2010 15th IEEE International Conference on Engineering of Complex Computer Systems

Supporting the Design of Safety Critical Systems Using AADL

T. Correa, L. B. Becker, J.-M. Farines
Federal University of Santa Catarina
Dept of Automation and Control Systems
Florianopolis, Brazil
{tiagotb,lbecker,farines}@das.ufsc.br

J.-P. Bodeveix, M. Filali
IRIT-CNRS
Université de Toulouse
Toulouse, France
{bodeveix,filali}@irit.fr

F. Vernadat
LAAS-CNRS
Université de Toulouse
Toulouse, France
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. A sequence of model transformations facilitates 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.

I. Introduction

The Architecture Analysis & Design Language (AADL) [6] is a textual and graphical language used to design and analyze the software and hardware architecture of safety critical real-time systems. AADL is used to describe the structure of systems as an assembly of software components mapped onto an execution platform. It is used to describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as timing). In order to support model analysis, AADL relies on a precise execution model. AADL is by now a standard [9]; The version 2 has been recently voted.

The goal of this paper is to present an approach that supports model checking over AADL models. This is possible by means of a sequence of model transformations, which finishes when the model is suitable for verification, as further discussed along the paper. Using the proposed approach we expect to enhance considerably the reliability of AADL models designed for safety-critical applications.

The remainder parts of the paper are structured as follows: Section II discusses some related methodologies and tool support. Sections III and IV detail the proposed verification approach and its application in a case study. Section V presents our conclusions and directs our future work.

II. 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) [10] 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 [4] 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 [7]. The tools Uppaal Port [8] and Pola[2] 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.

III. The Proposed Approach

Our proposed design-process for critical embedded systems supports the safe design of the system's architecture using MDE's principles. By safe design we mean that the resulting

978-0-7695-4015-3/10 $26.00 © 2010 IEEE
DOI 10.1109/ICECCS.2010.52

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.

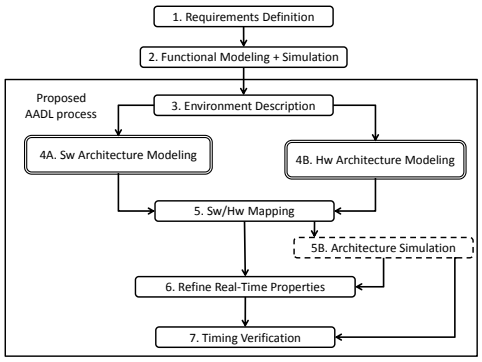

Fig. 1. Proposed Design Flow

The process starts in level-1 with the definition of the functional and non-functional requirements of the system, resulting in a textual set of requirements. It is followed by the design of a functional model for the system (e.g. Lustre or Simulink model). In level-3 it begins the design of the AADL model, providing the specification of the external devices (environment) that interact with the system. Level-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 level-5 a mapping from the modeled software components to the hardware model is performed. The result is a complete AADL model. In level-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 level-5B. The proposed development process is concluded in level-7 with the final model verification, which uses as input the AADL model updated with the precise timing information.

It is important to highlight that the design flow among the levels 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 (level-7), then the designer should be able to judge if the problem is due to the result of level-4A (proposed software architecture) or to the result of level-4B (target hardware architecture).

In this paper we concentrate the discussions on the software architecture modeling and in the verification chain (level-4A). The target hardware architecture definition (level-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. Given that the verification chain is detailed in the next section, the reminder parts of the current section details every level depicted in Figure 1. We use an Autonomous Parking (AP) System case study to elucidate the work performed in each level.

A. 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 III-A 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) parking the car. NFRs are like properties that must be satisfied by the related FR. For example, NFR2.2 states that if the speed is too high (over 20km/h), than it is not possible to search for a parking slot.

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

TABLE I
REQUIREMENTS SET OF THE AUTONOMOUS PARKING (AP) SYSTEM

B. 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.

C. Environment Description

The third level of the proposed process consists of using AADL to describe the environment that interacts with the system under development. In other words, it is necessary to define the set of interactions of the system with the external devices, such as sensors, actuators, user interface, etc.

For this reason it is suggested here the use of 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 is that it allows detailing each message exchanged between the system and the devices, including information like data type, arrival pattern, and time constraints.

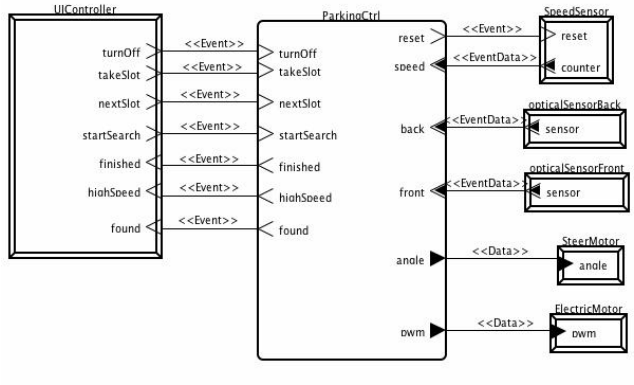

Fig. 2. AP System Environment Description.

In this phase it is assumed that two different kind 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.

D. Software Architecture Modeling

The software architecture modeling (level-4A) is probably the most important phase of the proposed design process. This phase may have several steps of iterations, given the fact that the designer may create several AADL models, from more abstract to more detailed ones. Moreover, each step should have its properties verified before the designer proceeds with detailing the AADL model.

In the first iteration the designer must detail the AADL system process (e.g. `ParkingCtrl` at Figure 2) into a set of subcomponents (that can be 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), a new refinement in each component should take action, starting new iterations.

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. Follows a more detailed discussion about the main steps of this phase, namely `Architecture Refinement` and `Model Verification`.

1) Architecture Refinement: The architecture refinement process consists of successive model refinements and verification, as suggested in the design flow from Figure 3. 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). The reminder of this section details these steps and presents its application on the AP system.

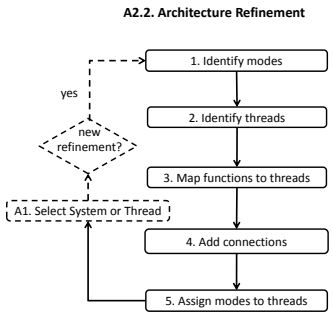

Fig. 3. Refined steps from Architecture Refinement

We suggest organizing the functionalities of the system using different operation modes. This can be seen as a kind of temporal decomposition of the set of available functions. Therefore it is necessary to identify how many different 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.

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 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 4.

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).

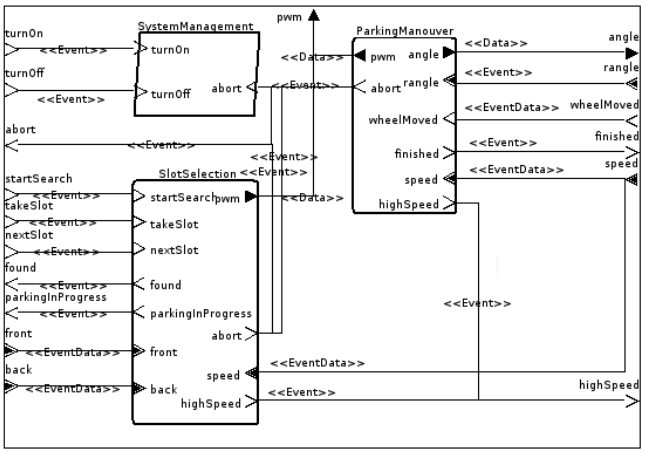

Fig. 4. AADL model of parking control system (in the first decomposition)

Once we have both functions and threads defined it is necessary to relate them, i.e. define which functions belong to each thread. Here, information like periodicity and deadlines of threads and functions can be defined. The result of this mapping in the AP system is shown in Figure 4. As it can be observed, in this level every thread is responsible for one FR of the system.

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.

2) Model Verification: It is a modeler decision whether he wants to perform further refinements (as discussed in the previous subsection) 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 next section and, for the remainder of this section, we conclude the presentation of the proposed methodology.

E. Time-Related Levels

To verify the real-time properties of the model it is necessary to make the Software/Hardware Mapping (level-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 (level-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.

IV. 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 5):

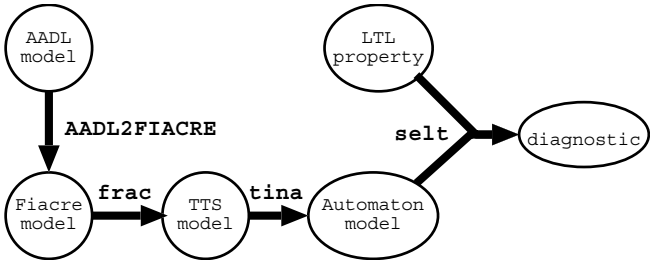

Fig. 5. The verification process.

- Use of the OSATE-TOPCASED [11], [12] environment for AADL model edition and XMI generation. We consider AADL together with its behavioral annex.
- Translation of AADL XMI models to Fiacre [1].
- 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.

A. 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 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 [3].

TINA offers various 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 [5], 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. Some typical formulas are the following (a formula evaluates to true if it does so on all computation paths, X, F, G and U are LTL modalities, p, q are 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.

We also use the weak until operator **W**. **p W q** holds until q holds. It is not mandatory that q eventually happens.

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.

B. 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.

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.

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 is used to show that the `err` state is unreachable.

```
thread implementation EnvironmentThread.imp
annex behavior_specification {**
states
 s0: initial complete state;
 s1,s2,err: complete state;
transitions
 s0 -[]-> s0 { speed!(0); rangle!(0); };
 s0 -[finished?]-> s0;
 s0 -[]-> s1 { speed!(10); };
 s0 -[]-> s2 { wheelMoved!; };
 s1 -[highSpeed?]-> s0;
    -- detected in less than the period
 s1 -[on highSpeed'count = 0]-> err;
 s2 -[abort?]-> s0;
 s2 -[on abort'count=0]-> err;
**};
end EnvironmentThread.imp;
```

Remark Response time information could be added to the AADL model as properties of flow specifications and thus be implicitly checked. However, this is not easy if response time is greater than the minimum period of the input signal. Here, our observer supposes that speed does not change while waiting for the `highSpeed` signal.

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.

  $\Box$ (`highSpeed` $\Rightarrow$ ($\neg$`found` **W** `startSearch`))

  This property is in fact not satisfied because taking into account the speed information and aborting the process needs one cycle. We use the hyperperiod event `H` to reformulate the property as follows: if the speed is too high, starting from the next hyperperiod signal, we cannot get the `found` message unless `startSearch` has been pushed.

  $\Box$`highSpeed` $\Rightarrow$ ($\neg$`H` **U** `H`$\wedge$($\neg$`found` **W** `startSearch`))

- 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.

$$\texttt{Parking} \not\models \Box\neg\texttt{finished}$$

- The car can be parked infinitely often. It is also expressed as a negated property:

$$\texttt{Parking} \not\models \Diamond\Box\neg\texttt{finished}$$

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 using the macro bellow, where $\mathcal{U}$ is the set of user events and φ the property to be reachable, i.e. the initial state. It defines the set of states from where φ is reachable by user events even if non user events are fired as a smallest fixed point (the `min` operator).

$$\text{reachable}(\mathcal{U}, \varphi) = \min\ X \mid \varphi \vee ([-\mathcal{U}]X \wedge \bigvee_{e \in \mathcal{U}} ([e]X \wedge \langle e \rangle X)$$

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.

It would also be possible to encode a possibly real-time winning strategy using the AADL behavior annex and check that the initial state is reachable using an LTL property over the generated abstract automaton. In our example, this is very simple because a user command can always be used.

V. 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.

Currently there is no automated process to transforms the requirements identified at a high level of abstraction and the final concrete properties to verify on the final formal model. This is currently under investigation in our group.

Finally, this study has made us aware of the fact that linear temporal logic although simple is not rich enough for expressing some required intuitive properties. In this paper, we have suggested the use of mu-calculus. We intend to study in future work suitable patterns to enhance the use of such a logic. Another further direction of this research would be providing a risk analysis to assist the design.

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

[1] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. 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)*, January 2008.

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

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

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

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

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

[7] R. B. Franca, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas. 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, 2007. IEEE Computer Society.

[8] J. Håkansson, J. Carlson, A. Monot, P. Pettersson, and D. Slutej. 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, 2008. Springer-Verlag.

[9] SAE. *Architecture Analysis & Design Language (AADL), AS-5506*. SAE International, 2004.

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

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

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