level is being viewed, since each level provides a complete configuration of the system with respect to the concerns of the level. Each abstraction level is modeled according to the metamodel, where mappings between elements among two neighbor abstraction levels are expressed by *realization* relationships, which provides means for expressing all the configurations at once.

Compositionality. Both languages support hierarchical description of systems at different levels of abstractions, however with a difference which we already have touched upon in Section 4.1. A system in AADL is modeled by specifying components and connections among components within a system component, which is not the case in EAST-ADL. In EAST-ADL a system is being viewed as completely specified according to a specific abstraction level with specific concerns. Note here that each abstraction level does not only provide a level of detail, but also specific concerns, and thus a certain perspective. However, systems are specified hierarchically in EAST-ADL since each architectural element in a specific abstraction level is realized by one or several elements in the subsequent (lower) level. Thus, with respect to the explicit abstraction layers, each component (excluding vehicle/top level and implementation/bottom level components) has relations to a "superior" and a "subordinate" component element(s). Consequently, the fundamental hierarchical differences between the languages are the relations between the members (components) of the hierarchy. In AADL, the hierarchy is generated by the notion of subcomponents, i.e., components are subsumed within another component and thus generates a kind of "subsumptive containment hierarchy". EAST-ADL, on the other hand, generates the hierarchy through the notion of realization. The realization is done through decomposition of components to more concrete elements provided at a subsequent abstraction level. Thus, the hierarchy is a kind of "compositional containment hierarchy" where vehicle level entities are composed of analysis level entities, which are composed of design level entities, which are composed of AUTOSAR entities.

Refinement and traceability. The languages' compositionality nature preserves traceability among properties and relationships throughout the refinement process. EAST-ADL explicitly relate requirement properties to each other, where requirements in the higher abstraction levels are refined to more detailed requirements in the lower levels.



Configurations	AADL	EAST-ADL
Understandability	In-line textual specification with related graphical view	Graphical and partly textual specification
Compositionality	Supported through subcomponents and their connections	Supported through realization relations between abstraction levels
Refinement and traceability	Implicit support (explicitly supported by subtyping, through extends and refines annotations)	Explicit support through realization relations for components and refine relations for requirements
Heterogeneity	Annexes and possible to associate source text	Extension packages
Scalability	Problems due to in-line configurations	Problems due to in-line configurations
Evolution	Supported	Supported
Requirements	Similar as with requirements of components (configuration created inside a component)	Through a model of requirements associated to the configuration (the abstraction level)
N-F properties	Through associated property annotations	Through requirements and constraint models

Fig. 2. Modeling of Configurations

Heterogeneity. AADL and EAST-ADL provide explicit support for specification by multiple specification languages, such as approved annexes (e.g. Behavioral annex, Error Model annex, etc.) for AADL and extension packages (e.g. ErrorBehavior, Requirements, Constraints, etc.) for EAST-ADL. AADL provide additional support for implementation details through predeclared properties where components can be associated with source text written in software languages such as C and Ada, modeling languages such as Simulink, and hardware languages such as VHDL. Implicitly, they support specification by multiple languages through model transformation into formal specification languages.

Scalability. Both languages have scalability issues since both are "in-line configuration ADLs", meaning that components and connectors are not modeled separately from the configurations. Adding new components to a configuration may require modifications to existing connections, since connections within in-line configurations are solely dependent upon the components they connect.

Evolution. Partial architecture specifications are supported by both languages. For example, the AADL language allows architectures of components without component implementation descriptions and with untyped data port interfaces. EAST-ADL allows architectures lacking of entire abstraction levels.

Requirements. Modeling of requirements on configurations is similar as modeling of requirements on components in AADL, since configurations are modeled inside components. EAST-ADL provide possibilities to associate requirements to a complete abstraction level.

Non-functional properties. Both languages support modeling of non-functional properties, such as timing and dependability, for architecture configurations.

4.4 Dependability

EAST-ADL consist of an explicit *dependability package* which provides means, such as hazard analysis, structuring of safety requirements according to their purpose in the safety life-cycle, formalizing requirements through safety constraints, analysis of fault propagation through error models and structuring evidence of safety, to specify and classify dependability. The dependability package is constructed to support the automotive safety standard ISO/DIS 26262. The AADL language does also support dependability modeling through the *Error model annex*, which defines a sub-language for modeling of error models that can be associated with AADL components. Through the error modeling features, the annex enables modeling and assessment of redundancy management, risk mitigation and dependability in architectures.

4.5 Timing

Specification of timing is provided by the AADL language through timing properties (such as deadlines, worst-case execution time, arrival rate, period etc.) as well as predefined concurrency, interaction and execution semantics. AADL has tool support for timing analysis through the Cheddar tool [14] and the Ocarina tool-suite [15]. Cheddar is a free real-time scheduling tool for analysis of temporal constraints. The tool supports both cyclic and aperiodic tasks, as well as a wide range of scheduling policies such as Rate Monotonic (RM), Earliest Deadline First (EDF), Deadline Monotonic, etc. Ocarina provides schedulability analysis of AADL models. EAST-ADL on the other hand has an explicit *timing package*, as with dependability, which provides means for modeling structures of timing constraints and timing descriptions. A timing structure is based on *events* and *event chains* that can be modeled across all abstraction levels. An *event* describes a distinct point in time where a change of state in the system takes place or it may also be an report of the current state. An *event chain* describes the temporal behavior of steps in a system, where the behavior is expressed by two related groups of events: stimulus and response. The chains is also used to specify built-in timing requirements on the different steps in the system. Timing analysis of EAST-ADL models is supported by the MARTE UML profile through the Papyrus add-in [16].

5 Conclusion

In this paper, we addressed the importance of an ADL for dependable software-intensive systems to support activities such as analysis, V&V, code generation/synthesis, etc., and at the same time support understandability and mutual communication. The classification and comparison framework for software Architecture Description Languages [6] developed by Medvidovic and Taylor was extended and used to compare the levels of support AADL and EAST-ADL provide these two aspects. The framework highlighted several areas when the languages were compared. One area was frequently highlighted during the comparison, which is that the metamodel of EAST-ADL has possibilities to describe systems at higher abstraction levels compared to the AADL standard. EAST-ADL provides means to model component types such as features, devices and functions of automotive systems, where a more detailed software architecture of concrete software components can be modeled by AUTOSAR, a complementary language to EAST-ADL. AADL on the other hand, models a system using abstractions of concrete system elements (e.g. processes and threads), which provide less freedom of the structure and how the functionality is obtained in the implementation. As EAST-ADL's point of view is on a higher abstraction level, hiding implementation solutions behind abstract features, devices and functionalities, it concentrates on system aspects of importance between the main parties within the automotive industry (e.g. between suppliers and OEMs) such as modeling of requirements, dependability, variability and timing of the system. This can be concluded in that the gap between an architecture description artifact and its implementation is larger when developing systems using EAST-ADL compared to using AADL, whereas the gap between the understandability of a system (as well as the controllability and the communicability) and its complexity is smaller. Therefore, EAST-ADL tend to primarily focus on understandability and communication of systems whereas AADL tend to be more appropriate for analysis tools, model checkers and compilers.

References

1. Wirsing, M.: Report of the beyond the horizon thematic group 6 on software intensive systems. Technical report, Thematic Group 6: Software-Intensive Systems (2006)
2. As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards no. AS5506 (2004)
3. The ATESST Consortium. East-adl 2.0 specification (November 2010), `http://www.atesst.org`
4. ATESST2. Advancing traffic efficiency and safety through software technology (November 2010), `http://www.atesst.org`
5. Medvidovic, N., Rosenblum, D.S.: Domains of concern in software architectures and architecture description languages. In: Proceedings of the Conference on Domain-Specific Languages (DSL 1997), p. 16. USENIX Association, Berkeley (1997)
6. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. 26(1), 70–93 (2000)
7. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis and design language (aadl): An introduction. Technical report (2006)
8. Hudak, J., Feiler, P.: Developing aadl models for control systems: A practitioner's guide. Technical report, CMU Software Engineering Institute (SEI) (2007)
9. As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards no. AS5506A (2009)
10. Cuenot, P., Frey, P., Johansson, R., Lönn, H., Reiser, M.-O., Servat, D., Tavakoli Kolagari, R., Chen, D.J.: Developing automotive products using the east-adl2, an autosar compliant architecture description language. In: European Congress on Embedded Real-Time Software (ERTS), Toulouse, France (2008)
11. AUTOSAR. Automotive open system architecture (November 2010), `http://www.autosar.org`

12. 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, Washington, DC, USA (2007)
13. SysML. Systems modeling language (November 2010), `http://www.sysml.org`
14. The cheddar project: a free real time scheduling analyzer (November 2010), `http://beru.univ-brest.fr/~singhoff/cheddar/`
15. Ocarina: An aadl model processing suite (November 2010), `http://www.ocarina.enst.fr`
16. Papyrus for east-adl (November 2010), `http://www.papyrusuml.org`