Development Process for Critical Embedded Systems

L.B. Becker[1], J.-M. Farines[1], J.-P. Bodeveix[2], M. Filali[2], F. Vernadat[3]

[1]Dept of Automation and Systems – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88040–900 – Florianopolis – SC – Brazil

[2]IRIT-CNRS, Université de Toulouse
Toulouse, France

[3]LAAS-CNRS, Université de Toulouse
Toulouse, France

{lbecker,farines}@das.ufsc.br, {bodeveix,filali}@irit.fr, francois@laas.fr

Abstract: Designing safety critical systems is a complex task due to the need of guaranteeing that the resulting model can cope with all the functional and non-functional requirements of the system. Obtaining such guarantees is only possible with the use of model verification techniques. This paper presents an approach aimed to fulfill the needs of critical system design. The proposed approach is based on the Architecture Analysis and Design Language (AADL), which is suitable to describe the system's architecture. It contains a sequence of model transformations that easies the verification of the designed AADL model and so assures its correctness. It must be highlighted that this is not performed in a single step, as it is possible to verify AADL models with different abstraction levels, which allows successive refinements in a top-down approach. We use a case study from an Autonomous Parking System to illustrate the proposed development process.

Keywords: safety-critical systems, design approach, model-verification

1. Introduction

Modern safety-critical systems are getting more and more complex and, at the same time, have become indispensable nowadays. Almost every system that in the past was simply mechanic (e.g. cars, trains, airplanes) is now equipped with an embedded computing systems. Also, most of the times, such systems are safety-critical.

In order to handle such increasing complexity it is necessary to use a development process based on System Engineering. These techniques should both facilitate the modeling discipline and provide model-verification facilities. Model-verification is crucial for safety-critical systems design because it allows guaranteeing that the designed model respect the application requirements and constraints.

In this context, the Architecture Analysis & Design Language (AADL) [Feiler et al. 2006] seems to be a suitable choice. AADL is a modeling language that allows early analysis of a system's architecture. It supports the modeling of both software and hardware components in a hierarchical manner using a set of components connected through ports. AADL defines properties that can be attached to modeling elements in order to give an abstract specification of the dynamic architecture of the system. Real-time constraints are attached to threads, ports, buses, and processors (e.g. dispatch protocol, period, deadline, processing power, hardware-software mapping, etc). The AADL language can also be extended by defining new properties or by attaching specific languages to some elements.

Although AADL precise semantics makes it suitable for model verification, how to perform such a task is still an open question. For this reason, we present in this paper a solution that overcomes this problem. Our approach consists in supporting model verification taking into account irregular behaviors and data. Another important feature from our proposal is that it follows the Model Driven Engineering (MDE) principles, as design is intended to remain in high-level abstraction levels and does not need to worry about the low-level details from the performed model transformations.

We can say that the proposed process supports the safe design of the system's architecture, once the resulting system architecture goes through several verification steps in order to assure its correctness. To reach this goal it is performed a sequence of model transformations, maintaining the principles of MDE. It starts with an AADL model and finishes with an automaton model that can be verified.

The rest of the paper is structured as follows: Section 2 discusses some related works. Section 3 gives a brief introduction to the AADL language. Section 4 presents the proposed development process and our autonomous parking case study. Section 5 presents the techniques and toolset used to verify temporal properties of AADL models. Finally, section 6 draws the conclusions of this work.

2. Related Methodologies and Tool Support

Designing new generations of embedded real-time systems is so complex that became mandatory to work with higher abstractions (namely computational models) previous to implementation. The Model Driven Engineering (MDE) [Schmidt 2006] is, for instance, an initiative to help developers to manage software development complexity using models at the very beginning, and with different abstraction levels. The key aspect from this technology is the design of models that are decoupled from their target platform. Among the main benefits of the emerging MDE approach it should be highlighted its enhanced possibilities for early model verification.

In fact, many recent tools have been proposed to support different kinds of verification. With respect to our concerns, timing verification tools have been an active area of research over these last years. It is interesting to remark that although most of these tools are based on existing theoretical models, e.g., timed automata, Petri nets, the limitations (especially with respect to combinatorial explosion and scalability) of which are well known, the effort has been undertaken to achieve them. In fact, it is hoped that first, the abstraction and the structure brought by the model driven approach and second, the adoption of a specific execution model will help to struggle against these limitations. Along these lines, we can cite the Cheddar [Dissaux and Singhoff 2008] scheduling tool which proposes dedicated analysis for the AADL execution model. Currently, it considers mainly analytical models. Future versions should take into account more detailed behavior descriptions [Franca et al. 2007]. The tools Uppaal Port [Håkansson et al. 2008] and Pola[Berthomieu et al. 2007] are based on the traditional model checking approach. Uppaal Port is based on timed automata and supports component based development. In order to reduce the combinatorial explosion Uppaal Port adopts a synchronous like execution model which restricts interleaving of the asynchronous approach. Moreover, it proposes partial order techniques for reducing space explorations. The tool Pola is based on timed Petri nets, and it proposes specific support for the AADL execution model.

3. A Brief Overview of AADL

AADL is an architecture design language standardized by the SAE. This language has been created to be used in the development of real time and embedded systems. As a successor of MetaH, AADL capitalizes more than 10 years of experiments. MetaH is a language developed by Honeywell Labs and used in numerous experiments in avionics, flight control, and robotic applications. AADL also benefits from the knowledge on ADLs acquired at CMU during the development of several ADLs, like ACME and Wright.

AADL contains all the standard concepts of any ADL: components, connectors used to describe the interface of components, and connections used to link components. The set of AADL's components can be divided in three partitions: the software components (process, thread, thread group, subprogram, and data), the hardware components (processor, bus, memory, device), and a system component. Components can communicate through ports, synchronous calls, and shared data. A process represents a virtual address space, or a partition, this address space includes the program defined by its subcomponents. A process must contain at least one thread or thread group. A thread group is a logical organization of threads in a process. A thread represents a sequential flow of execution, it is the single AADL component that can be scheduled. A subprogram represents a piece of code that can be called by a thread or another program. A data models a static variable used in the code, they can be shared by threads or processes.

A processor is an abstraction of the hardware and the software in charge of the scheduling and the execution of threads. The memory represents any platform component that stores data or binary code. The buses are communication channels used to connect different hardware components. The devices represent interfaces between the system described and its environment.

Systems allow composing software components with hardware components. The interactions can be defined at a logical and a physical level. At a physical level, software components are associated to hardware components, a thread to a processor, or a data to a memory for example. The logical level is used to describe the communication between hardware and software. At a logical level we can define communication connections between processors or devices and software components.

AADL uses the notion of mode to determine a set of active components. This mechanism allows describing dynamic architectures through a static set of components. We consider here the behavior annex [Franca et al. 2007] attached to threads or devices, which is used to specify an abstract behavior for these components, allowing to make data dependent analysis.

4. The Proposed Development Process

This section presents our proposed development process for critical embedded systems. It is possible to say that this process supports the safe design of the system's architecture using MDE's principles. By safe design we mean that the resulting system architecture goes through several verification steps in order to assure its correctness. To reach this goal it is performed a sequence of model transformations, which starts with an AADL model and finishes with an automaton model that can be verified. This section skips the details of the verification chain (which is covered in the next section) and concentrates in the high-level steps of the proposed process, which are shown in Figure 1.

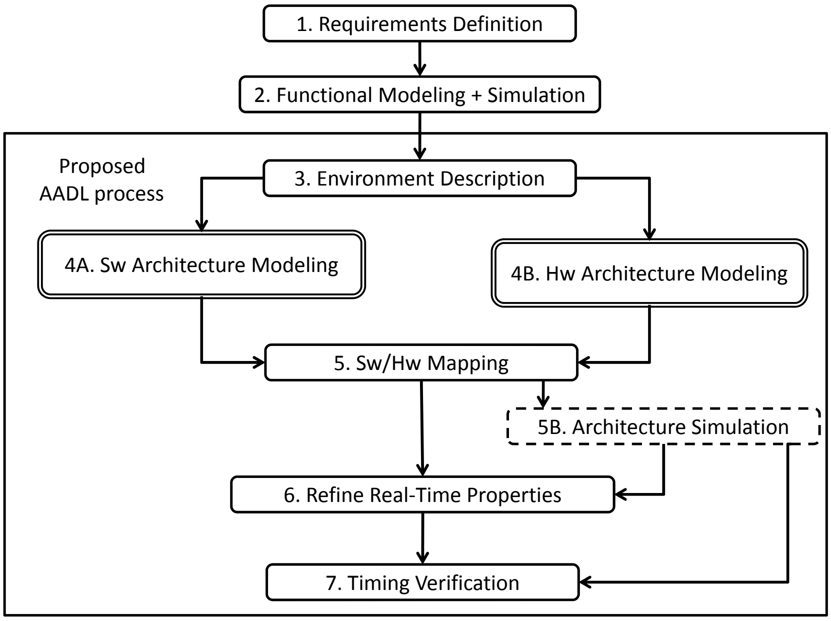
1. Requirements Definition
2. Functional Modeling + Simulation
Proposed AADL process
3. Environment Description
4A. Sw Architecture Modeling
4B. Hw Architecture Modeling
5. Sw/Hw Mapping
5B. Architecture Simulation
6. Refine Real-Time Properties
7. Timing Verification

Figure 1. Proposed Design Flow

We understand that, as in any system development, the initial step is the definition of the functional and non-functional requirements of the system, resulting in a set of requirements. Then it is followed by the design of a functional model for the system (e.g. Lustre or Simulink model). The proposed process itself starts in step-3 with the design of the AADL model, providing the specification of the external devices (environment) that interact with the system. step-4 is split in two parts: (4A) software architecture modeling/verification and (4B) hardware architecture modeling. The overall result here should be an AADL model with basic properties already verified and a hardware architecture potentially capable to run the designed software model. In step-5 a mapping from the modeled software components to the hardware model is performed. The result is a complete AADL model. In step-6 it is suggested that the real-time properties of the AADL model should be updated with the precise timing information coming from the simulation of the software in the target platform, which is conducted in step-5B. The proposed development process is concluded in step-7 with the final model verification, which uses as input the AADL model updated with the precise timing information. After that, it should be possible to make automatic code-generation of the application.

It is important to highlight that the design flow among the steps is not unidirectional. Every time that a verification step fails the designer should either backtrack to higher abstraction levels of the AADL model and its properties or change assumptions made in earlier levels. For example, if there is an error in the timing verification (step-7), then the designer should be able to judge if the problem is due to the result of step-4A (proposed software architecture) or to the result of step-4B (target hardware architecture).

The reminder parts of the current section details the steps depicted in Figure 1. We use an Autonomous Parking (AP) System case study to elucidate the work performed in each step. Moreover, we concentrate the discussions on the software architecture modeling (step-4A). The target hardware architecture definition (step-4B), although very important in the context of the proposed process, should be subject of additional investigation and therefore is left out of this work.

4.1. Requirements Definition

The initial step in any development methodology is to define the requirements of the system to be developed. This includes both functional requirements (FR) and non-functional requirements (NFR). While the former depicts the main functionalities to be performed by the system, the latter imposes restrictions to those functionalities.

Table 4.1 presents the list of requirements from the AP system, which has three main functionalities: (FR1) start/stop the system using a GUI; (FR2) search for a parking slot; and (FR3) park the car. NFRs are like properties that must be satisfied by the related FR.

4.2. Functional Modeling and Simulation

In many applications, especially those related with control systems, it is required to first design a functional model of the system and to simulate it before any design decision on the system architecture is carried on. This is used either to provide a deeper understanding of the system functionalities or to test/simulate control solutions in early development stage. Tools like Scade/Lustre and Matlab/Simulink are often used for this propose.

4.3. Environment Description

The third step of the process consists of using AADL to describe the environment that interacts with the system under development. So, the set of interactions of the system with the external devices, such as sensors, actuators, user interface, etc.

For this reason we use here a high-level AADL diagram. Figure 2 presents the diagram designed for the AP system, where it is possible to observe the main system in the center (named `ParkingCtrl`) surrounded by the devices. An advantage of using AADL for such purpose is that it allows detailing each message exchanged between the system and the devices, including information like data type, arrival pattern, and time constraints.

In this phase two different kinds of external devices can exist: reused devices and new devices. While devices like sensors and actuators are normally reused from previous applications, devices like User Interfaces (UI) are normally designed on demand for each application. New devices can be subject of formal verification prior to its use in the model. Therefore it is necessary to specify the device's behavior. In the scope of this work it is suggested to describe behavior using finite automatons.

To exemplify the verification of devices behavior in the AP system we selected the UI device (`UIController`). A possible behavior of this device is depicted in Figure 3. This state-transition diagram states that, independently of the status of the application, the

<table>
<tr><td colspan="2"><b>FR1 - Start/stop the system using a GUI</b></td></tr>
<tr><td colspan="2"><b>Description:</b> The system must be explicitly activated by the driver to start operation</td></tr>
<tr><td>NFR1.1 - Maximum speed</td><td>To start the system the speed must be kept at $\leq$ 20Km/h</td></tr>
<tr><td>NFR1.2 - On operation</td><td>The system must inform the user while it is working</td></tr>
<tr><td>NFR1.3 - Finished</td><td>The system must inform the user as it is turned off</td></tr>
<tr><td colspan="2"><b>FR2 - Search for a parking slot (<i>real-time operation</i>)</b></td></tr>
<tr><td colspan="2"><b>Description:</b> When activated, the system must start searching a new park slot as the vehicle moves forward</td></tr>
<tr><td>NFR2.1 - Driver alert</td><td>The system must inform the user when a new parking slot is found</td></tr>
<tr><td>NFR2.2 - Safety</td><td>If the speed is too high (over 20km/h) than it is not possible to search a parking slot</td></tr>
<tr><td colspan="2"><b>FR3 - Parking (<i>real-time operation</i>)</b></td></tr>
<tr><td colspan="2"><b>Description:</b> The driver must trigger the beginning of the parking after a parking slot is found.
The system controls the speed and direction of the vehicle.</td></tr>
<tr><td>NFR3.1 - Safety</td><td>The system is allowed to start parking only if the current speed is zero</td></tr>
<tr><td>NFR3.2 - Emergency Stop</td><td>The system must be halted immediately if the driver moves the wheel</td></tr>
<tr><td>NFR3.3 - Finish allert</td><td>The system must alert the driver when the parking maneuver is finished</td></tr>
</table>

Table 1. Requirements set of the Autonomous Parking (AP) System

driver can always turn off the system (NFR1.3). This can be proof by the existence of the user event `Off!` in every possible execution state of the system. Although very simple, this is an example to show that it is possible to use verification already at this level.

4.4. Software Architecture Modeling

The software architecture modeling (step-4A) is probably the most important phase of the proposed design process. This phase may have several steps of iterations, as designer may create several AADL submodels, from more abstract to more detailed ones, and that all these models should have its properties verified.

In the first iteration the designer must detail the AADL system process (e.g. `ParkingCtrl` at Figure 2) into a set of subcomponents (either processes or threads). As this detailing is completed, model verification is performed, as explained in the next section. If the verification fails (many times due to the lack of information in the model at the moment), a new refinement in each component should take action, starting new iterations.

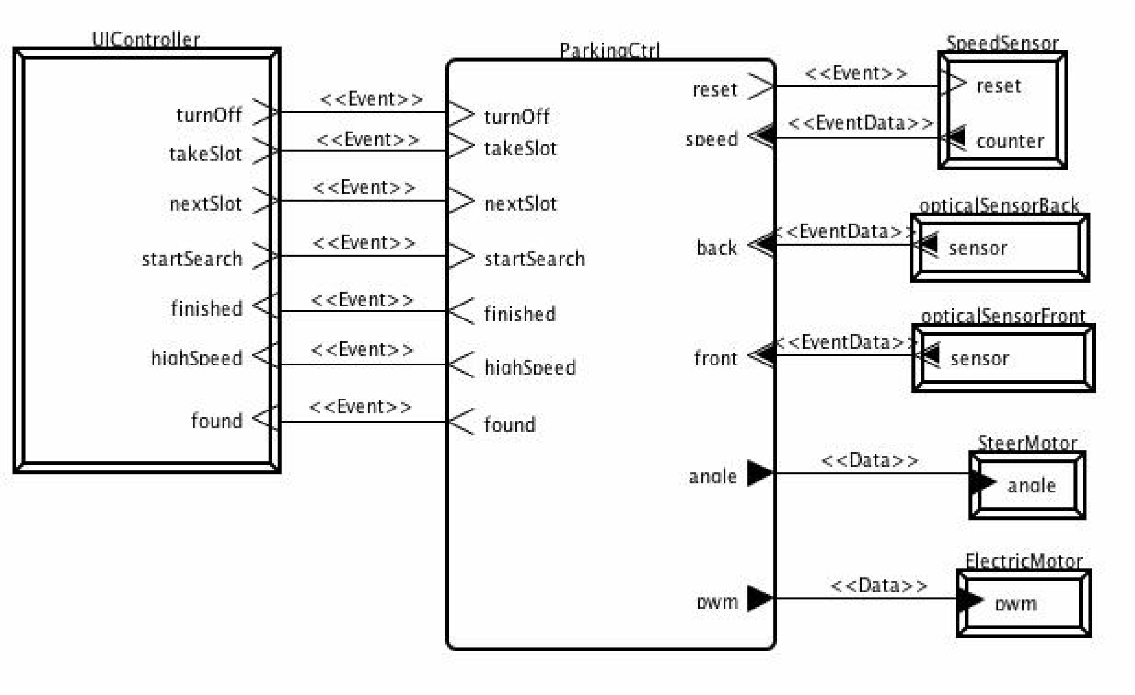

Figure 2. AP System Environment Description.

Following this approach, each component of the AADL model can derive into several subcomponents. By definition, the successive refinements will only finish as the model contains enough details to be proof correct or incorrect by the model verification. Each detailed model (i.e. iteration) should, however, cope with the abstract behavior defined for the higher level component.

4.4.1. Architecture Refinement

The architecture refinement process consists of successive model refinements and verification, as suggested in the design flow from Figure 4. It starts with identifying the operation modes (1) and threads (2) of the system, being followed by the mapping of functions to threads (3). Afterwards the designer can make the connections among the threads (4) and associate an execution mode to each thread (5).

We suggest organizing the functionalities of the system using different operation modes. This can be seen as a kind of temporal decomposition from the set of available functions. Therefore it is necessary to identify how many different operation modes the system should have. These modes can be used to guide the modeling of the distinct AADL processes that will be used to decompose the system in sub-parts. In our case study, the sub-functions of the first decomposition are more or less analogous to the operation modes. Figure 5 shows the automaton in charge of representing the AP system behavior.

After the identifications of the system (sub)functions it is possible to decompose the AADL model into different threads. This can be either the first level of decomposition of the AADL-system or a refinement of an existing thread. Defining connections means to establish the information exchange among the system subparts (threads). This also

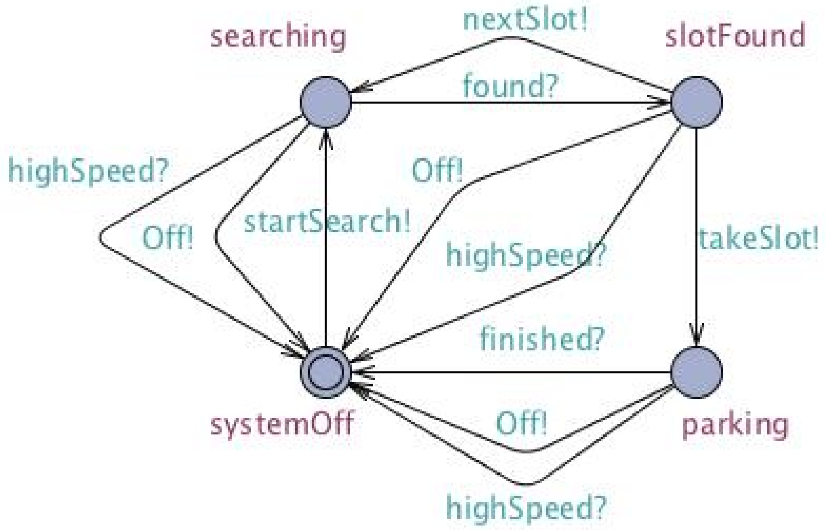

Figure 3. User interface behavior

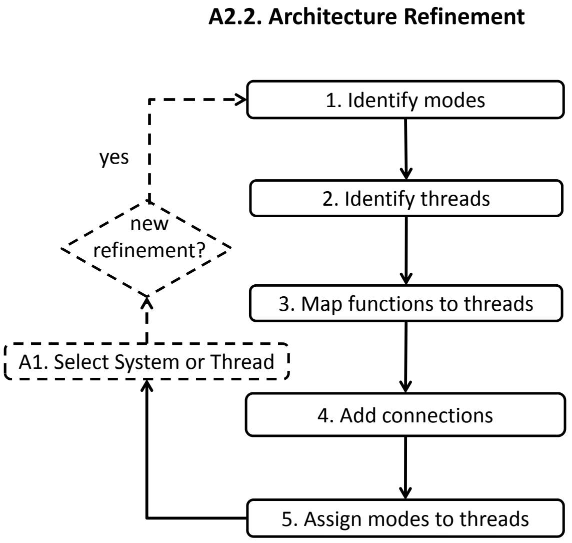
A2.2. Architecture Refinement
1. Identify modes
yes
2. Identify threads
new refinement?
3. Map functions to threads
A1. Select System or Thread
4. Add connections
5. Assign modes to threads

Figure 4. Refined steps from Architecture Refinement

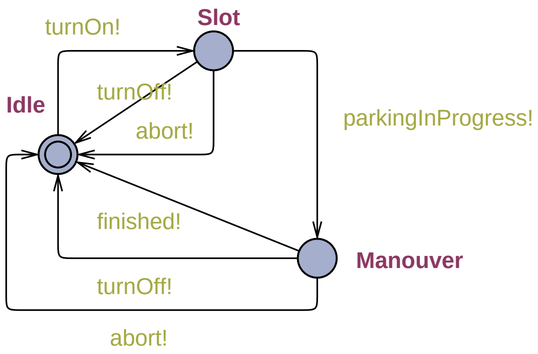

Figure 5. Basic operation modes of the AP System.

requires the definition of the data types associates with each port that transfer data.

For the AP system case study, the first level of decomposition consists basically in three threads, as shown in Figure 6. `SystemManagement` is used to start or halt the AP system by means of the graphical interface (FR1), `SlotSelection` is responsible to search for a parking slot (FR2), and finally `ParkingManeuver` is responsible to perform the parking (FR3). Every thread corresponds a FR of the system.

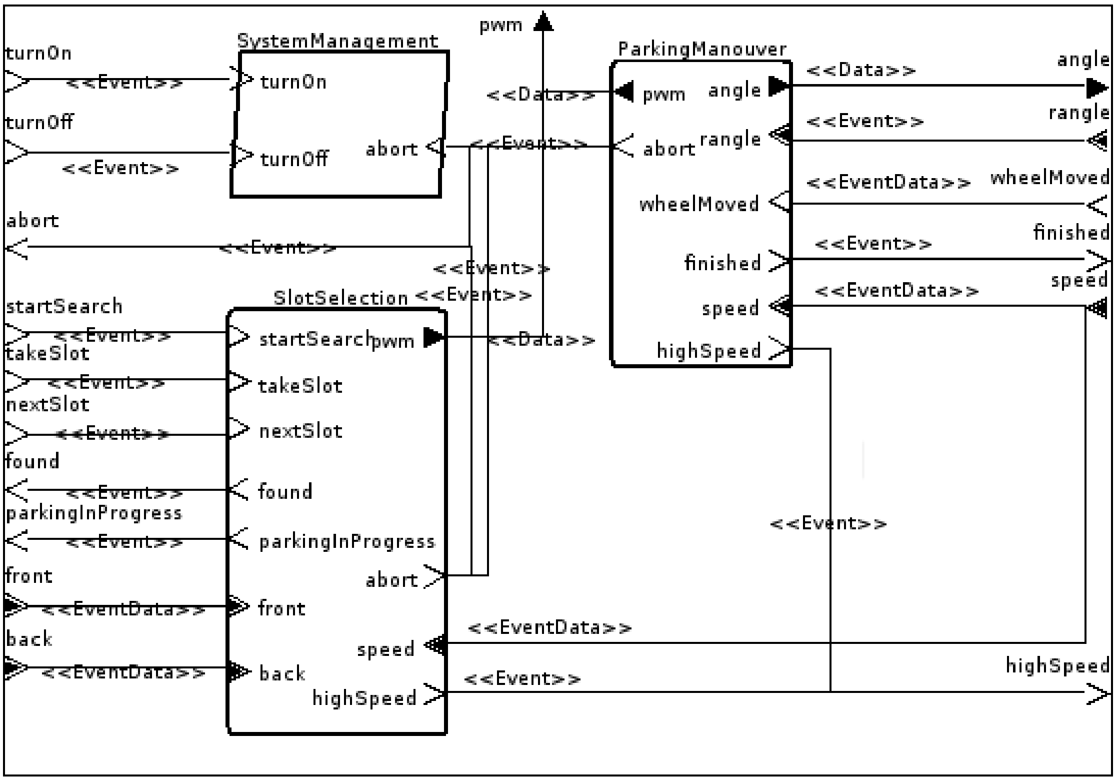

Figure 6. AADL model of parking control system (in the first decomposition)

Finally it is required to define in which operation modes each thread will be active. This represents a common modeling procedure to make the timing decomposition of the system functionalities. In AADL this is performed directly in the code, i.e. there is no graphical representation for this association. It must be highlighted, however, that it is possible to associate a thread with several operation modes.

4.4.2. Model Verification

It is a modeler decision whether he wants to perform further refinements or to verify the behavior of the current model. In order to make the model verification it is necessary to provide the abstract behavior of each thread that belongs to the AADL model. Afterwards designer should define the set of properties of interest to be verified and perform the verification process. Such process is detailed in the section 5.

4.5. Time-Related Levels

To verify the real-time properties of the model it is necessary to make the Software/Hardware Mapping (step-5). After this step, every thread must be associated with a specific processor. The hardware architecture must have at least one processor. Thereby, in the Real-Time Properties Refinement (step-6), the designer can add additional timing information in the AADL model to be further verified. Such information must be obtained using, for example, model simulation on top of the target architecture. Thereby it is possible to obtain the worst case execution time (WCET) for each function of the system prior to its implementation. The last step of the proposed process is in charge of making the verification of the timing properties. Schedulability and response-time analysis are exemples of possible properties to be verified.

5. Verification Process

It is possible to argue that our proposed verification process supports the safe design of the system's architecture using MDE's principles. By safe design we mean that the resulting system architecture goes through several verification steps in order to assure its correctness. To reach this goal it is performed a sequence of model transformations, which starts with an AADL-like model and finishes with an equivalent automaton model that is suitable for verification.

The verification process we have been working on uses AADL models as input and performs the model checking of LTL properties. Moreover, schedulability and buffer overflow can also be analyzed, as well as user defined properties. This process is split in the following phases (Figure 7):

- Use of the OSATE-TOPCASED [Team 2004, Topcased] environment for AADL model edition and XMI generation. We consider AADL together with its behavioral annex.
- Translation of AADL XMI models to Fiacre [Berthomieu et al. 2008].
- Translation of Fiacre to the timed transition system (TTS) input format of Tina toolbox.
- Translation to an untimed automaton via an LTL-preserving time abstraction.
- Verification of LTL properties using the Selt tool from the Tina toolbox.

5.1. Verification Tools

TINA is a software environment to edit and analyze Petri nets, Time Petri nets, Time Transition Systems, and also extension of these nets handling data, priorities and temporal

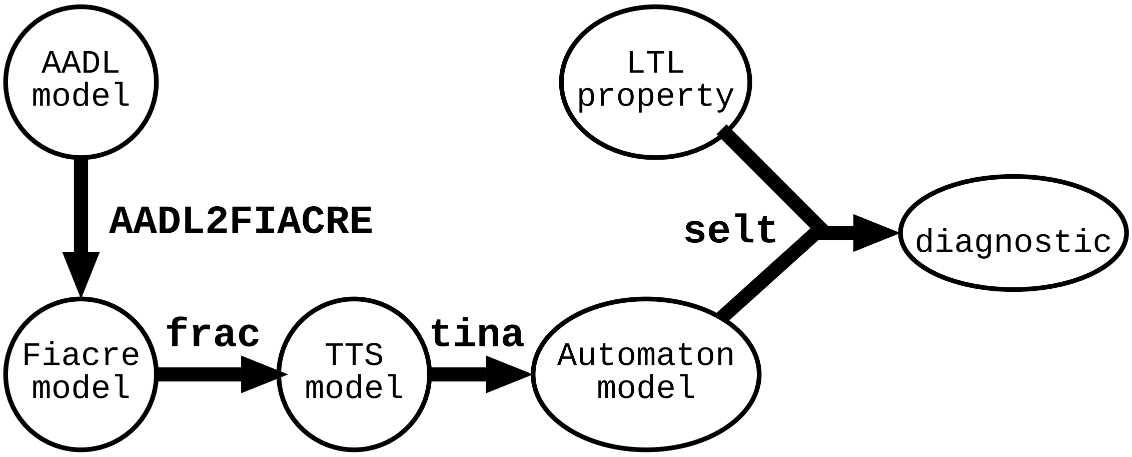

Figure 7. The verification process.

preemption. Beside the usual editing and analysis facilities of similar environments, the essential components of the toolbox are a state space abstraction tool (also called Tina) and a model checking tool (selt). Detailed information about the toolbox capabilities can be found in [Berthomieu et al. 2004].

TINA offers abstract state space constructions that preserve specific classes of properties of the state spaces of nets, like absence of deadlocks, linear time temporal properties, or bisimilarity. For untimed systems, abstract state spaces help to prevent combinatorial explosion. For timed systems, TINA provides various abstractions based on state classes, preserving reachability properties, linear properties or branching properties.

State space abstractions are provided in various formats suitable for existing model checkers. The TINA toolbox also provides a native model checker, selt. Selt allows one to check more specific properties than the general ones (boundedness, deadlocks, liveness) already checked by the state space generation tool. Selt implements an extension of linear time temporal logic known as State/Event LTL [Edmund et al. 2004], a logic supporting both state and transition properties. The modeling framework consists of Kripke transition systems (labeled Kripke structures, the state class graph in our case), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions.

State/Event-LTL formulas are interpreted over the computation paths of the model. They may express a wide range of state and/or transition properties. A formula p, q evaluates to true if it does so on all computation paths, constituted from the statements X (in the next step), G (globally), and F (eventually). Follows some typical formulas:

p	p holds at the start
X p	p holds at the next step (next)
G p	p holds all along the path (globally)
F p	p holds in a future step (eventually)
p U q	p holds until q holds (until) and q holds eventually.

Real-time properties, like those expressed in so called “timed temporal logics”, are checked using the standard technique of observers, encoding such properties into reachability properties. The technique is applicable to a large class of real-time properties and can be used to analyze most of the “timeliness” requirements found in practice.

5.2. Properties Verification

Currently, we support the verification of three kinds of properties: (i) implicit properties taken into account by the translator and leading to deadlock when not satisfied; (ii) user properties specified through AADL real-time observers; and (iii) properties specified directly in linear temporal logic.

5.2.1. Implicit properties

For the moment, two implicit properties are taken into account by the translator:

- **Schedulability**: threads are scheduled using a fixed priority protocol with user-specified preemption points. Deadline events are generated by the translator. If a deadline occurs while a thread is still active, a specific deadlock is generated.
- **Buffer overflows**: AADL defines the property *Overflow_Handling_Protocol* which specifies what to do in case of overflow. Either the oldest or the newest data is lost, or the component is erroneous. The latest case is handled by the translator to generate a specific deadlock if the capacity of the input buffer is exceeded.

5.2.2. Real-time observers

Some properties such as bounded response time can be expressed using AADL threads acting as real-time observers. The component to be checked is linked to an observer which plays the role of its environment and checks its responses.

For example, properties of the `maneuver` component of the parking can be verified by specifying an environment as the following. It checks that the `highSpeed` signal is emitted one period (fixed here at 10ms) after the speed becomes non zero. Otherwise, the `err` state would be reached. It also checks that the `abort` signal is sent if the wheels are moved. The `selt` model checker was used to show that the `err` state is unreachable.

5.2.3. Linear time Temporal Logic

Temporal properties can be checked on the closed system. They can be expressed in linear temporal logic (LTL) and passed to the `selt` tool. Atomic properties are either event properties or state properties. For example:

- If the speed is too high, the interface cannot get the `found` message while the search has not been restarted.
- It is possible to park the car, i.e. there exists an execution path leading to a state where the car is parked. It is expressed as a negated property: it is not true that in any execution, `finished` is never sent.
- The car can be parked infinitely often.

5.2.4. Modal mu-calculus

There exists some useful properties that cannot be expressed neither in LTL, nor in CTL. For example, the fact that the user interface can be reinitializable by the user whatever the system does. To solve this problem, it can be expressed in modal mu-calculus. Such a property can be verified on atemporal models by the `muse` tool of the Tina toolbox. It must be associated with a *stability* property expressing that non-user events do not leave the initial state.

6. Conclusions

In this paper we presented a verification approach and the related toolset to design safety critical systems using the AADL language. This work is part of a more general project, which also covers the hardware architecture definition in more details, going towards producing safe models for critical applications. It must be highlighted that in the end of the process it is possible to make automatic code generation from the AADL model for a given platform.

It should be noticed, however, that given the complexity of the situation, the guarantee of the existence of a correct solution cannot be asserted. This also applies to the implementation derived from the generated model. To overcome this problem, designer feedbacks are necessary and, more generally, it should be wise to superpose to the software engineering process risk management.

Future work should cover automatic derivation of the properties to be verified from the system requirements. By using such approach, we should also assess the property languages in more details.

Acknowlegements

This work was developed with the grant CAPES STIC-AmSud 003/07 **TAPIOCA** : *Timing Analysis and Program Implementation On Complex Architectures* and supported by the French AESE project **Topcased**.

References

Berthomieu, B., Bodeveix, J.-P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., and Vernadat, F. (2008). Fiacre: an intermediate language for model verification in the TOPCASED environment. *Proceedings of the 4th European Congress on Embedded Real-Time Software ERTS'08(Toulouse, France).*

Berthomieu, B., Peres, F., and Vernadat, F. (2007). Model checking bounded prioritized time petri nets. In Namjoshi, K. S., Yoneda, T., Higashino, T., and Okamura, Y., editors, *ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 523–532. Springer.

Berthomieu, B., Ribet, P., and Vernadat, F. (2004). The tool TINA – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14).

Dissaux, P. and Singhoff, F. (2008). Stood and cheddar: Aadl as a pivot language for analysing performances of real time architectures. In *4th European Congress ERTS EMBEDDED REAL TIME SOFTWARE.*

Edmund, S. C., Clarke, E. M., Sharygina, N., and Sinha, N. (2004). State/event-based software model checking. In *In Integrated Formal Methods*, pages 128–147. Springer-Verlag.

Feiler, P., Gluch, D., and Hudak, J. (2006). The architecture analysis & design language (AADL): An introduction. Technical report, Software Engineering Institute, Carnegie Mellon University.

Franca, R. B., Bodeveix, J.-P., Filali, M., Rolland, J.-F., Chemouil, D., and Thomas, D. (2007). The AADL behaviour annex – experiments and roadmap. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 377–382, Washington, DC, USA. IEEE Computer Society.

Håkansson, J., Carlson, J., Monot, A., Pettersson, P., and Slutej, D. (2008). Component-based design and analysis of embedded systems with uppaal port. In *ATVA '08: Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 252–257, Berlin, Heidelberg. Springer-Verlag.

Schmidt, D. (2006). Model-driven engineering. *IEEE Computer*, 39(2).

Team, S. A. (2004). OSATE: An extensible source aadl tool environment. Technical report, Software Engineering Institute, Carnegie Mellon University.

Topcased. (toolkit in open-source for critical apllications and systems development). `http://www.topcased.org`.