DESIGN OF REAL-TIME AUTOMATION SYSTEMS USING ARCHITECTURE DESCRIPTION LANGUAGES

Ricardo Bedin França*, Daniel Rech Gobbi*, Jean-Marie Farines*, Jean-Paul Bodeveix†, Leandro Buss Becker*, Mamoun Filali Amine†

*Departamento de Automação e Sistemas - DAS
Universidade Federal de Santa Catarina - UFSC - Florianópolis, Brasil

†Institut de Recherche de Toulouse - IRIT - Toulouse, France

Emails: rbedin@das.ufsc.br, gobbi@das.ufsc.br, farines@das.ufsc.br, bodeveix@irit.fr, lbecker@das.ufsc.br, filali@irit.fr

Abstract— This paper presents a component-based design methodology for real-time embedded systems, particularly well suited if they are complex and critical. The architecture description language AADL is the basis of this methodology. It constitutes an alternative to other modeling languages such as UML to represent all the system aspects, including its funcional/non-funcional features and platform constraints. The paper presents the main characteristics of the AADL language and shows its utilization in a case study on an autonomous vehicle control system. Next, the use of this model to verify desired system properties is also discussed.

Keywords— Critical Systems, Real-time, Programming, Design Methodology

Resumo— Este artigo apresenta uma metodologia baseado em componentes para desenvolver sistemas embarcados de tempo real, em particular quando se caracterizam pela sua complexidade e criticidade. A linguagem de descrição de arquitetura AADL é escolhida como alternativa de destaque a outras linguagens de modelagem como UML, pela sua capacidade para representar todos os aspectos do sistema com suas caracteristicas funcionais e não-funcionais e com as restrições da plataforma. Este artigo apresenta as principais caracteristicas da linguagem AADL e mostra sua utilização num estudo de caso de um sistema de controle de veículo autônomo. A seguir, é também discutido o uso deste modelo para a verificação das propriedades desejadas do sistema.

Keywords— Sistemas Críticos, Tempo Real, Programação, Metodologia de Desenvolvimento

1 Introduction

Critical automation systems are those systems in charge of control tasks requiring a high level of reliability, given that failures in such systems may lead to serious damage. Examples of such systems include avionics and spacial applications, satellite control, robotics, autonomous vehicles, and other ones. Normally these systems are also categorized as hard real-time, given the imposed time constraints. Moreover, they can also be considered as embedded systems, since they are composed of dedicated hardware and software co-located with the equipment under control.

Model-based techniques and tools have become popular to describe complex architectures including platform constraints and functional/non-functional requirements. A model is defined as a collection of all the artifacts that describe the system. The approach called model-driven development (MDD) (Selic, 2003), which is very popular for developing traditional computing systems, is now also used for embedded systems design.

Generally, MDD is a methodology for addressing complex development challenges by dealing with complexity through abstraction. Using this technique, complex systems are modeled at different levels of specificity. As the development proceeds, the model undergoes a series of transformations, with each transformation adding levels of specificity and detail. For the development of complex systems, MDD begins with the black-box specification of the system and, through a rigorous process of transformation, creates a model of the system; this model is ultimately implemented with tested system components. SysML[1] and UML (Booch et al., 1999) are examples of modeling languages that are currently used with MDD.

However, a lot of designers and researchers consider that these languages are not sufficiently well adapted to MDD methodology. The main arguments are: the duality among UML bottom-up approach and MDD top-down approach, the lack of abstractions to represent hardware and software architecture features, the non-hierarchical and informal relationship among the various diagrams, and the lack of refinement techniques.

Another problem about using UML and SysML languages is that they fail on providing a model from which the designer is able to verify functional and non-functional requirements such as time, safety, and reliability. To allow a more rigorous specification, formal modeling languages are needed.

In opposite, the Architecture Description Languages (ADLs) are well adapted to facilitate the design of real-time embedded systems, following a component system engineering approach with similarity of treatment for hardware and soft-

[1] Available at http://www.sysml.org

ware and a development process based on successive refinement in various abstraction levels.

Moreover, ADLs with formal extensions are a proper solution to address at the same time aspects like component building, model refinement, constraint satisfaction, and property verification.

This paper presents and discuss the use of an ADL to program critical automation systems. The reminder parts of the paper are organized as follows: section 2 presents an overview of the Architecture Analysis and Design Language - AADL; section 3 presents tools which supports AADL specifications; section 4 presents a case study that consists in the design of an autonomous vehicle control system, and discuss the proposed solution; finally section 5 presents the obtained conclusions.

2 Architecture Analysis and Design Language - AADL

2.1 Architecture Description Languages

Systems architecture is defined as the structure of the components of a system and their interactions. This concept can also mean guidelines and constraints which govern the system design and evolution over time (Garlan and Perry, 1995).

The system architecture has several roles in the system development (Clements, 1996), being a solid basis for communication among different designers, a blueprint or a pattern of development, and enforcing design decisions made in the beginning of the development.

The Architecture Description Languages (ADLs) have been evolved as a consequence of the systems architecture concepts.

A fairly high number of ADLs have been developed and tested to try and fulfill the needs of developers, and the scope of each language may vary. Currently there is no consensus about the optimal compromise between simplicity of representation and possibility of thorough architecture analysis, thus the choice of an ADL depends on the intention of the designers.

ADLs can be distinguished from some other languages (programming, module interconnection simulation, Petri Nets, Statecharts and object-oriented notations and languages) due to an explicit representation of connections that is not present in the other types (Medvidovic and Taylor, 2000). Clements (Clements, 1996) underlines some other differences:

- ADLs aim to design solutions, while requirement languages focuse problem description.
- ADLs avoid a deep level of abstraction, while programming languages design specific solutions for architecture elements.
- ADLs represent a system by its components, while modeling languages are most concerned with the behavior of the system as a whole.

The basic “building blocks” of an ADL are:

Components They are units of computation or data stores. Usually, ADLs separate component types from their instances, thus creating the possibility of reusing and extending component types to create multiple implementations and instances. Each ADL has its own means to declare the component’s constraints, as well as its internal behavior. Components must also be able to contain other components.

Connections ADLs have explicit connection declarations which link the components using their interfaces. Connection constraints which describe properties, such as the connection protocol, can also be used.

Configurations They are connected graphs of components and connectors that describe architecture. These configurations may be dynamic and so ADLs should be capable to represent dynamic configuration changes.

2.2 An overview of AADL

The Architecture Analysis and Design Language derived from the MetaH language (Vestal, 1998), which is aimed to meet the specific requirements of avionics systems. After years of evolution, AADL became a standard language for real-time systems architecture analysis and design. (SAE, 2004).

2.2.1 AADL Specifications

An AADL Specification describes a system architecture and contains component declarations. It has access to AADL Global Specifications – packages and property sets. Packages, much like in other languages, are global structures that organize component declarations in separate namespaces and may be accessed by other AADL specifications.

The AADL syntax can be extended by libraries and subclauses. The annex libraries enable designers to create their own sublanguages, if necessary, and use them in annex subclauses inside component declarations. Behavioral annex deals with the system dynamic behavior.

2.2.2 AADL Components

An AADL component is “some hardware or software entity that is part of a system being modeled in AADL”. An AADL component is usually declared in two parts: a ***type*** specification, which describes the component’s interface, and one or more ***implementation*** specifications, that represent internal aspects of a component.

The component implementations describe the contents of a component in terms of subcomponents, connections (between subcomponents or between a subcomponent and a feature of its containing component), operational modes, properties and flow implementations. The specification of subcomponents permits to create a hierarchic architecture.

Most component types contain ***feature*** declarations. Features represent the points of communication in a component's interface, and can be ports to receive data and/or events, component access which may be provided or required by another component, or subprograms that are entrypoints to the component. Component types may have properties that are common to all their implementations. Also, component types may extend other component types, refine partially declared features and change property values.

Software Components. The AADL Software Components are: data, subprogram, thread, thread group, and process.

Hardware Components. The AADL Execution Platform Components are: processor (which also contains scheduling and executing threads), device (which represents an interface with the environment), memory (which represents a generic storage unit) and bus (which is responsible for communication among the other execution platform components).

Processor and device may have ports and bus access features. A processor can contain memory subcomponents or communicate with it through common bus access. A device cannot contain any subcomponents.

2.2.3 AADL Connections

The connections in AADL are very simple, and do not represent high-level components. A connection may be between two ports (data, event or both) or a data/bus access connection. Ports are declared with their direction (in, out or both). Data accesses are declared specifying whether it is a provided or required data access.

Ports can be organized in port groups in order to simplify some connections; a port group can be declared as the inverse of another port group and then both port groups act like a plug and a socket.

The ports or port groups to be connected , has to be in the same direction when going through a hierarchy of components and in inverse direction when connecting components of the same level.

2.2.4 AADL Configurations

The basic mechanism provided by AADL to represent configurations of components and connections is the ***mode*** clause. Every component implementation may have operation modes, which look like finite automaton. One of the possible operational modes of a component must be declared as initial, and the mode transitions are triggered by the arrival of events. Implementation declarations, such as subcomponents, connections and properties, may be mode-dependent.

An architecture is described by the mapping of sofware components on the execution platform with the appropriate connections and configurations that will be explained *a posteriori*.

A system type may contain all kinds of ports, component access and server subprogram features, since it may represent a whole abstraction of architecture. Its implementation can have every AADL component (hardware and software) as its subcomponents, including other systems.

2.2.5 AADL Constraints

In order to give other characteristics to the components and connections, AADL has some predefined properties that can be used both in component types and implementations. Connections may also have properties.

Besides the standard AADL properties, it is possible for designers to create their own properties via property sets. Like packages, they are global AADL declarations that may be accessed by other specifications. A property has a name, a value (which can be constant or variable) and a list of elements (including components, connections and ports) for which the property may be used. A property value can be assigned with several types of expressions, such as numbers, intervals or references to components.

2.2.6 The AADL Behavioral Annex

The AADL Behavioral Annex [2] was created to improve some AADL weaknesses, especially concerning the component's behavior. This annex has introduced high-level composition concepts and a richer state representation than the standard AADL mode automaton.

A very important improvement in this Annex is the declaration of basic data types and their integration with the state machines. The types 'integer', 'float' and 'boolean' can also be used from the package Behavior.

The `property set` Behavior_Properties contains some properties which may be useful in data types. For example, an array of data can be specified with the property 'Multiplicity' and the 'Abstract' property is used in enumerated data type declarations. These enumerated data types can be described with each of their possible values being a feature.

The behavioral specifications are made with the use of extended automata, which may trigger

[2] Available at `http://gforge.enseeiht.fr/frs/?group_id=37`

a transition not only by the reception of an event, but also by the verification of a boolean expression, or even both (clause `when`). A transition may also trigger one or more actions, assigning values to variables and sending data and/or events. Thread states can be also declared as "complete": when such a state arrives, the thread is suspended and waits to be awaken (or, in case of a periodic thread, wait the next period).

3 A Development Framework

Since the beginning of standardization efforts, some tools have been developed to work with AADL. The main tool at the time of this work is the Open-Source AADL Tool Environment OSATE[3], which consists in a series of plug-ins running under the Eclipse platform[4]. OSATE permits creating AADL textual specifications (.aadl files) and XML-related object models (.aaxl files). It includes a parser to verify the syntax and some semantic aspects of the specification, and is able to translate an .aadl file to a .aaxl one and vice-versa. OSATE includes also some analysis prototype plug-ins that enable basic model verifications, such as the binding between threads and processors, flow checks and resource allocation.

An AADL graphical editor is provided by the Toolkit in OPen-source for Critical Applications & SystEms Development TOPCASED[5] project, which is the result of a cooperation among some French enterprises and academic institutions including UFSC. This graphical editor exchanges AADL specifications with OSATE.

Other tools can be used from the model: the scheduling analysis tool Cheddar[6], which is able to make schedulability analysis in AADL specifications; the Architecture Description Simulation ADeS[7] which is an Eclipse plug-in that runs with OSATE and simulates the execution of an AADL specification; and presently tools (as TINA[8], ALDEBARAN[9], UPPAAL[10]) which allows a formal verification of properties and are integrated in the TOPCASED environment.

4 Case Study: AADL Design of an Autonomous Vehicle Control System

This section presents the AADL design of a real-time embedded system for movement control of autonomous vehicle. This system is called Vehicle Control (VC) system. It is considered critical because some system failure or misfunction may result in serious injury to people as well as equipment damage. The VC system is also considered hard-real time due to time constraints as imposed computations, communications, and response times. Given all these constraints, designing such a system is a difficult and complex task.

The VC system was built using an on-board computer connected to external devices such as speed and distance sensors, a compass, a motor drive, a start/stop switch and a simple user interface. It also makes use of a pre-developed Generic Predictive Controler (GPC) for path following control, presented in (Gomes et al., 2006). The goal of this system is to follow a pre-defined path avoiding possible obstacles that may appear. It contains different operational modes, which are selected by user, according to the desired system configuration: it can optimize speed (optimal mode), save gas consumption (economic mode), or work in an intermediate configuration (normal mode).

4.1 General Architecture

The AADL specification consists of a top-down approach, which starts with the modeling of different types of components that offer a high-level view of the system architecture. Those components are further detailed through their implementations, where subcomponents and other internal aspects like concurrency are described. These basic steps may be done as many times as needed to attain a sufficiently detailed model according to its application. This level of the VC system architecture is presented in Figure 1.

First the computational platform (hardware) of the vehicle is specified. The VC system is composed by a processor, a memory within the processor, and a communication bus. Other hardware components (compass, speed sensor and so on) are modeled as AADL devices.

The next step is to describe the software components, which consist on a single process composed by threads, that are organized according to the functionality they are responsible for: path following control, anti-collision control, path selection and so on. These AADL thread components call subprograms whenever needed to perform computations (lines 19-20 and 23-24 in Figure 4). For instance, in this case study, a reusable subprogram implements the generic predictive control algorithm, GPC.

Then hardware and software components are assembled in a system component. The software deployment to the hardware components is done by means of the AADL properties "Actual_Processor_Binding" and "Actual_Memory_Binding". This is shown in lines 16 to 19 from Figure 2, which contains part of the AADL specification of the VC system.

[3] Available at `http://www.aadl.info`
[4] Available at `http://www.eclipse.org`
[5] Available at `http://www.topcased.org`
[6] Available at `http://www.beru.univ-brest.fr`
[7] Available at `http://www.axlog.fr/aadl/ades_fr.html`
[8] Available at `http://www.laas.fr/TINA`
[9] Available at `http://www.inrialpes.fr/vasy/cadp.html`
[10] Available at `http://www.uppaal.com`

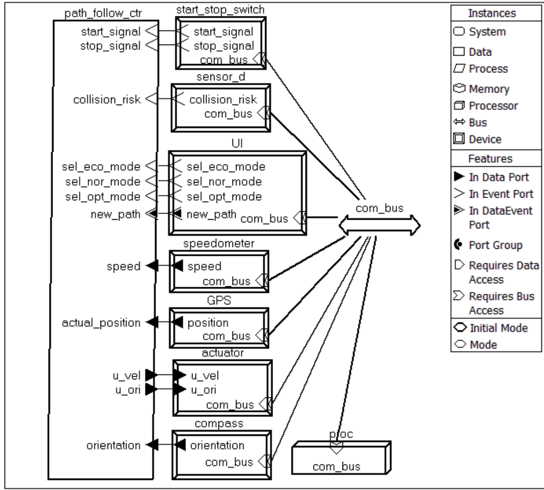

Figure 1: VC Component Diagram

The last remark on the general architecture of the VC system regards the different operation modes. Properties and mode-dependent declarations can be seen in the component specification, as shown in Figure 2.

```
1  system implementation VC.impl
2  subcomponents
3   path_follow_ctr: process control.impl;
4   ...
5   com_bus: bus communication;
6  connections
7  ...
8   event data port UI.new_path ->
9       path_follow_ctr.new_path;
10 ...
11 properties
12  Actual_Processor_Binding => reference
13      proc applies to path_follow_ctr;
14 end VC.impl;
```

Figure 2: The VC components in AADL

4.2 Modeling the System Behavior

This section addresses the behavior of the GPC thread that is part of the path-following controller component from figure 1, which constitutes the core of the VC system. The GPC thread is periodic and contains three states named *running*, *suspended*, and *emergency_stop*. These states are characterized "complete", following the definition presented in section 2.2.6. Figure 3 shows the extended automaton that represents the behavior described with AADL in Figure 4.

The GPC thread is created in the *running* state. This occurs as soon as a reference path is set by the user through the interface, and after the system has been started through the start/stop switch. The periodic behavior is represented by the **dispatch** event, which is triggered every period. In each period, the system reads the sensors data, computes the GPC algorithm (based

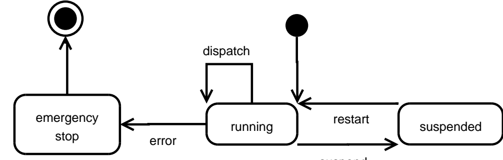

Figure 3: Extended automaton of the GPC thread

on the selected operational mode), and sends the calculated references to the motor drives. If the controller is not able to compute these references within the period, an **error** event occurs and the thread state changes to the *emergency_stop* state.

In case of a potential collision, the anti-collision control thread reports the occurrence to the GPC thread by means of the **suspend** event. The GPC thread is suspended when the *running* state is complete. Immediately, the anti-collision thread takes control of the vehicle. As soon as this component ends its activity, a **restart** event causes the GPC thread to retake control of the vehicle, returning to the *running* state.

4.3 Modeling Properties

Properties of the system can be also described in AADL, before checking their satisfaction on the system behavior.

The verification process consists in checking whether the desired system properties are satisfied by the behavior description. Properties are modeled with Temporal Logic, such as the Temporal Logic of Actions (TLA) (Lamport, 2002) or another one; model-checkers can be used for verification. If a property is not satisfied, a counter-example scenario is normally provided by the model-checker to help the designer to find the cause of non-satisfaction in the system specification and to correct it on the system model. Some tools used for verification in the TOPCASED context are cited in section 3.

It is also possible to use Temporal Logics (as TLA) to define the semantic of the AADL specification. Afterwards, holding the semantic, the specification can be refined (for example, introducing operating systems features or platform details from a more abstract specification), and new properties can be introduced and checked.

4.4 Discussion

The component system engineering seems a more natural and clear approach to build automation embedded systems, considering the engineer view for system modeling and implementation.

In order to support this component approach, the Architecture Description Languages (ADLs), particularly Architecture Analysis and Design Language (AADL) is a well suited tool to deal with complex control systems. AADL represents

```
1  thread GPC_computation
2  features
3    suspend: in event port;
4    ...
5  properties
6    Dispatch_Protocol => Periodic;
7    SEI::Priority => 1;
8    Period => 20 Ms;
9  end GPC_computation;
10
11 thread implementation GPC_computation.impl
12 annex behavior_specification {**
13 states
14   running: initial complete state;
15   suspended: complete state;
16   emergency_stop: complete state;
17 transitions
18   running -[]-> running { subprograms::
19 GPC_algorithm!
20 (ref_path->r_path, u_ori->u_orientation); };
21   running -[ error? ]->
   emergency_stop { u_speed:= 0; };
22   running -[ suspend? ]-> suspended {  };
23   suspended -[ restart? ]->
   running { subprograms::GPC_algorithm!
24 (ref_path->r_path, u_ori->u_orientation); };
25 **};
26 end GPC_computation.impl;
```

Figure 4: AADL behavior of the GPC thread

systems as a set of components (hardware and software) that communicate by means of connectors. The main difference between AADL and other modeling languages lies on its component-based approach, while the other ones represent system as a whole.

AADL also allows to model the behavior in software component as an extended automaton, with receiving events and boolean expressions to trigger transitions. This formal behavior representation and the possibility to model properties with temporal logic formulas allow to verify the satisfaction of desired properties by the system and to correct it, if necessary.

The use of AADL language provides a more clear view of the system architecture in comparison to UML and SysML. It is also better adapted to MDD system development than these languages. Following the MDD approach, AADL uses refinement techniques. Additionally, it provides a formal model and allows early and lifecycle tracking of modeling and analysis. AADL specification is a blueprint for all design phases; moreover the use of formal models for properties and behaviors guarantees the correctness and the coherence of the system building.

5 Conclusions

Powerful design methodologies are required to help designers to build complex systems, guaranteeing implementation correctness with respect to functional requirements but also non-functional ones, such as time, safety, reliability, and energy-economy. At the same time, these methodologies result in reducing development time and cost.

In this paper, a component-based methodology is presented to build complex and critical systems. This methodology is supported by a well suited architecture language, AADL. Its behavioral extension (based on extended automata) and the associated formal representation of properties (from a temporal logic language) allow to guarantee the correctness of the system conception and to provide all the system requirements and constraints.

An actual application of an autonomous vehicle control system allowed to show the main characteristics of the methodology, particularly with respect to modeling.

References

Booch, G., Rumbaugh, J. and Jacobson, I. (1999). *The Unified Modeling Language User Guide*, Addison-Wesley.

Clements, P. C. (1996). A Survey of Architecture Description Languages, *IWSSD '96: Proceedings of the IWSSD '96*, IEEE Press, Washington, USA, p. 16.

Garlan, D. and Perry, D. E. (1995). Introduction to the Special Issue on Software Architecture, *IEEE Trans. Soft. Eng.* **21**(4): 269–274.

Gomes, G. K., Raffo, G. V., Kelber, C. R., Rico, J. E. N. and Becker, L. B. (2006). Seguimento de trajetoria de um veiculo mini-baja com cpbm, *Anais do XII Congresso Brasileiro de Automática*, Sociedade Brasileira de Automática.

Medvidovic, N. and Taylor, R. N. (2000). A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Trans. Soft. Eng.* **26**(1): 70–93.

SAE (2004). *Architecture Analysis & Design Language (AADL) AS5506.*

Selic, B. (2003). The pragmatics of model-driven development, *IEEE Software* **20**(5): 19–25.

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley.

Vestal, S. (1998). *MetaH User's Manual*, Honeywell Technology Drive.