ACM/IEEE 17th International Conference on
Model Driven Engineering Languages and Systems

September 28 – October 3, 2014 Valencia (Spain)

ACVI 2014 – Architecture Centric Virtual Integration Workshop Proceedings

Julien Delange and Peter Feiler

Published on Sept 2014 v1.0

Organizers

Julien Delange (co-chair)	Carnegie Mellon Software Engineering Institute
Peter Feiler (co-chair)	Carnegie Mellon Software Engineering Institute

Program Committee

Canals Agusti	C-S
Etienne Borde	TELECOM ParisTech
Matteo Bordin	Adacore
Jörgen Hansson	University of Skövde, Sweden
Jerome Hugues	ISAE
Emilio Insfran	Universitat Politcnica de Valncia
Akihito Iwai	DENSO Corporation
Alexey Khoroshilov	ISPRAS
Bruce Lewis	US ARMY
Oleg Sokolsky	University of Pennsylvania
Jean-Pierre Talpin	INRIA
Steve Vestal	Adventium Labs
Bechir Zalila	National School of Engineers of Sfax

Table of Contents

Preface

New real-time systems have increasingly complex architectures because of the intricacy of the multiple interdependent features they have to manage. They must meet new requirements of reusability, interoperability, flexibility and portability. These new dimensions favor the use of an architecture description language that offers a global vision of the system, and which is particularly suitable for handling real-time characteristics. Due to the even more increased complexity of distributed, real-time and embedded systems (DRE), the need for a model-driven approach is more obvious in this domain than in monolithic RT systems. The purpose of this workshop is to provide an opportunity to gather researchers and industrial practitioners to survey existing efforts related to behavior modelling and model-based analysis of DRE systems.

Cyber-Physical systems (CPS) combine many challenges to meet requirements for reusability, interoperability, flexibility or dependability. The use of architecture description language helps to integrate components before implementing the system. Such integration approach eases system design analysis and implementation, detects design errors and potential defects before development efforts, avoiding re-engineering costs and making the system more robust and safe. This first edition of this workshop seeks contributions from researchers and practitioners interested in architecture-centric methods and their use to design and analyze systems. The conference topics of interest are:

- Modeling Notations: new languages, inter-operability between languages
- Architecture Centric Analysis Tools
- Virtual Integration Process and Tools
- Definition of extensions for the design of specific systems (e.g. avionics) or support of a particular analysis (e.g. safety)
- Automatic Code Generation from Models
- Model Transformation
- Model Analysis Methods
- Support of Certification (e.g. DO178C) using Models
- Industrial experiences of use of Model-Based technologies

The Architecture-Centric Virtual Integration Workshop is colocated with the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems.

September 2014 Julien Delange and Peter Feiler

Keynote

The Story of AADL

Peter Feiler

Carnegie Mellon Software Engineering Institute

5 years ago the SAE AS-2C subcommittee started to work on the Architecture Analysis & Design Language (AADL) standard. AADL was targeted to address issues in mission and safety critical software-reliant systems, aka. Cyber-physical systems. AADL addresses the increasing challenges of such systems - the exponential increase in verification related software rework cost. Industry studies show that 70% of defects are introduced in requirements and architecture design, while 80% are discovered post-unit test. After a short history and summary of the challenges, the presentation highlights the expressive, analytical, and auto-generation capabilities of the AADL core language as well as several of its standardized extensions to address multiple quality dimensions and do so incrementally at different levels of fidelity. The presentation then illustrates these capabilities on several realistic industrial examples. The presentation concludes by outlining a four part improvement strategy: architecture-led requirement specification to improve the quality of requirements, architecture refinement and incremental virtual system integration to discover issues early, compositional verification through static analysis to address scalability, and incremental verification and testing throughout the life cycle as assurance evidence.

Peter Feiler *is a 29 year veteran and Principal Researcher of the Architecture Practice (AP) initiative at the Software Engineering Institute (SEI). His current research interest is in improving the quality of safety-critical software-reliant systems through architecture-centric virtual system integration and incremental life cycle assurance to reduce rework and qualification costs. Peter Feiler has been the technical lead and main author of the SAE Architecture Analysis & Design Language (AADL) standard. He has a Ph.D. in Computer Science from Carnegie Mellon.*

DM-0001610

Contract-based specification and analysis of AADL models*

Ernesto Posse Juergen Dingel
{eposse,dingel}@cs.queensu.ca

School of Computing – Queen's University
Kingston, Ontario, Canada

Abstract. We describe an approach to the specification, analysis and verification of AADL models using assume/guarantee behavioural contracts specified with the Property Specification Language (PSL). This approach aids the development process by 1) supporting the reuse and replacement of components based on their contracts rather than only their interface or their implementation and thus reducing the need for re-engineering; 2) providing early discovery of behavioural inconsistencies that may pose problems with integration; and 3) allowing an incremental and flexible application of specification and verification instead of requiring an all-or-nothing approach. It also helps improving the product itself by detecting safety and liveness problems via model-checking. We also briefly discuss a prototype plug-in for OSATE supporting an annex language which we call AGCL.

1 Introduction

The development of distributed, real-time embedded systems (DRE) presents multiple challenges born out of their inherent complexity. In order to address the complexity of these systems and their design, component-based and model-driven approaches are often used. Such approaches often rely on modelling and architecture description languages such as the Architecture Analysis and Design Language, AADL [6], which provides the means to describe systems in terms of interacting components and their composition.

Complex patterns of interaction between components pose a challenge to developers, making it difficult to understand how a system behaves and whether it satisfies its requirements and behaves correctly, *e.g.*, satisfying safety and liveness constraints. In order to provide some relief to the developer, automatic formal verification techniques such as model-checking can help to analyze a system's behaviour. Nevertheless, formal verification often faces the so-called *state-explosion problem*, whereby adding a component multiplies the number of states in a system, resulting in an exponential growth in the state-space which provides a challenge to verification techniques and tools.

* This work was financed in part by Edgewater Computer Systems Inc., Ontario Centres of Excellence and Connect Canada.

An approach to deal with the state-explosion problem is the use of *compositional analysis* which leverage the structure of the system. In these techniques, the analysis of a composite system is reduced to the analysis of its parts. A main advantage of such techniques is that if a single component changes, there is no need to reanalyze the whole system, only the portion directly affected. This provides the basis for *incremental analysis*, which aids development by focusing verification only on the components on which the developer is working.

A well-known compositional approach is based on *assume/guarantee contracts* where each component is annotated with a *contract* consisting of an *assumption* specifying how the component expects its environment to behave, and a *guarantee* specifying the behaviour guaranteed by the component if the assumptions hold. Contract-based specification facilitates integration not only by making expectations and assurances explicit, but also by ensuring preservation of correctness when a component with a given contract is replaced by another component whose contract conforms to or refines the first. Contract-based analysis uses the contracts to automatically establish whether a composition of components satisfy the contract of the composite component to which they belong.

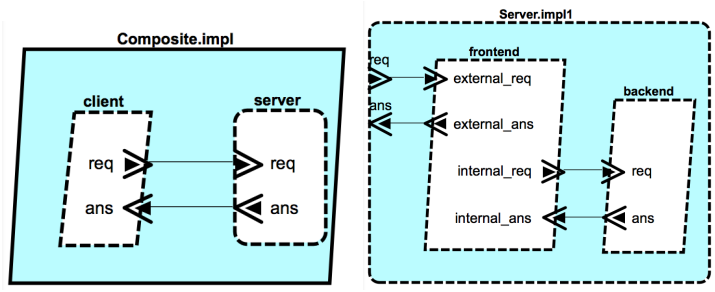
Most approaches to assume/guarantee analysis (*e.g.*, [3]) limit the scope of assumptions to component inputs and guarantees to component outputs. Furthermore, in many approaches the form of a contract is of the form "assuming these inputs, we guarantee these outputs". These are two big limitations. Assumptions and guarantees are supposed to capture *behaviour*, not just individual inputs and outputs. Furthermore, both assumptions and guarantees should describe "conversations" between a component and its environment, with assertions about information flowing both ways. For example, a component may assume that whenever it sends a particular output to its environment, the environment will send back some particular message as input to the component.

In this paper we address these shortcomings by we proposing an AADL annex sub-language for annotating components with assume/guarantee contracts and a prototype verifier that performs compositional analysis. In this sublanguage we use the Property Specification Language, PSL, an IEEE Standard [4] which allows the specification of behaviour combining the expressive power of ω-regular expressions and linear temporal logic (LTL), and in which both assumptions and guarantees can refer to inputs and outputs.

Another shortcoming of many formal approaches to analysis is that they usually require an all-or-nothing commitment on the part of the developer, for example requiring a full, formal account of all components' behaviours. We address this by supporting the notion of *viewpoints*. A viewpoint represents a particular set of requirements distributed across components. The designer may annotate any given component with several contracts. All contracts sharing the same name across different components form a *viewpoint*. For example, the designer can define some safety viewpoints separately from some liveness viewpoints. This allows the developer to add contracts and viewpoints as the design progresses. This notion of viewpoint is simpler than that found in [2] where the developer is required to explicitly use more complex operators to combine contracts.

2 AGCL: a sublanguage for assume/guarantee contracts

Consider the model shown in Figure 1 depicting a system consisting of a client and a server which itself consists of a front-end or mediator, and a back-end. In this common pattern, the client may issue requests via a channel `req` and expects an answer on channel `ans`. The front-end of the server receives these requests, may perform some preprocessing, and delegates the requests to the back-end server via the `internal_req` channel. When the back-end server responds on the `internal_ans` channel, the front-end may do some post-processing, and deliver the final answer to the client.



(a) composite (b) server

Fig. 1. A simple client-server architecture with a mediator process.

To annotate components with contracts we need to declare viewpoints, which is done at the package-level annex library as shown below (using the `viewpoint` keyword). The `enforce` keyword is used to inform the tool which viewpoints should be analyzed.[1]

```
1 package client_server_mediator
2 public

3 annex AGCL {**
4     viewpoint normal_operation;
5     viewpoint alternative_operation;
6     enforce normal_operation;
7 **};

8 -- etc.

9 end client_server_mediator;
```

[1] To keep the presentation of our example simple we show only the annex for each classifier. We also ommit the specification of the top-level process, the client and focus on the server only, and we ommit the thread type declarations with ports which are visible in Figure 1.

2.1 Contracts for atomic components (threads)

Figure 2 shows the backend server. Its annex has a `behaviour` clause describing the behaviour of the actual implementation, and a `contract` clause defining a contract for this component within the `normal_operation` viewpoint. The behaviour states that whenever the server receives a request (an `in` event on the `req` port with some signal `s1`), then it will produce an output on the `ans` port in the next state or cycle. The contract in this case has no assumptions and therefore it is simply `true`. The guarantee is that whenever the backend receives a request, it will eventually produce an answer. In this case, it should be fairly trivial that the beahviour satisfies the contract.

```
thread implementation BackendServer.impl1
annex AGCL {**
    behaviour always (in req:s1 -> next out ans:s2);
    contract normal_operation
        assumption TRUE;
        guarantee always (in req:s1 -> eventually out ans:s2);
    end normal_operation;
**};
end BackendServer.impl1;
```

Fig. 2. Backend server.

Note that the guarantee can talk about both inputs and outputs. The same is true for assumptions. A guarantee represents an obligation on the component, whereas an assumption represents an obligation on its environment. Hence, when a guarantee states an atomic proposition labeled `in`, it is stating the component's obligation to accept or receive an input. When a `in` atomic proposition appears in an assumption, the input direction is stated from the point of view of the component but it actually represents an output obligation from the component's environment to the component. Similarly, an `out` in a guarantee is an obligation for the component to produce output, whereas an `out` in an assumption, while stated from the point of view of the component, actually represents an obligation on the environment to accept or receive input coming from the component.

Figure 3 shows the frontend. Its behaviour clause specifies that whenever an external request arrives (from the client), eventually it will reach a state where it will send a request to the backend (through the `internal_req` port) and from that point onwards, whenever it receives an answer from the backend, it will eventually forward the answer to the client on the `external_ans` port. The contract clause specifies as assumption that whenever it sends a request to the backend server, it will get an answer from it eventually. The guarantee states that whenever it receives an external request from the client, it will eventually send an internal request to the backend, and whenever it gets a response from the backend it will eventually send an answer back to the client. In this case it

is less trivial that the behaviour satisfies the contract, but this follows from the formal semantics of PSL.

In general, for threads, a behaviour B satisfies a contract $C = (A, G)$ with assumption A and guarantee G, if the formula $B \wedge A \Rightarrow G$ is valid. Intuitively, the behaviour and the assumptions must be enough to imply the guarantee. A (linear) temporal logic formula (including PSL) is *valid* if it holds in all possible paths for every possible model. In our case, the premise of this implication captures the model: the guarantee will be required to be true only on those models with behaviour B, if the assumption A is true as well. The validity of PSL formulas can be established with a model-checker (see Section 4).

```
thread implementation Frontend.impl1
annex AGCL {**
    behaviour always (in external_req:s1
                -> eventually (out internal_req:s1
                    & always (in internal_ans:s2
                        -> eventually out external_ans:s2)));
    contract normal_operation
        assumption always (out internal_req:s1
                        -> eventually in internal_ans:s2);
        guarantee always (in external_req:s1
                        -> eventually out internal_req:s1)
                & always (in internal_ans:s2
                        -> eventually out external_ans:s2);
    end normal_operation;
**};
end Frontend.impl1;
```

Fig. 3. Server frontend (mediator).

An AGCL annex can contain multiple contracts, which can be verified independently. This allows the developer to add contracts as the design progresses, and define contracts which focus only on particular aspects of interest.

2.2 Contracts for composite components (thread groups)

Figure 4 shows the server combining frontend and backend. In this case, the thread group does not have a behaviour specification, but only a contract. It's contract doesn't make any assumptions, but it states the guarantee that whenever an external request comes from the client, eventually it will answer it.

The problem in this case is the following: if we already know that the subcomponents satisfy their respective contracts, how do we establish if the composition (the `Server.impl1`) satisfies its contract? This can be established as follows: let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts for the two subcomponents K_1 and K_2 of a composite component K with contract $C = (A, G)$. Assuming that K_1 satisfies C_1, and K_2 satisfies C_2, then K satisfies C if the following two PSL formulas are valid:

1. $G' \Rightarrow G$ where $G' \stackrel{def}{=} G_1 \wedge G_2$, and
2. $A \Rightarrow A'$ where $A' \stackrel{def}{=} (G_2 \Rightarrow A_1) \wedge (G_1 \Rightarrow A_2)$

Intuitively the first one states that the guarantees of the subcomponents together must imply the guarantee of the composite. The second one states that the assumption of the composite must be enough to ensure that 1) the guarantee of the second must imply the assumption of the first, and 2) the guarantee of the first component implies the assumption of the second. This is because the subcomponents may be connected and information may flow both ways between them, and they are part of each other's environments: the behaviour of K_1's environment is given by K_2's guarantees G_2 together with K's environment given by A. Hence, A and G_2 must imply A_1. Similarly for K_2. To be precise, there is a little processing that needs to be done on the formulas G_i and A_i, namely we need to replace port references ocurring in atomic propositions by connector references so that they refer to the same entity, and we need to flip the direction (in/out) of those atomic propositions in assumptions for the same reason. For composite components with n subcomponents, the formulas are generalized to $G' \stackrel{def}{=} G_1 \wedge G_2 \wedge \cdots \wedge G_n$ and $A' \stackrel{def}{=} \wedge_{i=1}^{n}((\wedge_{j \neq i} G_j) \Rightarrow A_i)$ respectively. In other words, the guarantees of all subcomponents must imply the guarantee of the composition, and the assumption of each subcomponent must be implied by the guarantees of all other subcomponents. This later requirement can be relaxed in that it is only needed that the assumption of each subcomponent must be implied by the guarantees of only those subcomponents connected to it.

In our example, K_1 and K_2 are `Backend.impl1` and `Frontend.impl1`, and K is `Server.impl1`. As before, we establish the validity of the formulas above with a model-checker (see Section 4), and in this case they happen to be true.

```
thread group implementation Server.impl1
subcomponents
    backend : thread BackendServer.impl1;
    frontend : thread Frontend.impl1;
connections
    client_req : port req -> frontend.external_req;
    client_ans : port frontend.external_ans -> ans;
    server_req : port frontend.internal_req -> backend.req;
    server_ans : port backend.ans -> frontend.internal_ans;
annex AGCL {**
    contract normal_operation
        assumption TRUE;
        guarantee always (in external_req:s1
                    -> eventually out external_ans:s2);
    end normal_operation;
**};
end Server.impl1;
```

Fig. 4. The server.

Incremental analysis is supported in the following way: if one component changes its behaviour, for example the frontend, we only need to check whether this behaviour satisfies its contract(s). If the result of this analysis is positive, then there is no need to check other components, or the validity of the composite formulas, as the contract has not changed and therefore the validity of formulas 1 and 2 is preserved. If the result of this analysis fails, then the developer needs to either modify the behaviour or the contract for the component in question. If the contract for a component changes then one must re-analyze that component (recursively if it is a composite component) and then re-evaluate the implications $G' \Rightarrow G$ and $A \Rightarrow A'$ as above, but there is no need to re-analyze components which have not changed or whose contract has not changed, as they would not change the validity of these formulas.

2.3 Conformance

Contracts can annotate not only implementations but also types. This opens a set of closely related problems that need be addressed. The first one is this: if we have a component implementation K of type T and K has a contract $C_K = (A_K, G_K)$ and T is annotated with contract $C_T = (A_T, G_T)$, how do we know that C_K conforms to C_T? This can be answered by checking two implications: $G_K \Rightarrow G_T$ and $A_T \Rightarrow A_K$. Note that the implication is covariant on guarantees and contravariant on assumptions. For guarantees, this is because the guarantee of the type must be a guarantee of any of its implementations: the set of possible observable behaviours described in G_K must be a subset if the set of behaviours defined by G_T, otherwise there would be at least one behaviour guaranteed by the implementation which does not conform to what the type prescribes. For assumptions the direction is contravariant because the set of behaviours specified by A_T must be a subset of the set of behaviours specified by A_K. If this wasn't required, there would be at least one environment behaviour acceptable by A_T but not by A_K which would entail that component K would not be able to be placed in some composite components expecting type T.

The other related problems occur when an implementation extends another implementation or a type extends a type and both have contracts in the same viewpoint. These cases can be handled as the above: if K' (or T') has contract $C' = (A', G')$ and it extends K (resp. T) with contract $C = (A, G)$, then conformance can be established by checking the validity of $G' \Rightarrow G$ and $A \Rightarrow A'$.

3 Relation between PSL sequences and AADL behaviours

A key issue in the use of a specification language or temporal logic such as PSL to describe behaviours and contracts of AADL models is the correspondance between the semantics of PSL expressions and the behaviour of the AADL model which they intend to describe. However, there is a fundamental obstacle: the core AADL standard doesn't define a unique way of specifying behaviour. It is up to annexes or external languages to provide the implementation of a component and

therefore it is not possible to define a general correspondance, but only consider specific types of implementation. One such possibility is to use the behaviour annex where the implementation is defined as a kind of (hierarchical) state machine. In this paper we do not assume any particular formalism, annex or type of implementation. Nevertheless, if behaviour is specified with the behaviour annex or a similar state-based formalism, we can infer the PSL behaviour specification from such state machine using standard transformations (*e.g.*, automata to regular expression, [7]) and then apply the analysis algorithms as described. Alternatively, we could use the behaviour clause itself to infer an automaton that implements it, using well-known algorithms that can transform such expressions and formulas into automata (*e.g.*, [7,8]).

Another way of relating the PSL specifications with the behaviour of AADL components is to establish a correspondance with the thread semantics defined by the AADL standard ([6] Subsection 5.4).

A PSL expression is evaluated with respect to a path or sequence of states labelled with the atomic propositions which are true in such state. Given a sequence, a PSL expression may *hold strongly*, *hold*, *be pending* or *fail*. The expression holds strongly when it contains no bad states, all future obligations have been met, and the expression holds on all extensions to the sequence. The expression holds (but does not hold strongly) when it contains no bad states, all future obligations have been met, and the expression may or may not hold on any given extension of the path. The expression is pending when it contains no bad states, but future obligations have not been met, and the expression may or may not hold on any given extension of the path. Finally, the expression fails when there is some bad state in the path, future obligations may or may not have been met and the expression will not hold on any extension of the path. Additionally, a PSL expression is evaluated with respect to a *clock context*, a boolean expression that determines in which cycles the expression is to be evaluated. The PSL standard does not specify any particular time granularity or what counts as a cycle or clock tick. It is up to verification tools to decide. The default context is true so that the expression is evaluated at every cycle.

There are several alternative ways to establish a correspondance between these paths and cycles and the states of an AADL thread. One possibility, is to consider a cycle every time the thread is dispatched. This is the natural choice when the thread is periodic. For aperiodic threads it is also possible to consider a cycle when the thread is dispatched, but in this case the dispatch occurs only when an event arrives at a port. For sporadic, timed or hybrid threads, the cycle would occur either by an event or by the specified period. If one adopts such convention, then the designer must be aware that the meaning of the PSL expressions depend on the type of thread. For example, the formula $a \wedge \mathsf{X} b$ asserting that a holds in the current cycle and b holds in the next cycle means, for a periodic thread, that a holds at the current time t according to the clock, and b holds at time $t+p$ where p is the thread's period. On the other hand, for an aperiodic thread the formula would mean that when an event arrives to one of the thread's ports, a holds, and b will hold the next time an event arrives.

If we are using the behaviour annex to specify implementations, the choice of associating cycles with dispatches may lead to the traditional interpretation of temporal operators with respect to automata, where "next" does really mean the next state. Since the behaviour annex allows for hierarchical state machines, by "state" we would mean a state in the flattened state machine, with a particular assignment of variables to values.

Another possibility is to treat all kinds of threads in the same way, as periodic threads, *i.e.*, assuming that there is an underlying periodic clock, even for aperiodic threads. In this case, there must be a way for the verification tool to obtain the current state of a thread at any time t of this underlying clock.

Since there are several possibilities, none of which seems to be *a priori* any more fundamental than the others, it is up to the developer to decide which interpretation of PSL expressions is more suitable.

4 An AGCL analysis tool

We have implemented a prototype of the AGCL annex and the analyses outlined in the previous sections as a plugin for OSATE. The tool allows the user to apply the analyses outlined in this paper, providing results sorted by either viewpoint or by component. When the result of a particular analysis fails, a counterexample is generated by the model checker. Our plug-in uses the NuSMV model-checker to check the validity of the formulas in question, but the underlying architecture can easily be extended to support other model-checkers.

A model-checker receives as input a model and a specification (temporal logic formula) and decides whether the model satisfies the formula or not. A model-checker can be used to check validity by checking the formula against a *universal model* for the formula, this is, a model that contains all possible states and transitions about which the formula could talk. For example, if a formula contains three atomic propositions, the universal model has three boolean variables and therefore eight states, all of which are initial, and all possible transitions between them. Such universal model contains every possible model of the formula embedded in it, and therefore every possible path. A linear temporal formula is valid if it holds in every path of every model, hence, it is valid if it holds in every path of the universal model. On the other hand, if there is at least one path in the universal model for which the formula doesn't hold, then there exists at least one model for which the formula doesn't hold and therefore the formula is not valid.

In terms of complexity, dealing with universal models might appear untractable, but the size of such models depends only on the size of the formulas (the number of atomic propositions) and not on the size of the state space of the components themselves. This observation combined with the fact that contracts don't need to describe all aspects of behaviour, and can be specified in separate viewpoints and analyzed independently makes the technique feasable.

5 Final remarks

We have sketeched an approach to specify and verify assume/guarantee contracts for AADL components and briefly discussed the kinds of analyses that can be performed and discussed our prototype implementing these. Given the space limitations we are unable to provide here the actual algorithms and their proof of correctness, but these are available in detail as a technical report [5]. The theory behind this work is based on [1] which developed a generic theory of contract-based reasoning applicable to a wide range of specification formalisms. In our technical report we extended and specialized that theory to PSL, showing in particular that when using PSL we can compose contracts, the basis for the compositional analysis. Our approach differs from other compositional techniques such as [3] in that we do not restrict assumptions to inputs and guarantees to outputs. Furthermore, with viewpoints, we make it possible to divide requirements into sets of smaller contracts, providing the developer with flexibility as well as making automatic verification more feasible. While this work is preliminary and we have yet to test the plugin on large-scale models, we believe our early results show promise, and contract-based analysis can provide a fundamental support to the development process of DRE systems.

References

1. S. S. Bauer, A. David, R. Hennicker, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Moving from specifications to contracts in component-based design. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7212 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2012.
2. A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *LNCS*, pages 200–225. Springer, 2007.
3. D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2012.
4. IEEE Computer Society. IEEE Standard for Property Specification Language (PSL). IEEE Standard 1850™-2010, June 2012.
5. E. Posse. Contract-based compositional analysis for reactive systems in RTEdge™, an **AADL**-based language. Tech. Rep. 2013-607, School of Computing – Queen's University, August 2013. http://research.cs.queensu.ca/TechReports/Reports/2013-607.pdf.
6. SAE International. Architecture Analysis & Design Language (AADL). SAE Standard AS5506b, 10 September 2012.
7. M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1997.
8. P. Wolper. The Tableau Method for Temporal Logic: An Overview. *Logique et Analyse*, 28(110–111):119–136, June–September 1985.

An Extension for AADL to Model Mixed-criticality Avionic Systems Deployed on IMA architectures with TTEthernet

Tiyam Robati[1], Amine El Kouhen[1], Abdelouahed Gherbi[1], Sardaouna Hamadou[2], and John Mullins[2]

[1] Dept. of Software and IT Engineering, École de Technologie Suprieure, Canada
{tiyam.robati.1@ens.etsmtl.ca, amine.elkouhen@etsmtl.ca, abdelouahed.gherbi@etsmtl.ca}
[2] Dept. of Computer and Software Eng, Ecole Polytechnique de Montreal, Canada
{firstname.lastname@polymtl.ca}

Abstract. Integrated modular avionics architectures combined with the emerging SAE TTEthernet standard provides a strong infrastructure for the deployment of mixed-critical avionic applications having stringent safety, reliability and performance requirements. The integration of such systems is a very complex and challenging engineering task. Therefore, a model-based approach, which endows system engineers with a methodology and the supporting tools to cope with this complexity, is of a paramount importance. In this research paper, we present an extension for the standard architecture and analysis modeling language AADL to enable modeling integrated multi-critical avionic applications deployed on TTEthernet-based IMA architectures. In particular, we present a metamodel which extends the core AADL metamodel with concepts and constraints relevant for this domain, we define the concrete textual syntax for this extension and we outline the implementation of this extension using the Open Source AADL Tool Environment (OSATE). Finally, we illustrate our AADL extension using a case study based on the Flight Management System.

Keywords: AADL, Time-Triggered Ethernet, AFDX, IMA

1 Introduction

On-board avionic systems are safety-critical systems which should meet strict safety, reliability and performance requirements. These systems have traditionally been engineered using what is called a federated architectures approach, where each function is designed and deployed to use its exclusive resources. This approach is however costly in terms of equipments and wiring. The Integrated Modular Avionics (IMA) architecture is an alternative approach, which is based a consolidation of resources [22]. This is achieved through resources sharing between functionalities. With IMA different avionic functions having different criticality levels (e.g. control functions and comfort functions) share the same

hardware resources leading to mixed-criticality systems. Moreover, IMA architectures are distributed using a communication infrastructure, which should also be able to meet the same level of safety and performance requirements.

Ethernet is a widely used standard network (IEEE 802.3) which is not only used as infrastructure for classic office systems but is increasingly supporting industrial and embedded systems due to the high bandwidths it provides. However, Ethernet does not meet strict time and safety critical applications. Several extensions to enhance the predictability of Ethernet have been developed. One of these extensions is the Avionic Full Duplex AFDX standard ARINC 664 [11]. AFDX is a deterministic real-time extension of Ethernet based on an static bandwidth scheduling and control using the concept of virtual links. The SAE standard TTEthernet [3] is the most recent Ethernet extension based on the time-triggered communication paradigm [14] [19] to achieve bounded latency and low jitter. A TTEthernet network implements a global time using clock synchronisation and offers fault isolation mechanisms to manage channel and nodes failures. TTEthernet integrates three data flow: Time-Triggered (TT) data flow which is the higher priority traffic; Rate Constrained (RC) traffic, which is equivalent to AFDX traffic, and Best Effort (BE) traffic. This makes TTEthernet suitable for mixed-criticality applications such as avionic and automotive applications where highly critical control functions such as a flight management system cohabit with less critical functions such as an entertainment system.

The focus of this research work is on avionic applications deployed on IMA architectures interconnected using TTEthernet. The advantages of this infrastructure are numerous. First, the IMA modules enable the resource sharing. Second, the combination of IMA and TTEthernet enables the error isolation provided not only at the level of the modules through the partitioning but also the level of the network using different data traffics and the concept of virtual links. Third, TTEthernet enable the safe integration of data traffics with different performance and reliability requirements. However, these systems are on the other hand complex and the integration of diverse applications with mixed-criticality levels having strict real-time requirements is very challenging. In order to control the complexity of such systems, a model-based approach, which provides the systems engineers with a methodology and the supporting tools to accomplish correctly and efficiently this integration, is required. A key element of such approach is a modeling language which allows the engineers to express the system at a convenient level of abstraction and to interface with sophisticated formal analysis techniques to verify safety and performance properties of the system.

AADL is a well-established standard modeling language in the domain of real-time critical systems. AADL has been extended to support the modeling of IMA with an Annex ARINC 653 [2]. However, there is no support for AADL to model the networking of IMA modules through the recent technology TTEthernet. We present in this paper an extension for AADL to support the modeling of IMA architectures interconnected using TTEthernet. In particular, we present a metamodel for the domain of IMA and TTEhernet. We provide a concrete

textual syntax based on this metamodel, which enables the system engineers to describe a full IMA-based avionic systems interconnected with TTEthernet. We have implemented this extension in the framework of the Open Source AADL Tools (OSATE2)[5] . We illustrate the expressiveness of this extension through it application to model a subsystem of the the Flight Management System [18].

This paper is organized as follows: In Section 2, we introduce the concepts of IMA and the main features of the TTEthernet standard. We describe in Section 3 the main components of the proposed extension metamodel and discuss the rational behind its design. We outline the implementation of the proposed extension in the framework of OSATE in Section 4. We show the application of the proposed extension with an illustrative example in Section 5. In Section 6, we succinctly review the most close related research works to ours. We conclude the paper and outline our ongoing and future research work in Section 7.

2 Background

In order to make this paper as self-contained as possible, we briefly introduce in this section the main concept of IMA and TTEthernet.

2.1 Integrated Modular Avionic Architecture (IMA)

The main idea underlying the concept of IMA architecture [22] is the sharing of resources between some functions while ensuring their isolation to prevent any interference between them. Resource sharing reduces the cost of large volume of wiring and equipment while the non interference guarantee is required for safety reasons. The IMA architecture is a modular real-time architecture for avionics systems defined in the standard ARINC653 [12]. Each functionality of the system is implemented by one or a set of functions distributed across different modules. A module represents a computing resource hosting many functions. Functions deployed on the same module may have different criticality levels. For safety reasons, the functions must be strictly isolated using partitions. The partitioning of these functions is two dimensional: spatial partitioning and temporal partitioning. The spatial partitioning is implemented by assigning statically all the resources for the partition being executed in a module and no other partition can have the access to the same resources at the same time. The temporal partitioning is rather implemented by allocating a periodic time window dedicated for the execution of each partition.

2.2 Time-Triggered Ethernet (TTEthernet)

The new SAE Time-Triggered Ethernet standard (TTEthernet) [3] specifies time-triggered services extending the Ethernet IEEE standard 802.3. TTEthernet is based on the Time-triggered communication paradigm [13] and therefore establishes a system-wide time base implemented through a synchronisation of the clocks of the end systems and switches. This results in bounded latency and

low jitter. TTEthernet integrates both time-triggered and event-triggered communication on the same physical network. TTEthernet limits latency and jitter for time-triggered (TT) traffic, limits latency for rate constrained (RC) traffic, while simultaneously supporting the best-effort (BE) traffic service of IEEE 802.3 Ethernet. This allows application of Ethernet as a unified networking infrastructure. It supports therefore the deployment of mixed-criticality applications at the network level.

3 Metamodel Extending AADL capability to model TTEthernet

In this section, we present the metamodel for our extention to AADL in order to support the modeling of TTEthernet, which will henceforth be called the AADL-TTEthernet metamodel. This meta-model captures the main concepts and characteristics of the TTEthernet standard. The AADL-TTEthernet metamodel will enable building a set of tools to perform the design and analysis of distributed IMA architectures using TTEthernet as communication infrastructure. We have designed the AADL-TTEthernet metamodel using the Eclipse Modeling Framework (EMF), which is also used to specify the AADL Core metamodel. This allows for a seamless integration of the AADL-TTEthernet in OSATE2 environment [5] in terms of dependencies and embedded Java API. On the other hand, using the same mechanism (i.e. Ecore) to specify the two metamodels eases the expression of the domain concepts dependencies and simplifies the navigation between them. This mechanisim also has been used in other works [17] aim at implementing new annex and extension to AADL. Our

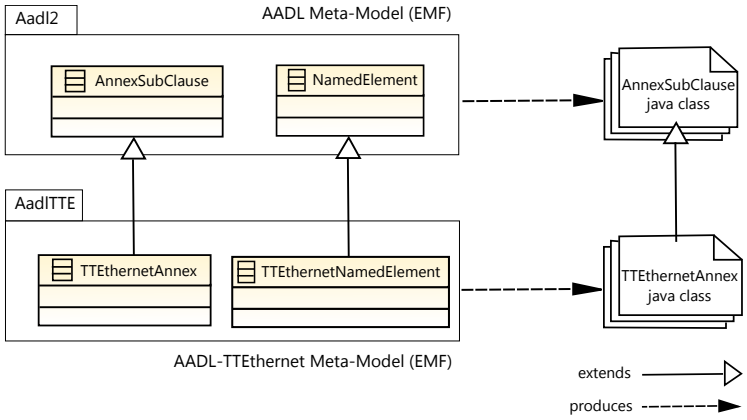


Fig. 1. AADL-TTEthernet meta-model dependencies

AADL-TTEthernet metamodel describes the structural aspect of a distributed IMA systems interconnected using TTEthernet and makes explicit all concepts

specified by this standard. The EMF framework generates automatically the Java implementation classes corresponding to the metamodel objects as it is shown in Figure 1. In order to extend AADL with our metamodel it is required to attach a TTEthernet model to an AADL component and to link the objects of our TTEthernet extension with AADL core objects. This is achieved by the implementation of the OSATE2 extension mechanism, which requires to link the TTEthernetAnnex concept in our metamodel to the AnnexSubclause concept of the AADL core as it is shown in Figure 1. This figure shows also how we use the EMF/Ecore inheritance mechanism to express the dependencies between the two metamodels. Consequently, a *TTEthernetAnnex* extends an *AnnexSubclause* and an *TTEthernetNamedElement* extends a *NamedElement*. In the metamodel, the TTEthernetAnnex concept, which links as shown in Figure 2 the metamodel to the AADL core metamodel, represents the overall model of a TTEhernet-networked IMA system which will undergoes different analysis to verify safety and performance properties. The global information about the network elements and the underlying implementation is described in the *TTEthernetAnnex* concept. The *TTEthernetAnnex* is composed of the following (Figure 2):

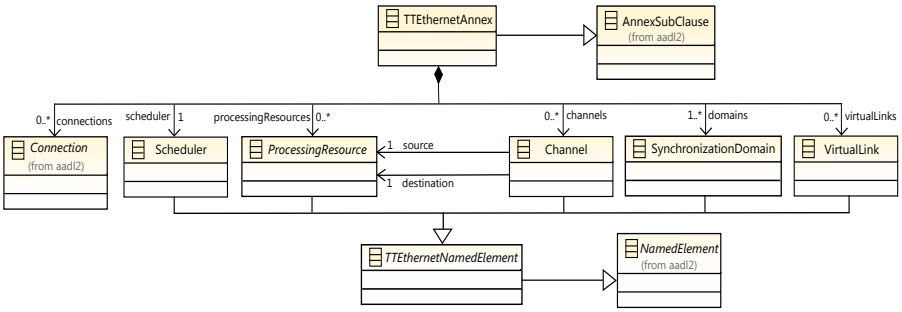


Fig. 2. AADL-TTEthernet meta-model overview

1. The *Synchronization domains* concept of TTEthernet standard. TTEthernet supports system-of-system communication by introducing *Synchronization domains* and *Synchronization priorities* as shown in Figure 3. *Synchronization domains* specify independent subsystems with respect to their synchronization. All the resources configured to belong to the same synchronization domain should synchronize with each other and components belonging to different synchronization domains in one TTEthernet network do not synchronize their local clocks.
2. The *Scheduler* is the entity in TTEthernet that is capable of producing a schedule, which should be compliant with the scheduling constraints of TTEthernet. These constraints are depicted in figure 3. The scheduler of TTEthernet request specific constraint which are presented mathematically

in [21]. These constraints are mentioned in figure 3 as constraints type which is related to constraint class.

3. The *Processing Resources* represent active hardware components in a network. They can be *Computing Resources* such as *Modules* (i.e. end systems) or *Networking Resources* such as switches as shown in Figure 4. All processing resources have *features* which can be parameters, access to physical buses, or ports (i.e, interfaces for frames inputs and outputs). A processing resource can be a synchronization master and can then transmits its local time to synchronize the whole network as shown in Figure 3. Several processing resources can be aggregated into logical groups called clusters as shown in Figure 4. A *Cluster* is associated with one synchronization domain. Each single cluster can establish and maintain synchronization by itself.

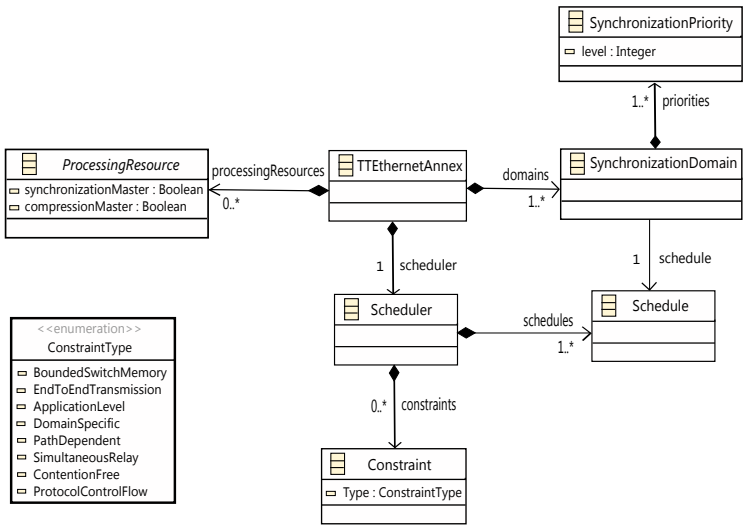


Fig. 3. Synchronization elements

The AADL-TTEthernet metamodel captures the different possible links between the components of a TTEthernet Network. These links can be either physical ones such as connections or logical ones such as channels or virtual links.

- A *Connection* is a link between two physical ports, usually realized as a copper or optical fiber cable. A connection may be unidirectional or bidirectional.
- A *Channel* is a logical connection from one source processing resource to another processing resource destination. A Channel is defined to map multi-cluster architectures.
- A *Virtual Link* is a logical link defined by ARINC 664 standard [11]. Each virtual link is associated with a dedicated maximum bandwidth, specified

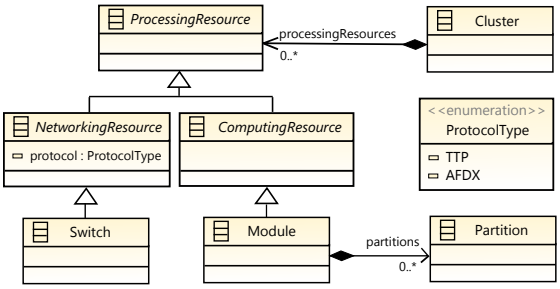


Fig. 4. Processing resources

by the minimum frame interval, called bandwidth allocation gap, and the maximum frame length.

The *Schedulable Resources* represents all the elements which are managed using the network scheduler. These resources can be the partitions hosted by module, the data transferred through the network (i.e, Frames), the networks communication channels or the virtual links as shown in Figure 5. A *Frame* is unit

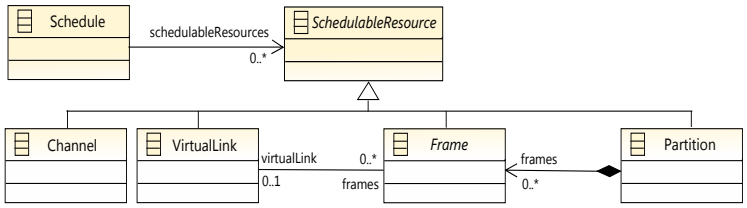


Fig. 5. Schedulable resources

of transmission, a data packet of fixed or variable length, encoded for digital transmission over a communication link as depicted in Figure 6. Considering its order of priority, a frame could be *Protocol Control Frame* (PCF), *TT* frame, *Rc* frame or *BE* frame.

4 Implementation of the TTEthernet Extension for AADL

4.1 Textual Syntax for the TTEthernet Extention for AADL

The definition of a textual syntax is provided by a grammar (i.e, a set of rules which define the composition of a language). In order to translate the textual syntax to its corresponding model, a lexer, a parser as well as a component for the semantical analysis (type checking, resolving of references, etc.) are required.

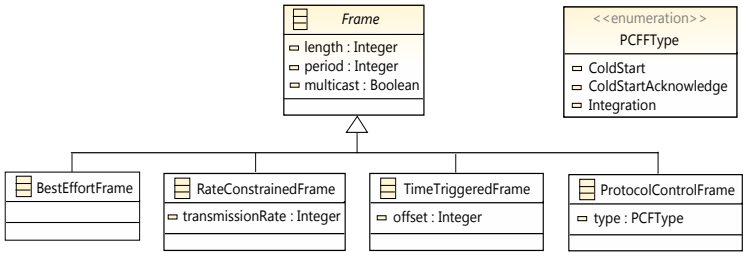


Fig. 6. Frame kinds

The backward transformation, from model to text, is provided by an emitter. All the components can be generated using the grammar $\Longleftrightarrow$ meta-model mapping definition [10]. Figure 7 demonstrates the selected framework to define textual syntax of our extension. It employs the data provided by the mapping definition used to generate the parser, emitter and an editor for the corresponding language to the metamodel. This editor can then use the generated parser and emitter to modify the text and the model. Therefore it is responsible for keeping the text and the model in sync, e.g., by calling the parser upon any changes on the text. Based on this mapping definition, several features of the editor can be generated, such as syntax highlighing, autocompletion or error reporting. To

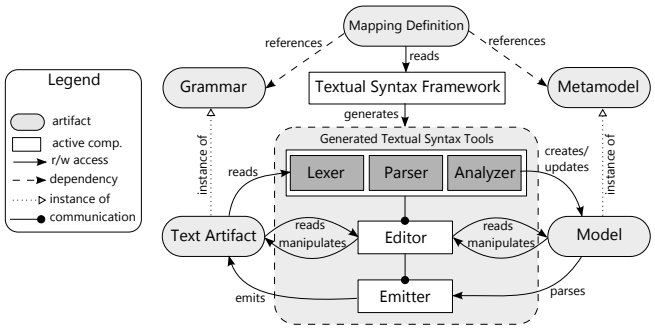


Fig. 7. General structure of a textual syntax framework

build the textual editor tool for our AADL-TTEthernet extension, we used the xText framework [8]. It implements the textual syntax according to an extended BNF. Figure 8 shows an excerpt of this xText grammar. In this xText framework, the AADL-TTEthernet metamodel concept is mapped to a Java implementation, where the TTEthernet objects names are used as class names. All attributes are implemented as private fields and public get- and set- methods. The composition relationships are realized in the same way as attributes and contribute to the constructor of the class. All classes support the Visitor pattern [9] to traverse the abstract syntax along the composition relationships [15].

The analyzer module scans the Abstract Syntax Tree (AST) and checks the semantics of the AADL-TTEthernet model. First, it proceeds to a resolution phase (e.g, naming resolver), which links TTEthernet objects to their corresponding AADL objects. In order to achieve this phase, we use the visitors (e.g, java classes) provided by OSATE2 to retrieve AADL objects. For the sake of the implementation of our AADL-TTEthernet extension, we have developed the visitors required to navigate through the AADL-TTEthernet AST. This phase adds information to the AST and makes its use easier.

```
1  grammar org.osate.ttethernet.xtext.Aadltte
2
3  import "http://ca.estmtl.aadl2/aadltte/1.0"
4  import "http://aadl.info/AADL/2.0" as aadl2
5  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
6
7  Partition returns Partition:
8    'Partition' name = ID
9        'frames' ':'frames += Frame*
10   'end' ID ';';
11 Frame:
12   RateConstrainedFrame | TimeTriggeredFrame | BestEffortFrame
         | ProtocolControlFrame
13 ;
14 SynchronizationPriority returns SynchronizationPriority:
15   'Synchronization Priority' name = ID
16     'level' level = Integer ';'
17   'end' ID ';';
```

Fig. 8. xText grammar overview for AADL-TTEthernet

4.2 Integration of the AADL-TTEthernet Compiler to OSATE2

Sublanguages are included into AADL specifications as annex subclauses. The latter may be inserted into AADL component types and AADL component implementations of an AADL model. OSATE2 currently provides four extension points that can be used to integrate a sublanguage into the tool environment. These extension points are designed to support parsing, unparsing, name resolution / semantic checking, and instantiation of annex models. From the AADL-TTEthernet EMF meta-model in the EMF framework, we generate the AADL-TTEthernet builder factory to build and manipulate TTEthernet objects used in the compiler. The compiler plug-in contains two modules: a parser/lexer and an analyzer. The integration of the AADL-TTEthernet plug-in is a two-steps process. First, we link the AADL-TTEthernet plug-in to the OSATE2 annex plug-in

using the Eclipse extension points mechanism. The annex plug-in defines extension points which allow to plug-ins be connected together as depicted in Figure 9. Second, we have to register our parser in the OSATE2 annex registry. As the AADL-TTEthernet metamodel becomes a part of the AADL description and the AADL-TTEthernet textual syntax tool is connected to OSATE2 registry, the AADL-TTEthernet plug-in is directly integrated and driven by OSATE2.

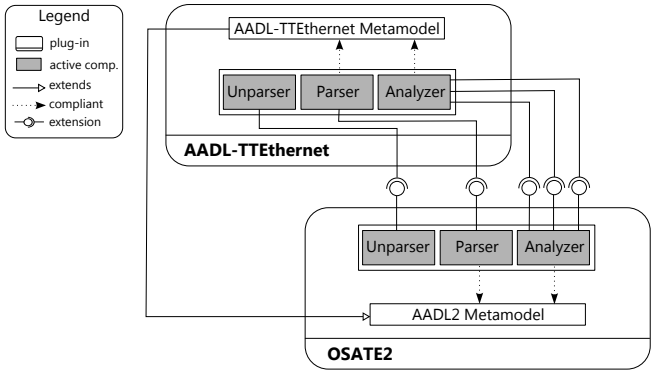


Fig. 9. AADL-TTEthernet plug-in integrated to OSATE2

5 An Example: A Model of a Subsystem of the Flight Management System

In this section, we illustrate the modeling of a distributed IMA system based on TTEthernet as a communication network using our extension for AADL. In order to do so, we use as a subsystem of the Flight Management System presented in [18]. This subsystem controls the display of static navigation information in the cockpit screens. The structure of the considered FMS subsystem in terms of modules and the partitions they host is shown in Figure 10. In the original version of the system considered in [18], the system is interconnected using AFDX. In our context, the modules are instead interconnected using TTEthernet. The AFDX data traffic in the original system corresponds to the RC traffic in the TTEthernet context. Table 1 shows a subset of the virtual links used in the FMS subsystem with their corresponding characteristics including the Bandwidth Allocation Gap (BAG), the sender modules of the VLs and the corresponding receiver modules. This subsystem can be modeled using our TTEthernet extension for AADL as follows. The extension is a sub-language for AADL, which can be included in *system implementation* of the AADL model of this system. The concrete textual syntax of AADL-TTEthernet extension provides several new reserved words, which correspond to the main concepts of the metamodel described previously, such as module, switch, partition, connection, virtual link,

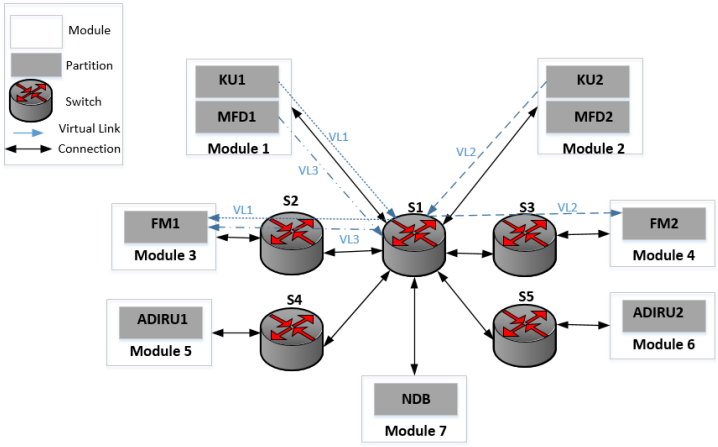


Fig. 10. A Subsystem of the Flight Management System

Virtual Link	Source	Destination	BAG	Direction
VL_1	KU_1	FM_1, FM_2	32	$\{S_1, S_2\}$, $\{S_1, S_3\}$
VL_2	KU_2	FM_1, FM_2	32	$\{S_1, S_2\}$, $\{S_1, S_3\}$
VL_3	FM_1	MFD_1	8	$\{S_2, S_1\}$

Table 1. Virtual Links details

Time-triggered frame, Rate Constraint frame and Best Effort frame. An excerpt of the model of the FMS subsystem using our AADL-TTEthernet is shown in Figure 11. The full specification of the model can be downloaded at [20].

6 Related Work

AADL presents two extension mechanisms, namely the property sets and the sublangues (i.e. annex). Several AADL extensions based on these mechanisms are now standardized as official annexes. These include the Data modeling annex, ARINC653 annex, the AADL Behavor Annex [2] , and Error Model Annex [1]. In addition, some research work have focused on extending the language using these extension mechanisms or investigating alternative ways. The most close research works to ours are reported in [7] and [17] . J. Delange et al. [7] present an approach based on AADL, which covers the the modeling, verification and implementation of ARINC653 systems. The authors describe in the work the modeling guidelines elaborated in the ARINC653 annex of the AADL standard. This approach is supported by a tool chain composed of Ocarina AADL toolsuite, AADL/ARINC653 runtime POK and Cheddar scheduling tool. G Lasnier et al. [17] present an implementation of the AADL behavior annex as an extension plug-in to the OSATE 2. We have implemented our AADL TTEthernet extension using similar techniques. M. lafaye et al. [16] define a modeling approach based

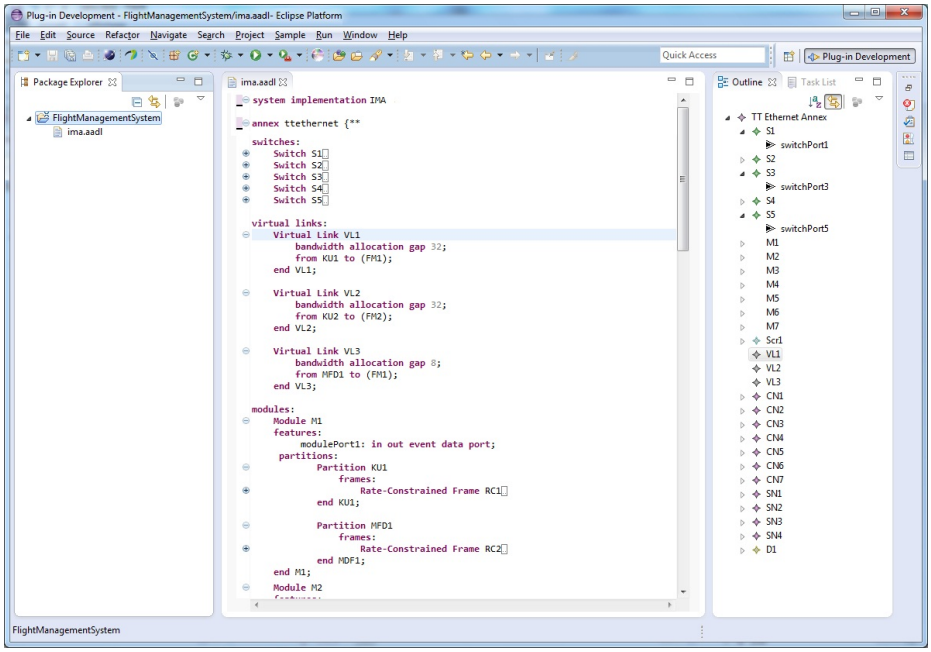


Fig. 11. Flight Management Subsystem Model using AADL TTEthernet Extension

on AADL and SystemC, which aims at the design and dynamic simulation of a IMA-based avionics platform. This is component-based approach, which can be used to dimension the architecture taking into consideration the application to be deployed while achieving early platform validation. De Niz and Fieler discuss in [6] how to extend the AADL language to include new features for the separation of concerns (i.e. Aspects). Based on this research work, it seems that the AADL extension mechanisms do not support the separation of concerns and new aspect-like constructs and mechanisms are then investigated. G. Brau et al. present in [4] a model of a subsystem of Flight Management System using AADL and show how to establish important parameters in the AADL model including the virtual links characteristics for instance. To the best of our knowledge, there is no published research work, which addresses the modeling of the TTEthernet standard as networking infrastructure for IMA architecture, which is the contribution of this work.

7 Conclusions and Future Work

The IMA architecture combined with the new SAE standard TTEthernet as communication infrastructure provide a strong platform for the deployment of distributed avionic applications. The integration of mixed-criticality applications on such platforms is a complex and challenging engineering activity. A model-

based approach based on the SAE standard architecture language AADL provides the system engineers with the tools to cope with this complexity. Modeling TTEthernet infrastructure using AADL is the research gap that we are targeting in this work. We have presented in this paper our contribution to address this issue, which consists in an extension for the standard architecture and analysis modeling language AADL to enable modeling integrated mixed-criticality avionic applications deployed on TTEthernet-based IMA architectures. In our ongoing research work, we aim at formalizing this extension in the form of a new annex through the SAE standardization process. Moreover, we aim at defining a formal semantics for our extension to allow transforming the AADL models built using our extension to models that are suitable for analysis techniques that can be used to verify relevant safety and performance properties.

References

1. SAE Aerospace. *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface Annex E: Error Model Annex,AS5506/1*, 2011.
2. SAE Aerospace. *SAE Architecture Analysis and Design Language (AADL) Annex Volume 2: Annex B: Data Modeling Annex Annex D: Behavior Model Annex Annex F: ARINC653 Annex, AS5506/2*, 2011.
3. SAE Aerospace. *Time-Triggered Ethernet*, sae as6802 edition, 2011.
4. Guillaume Brau, Jérôme Hugues, and Nicolas Navet. Refinement of aadl models using early-stage analysis methods: An avionics example. Technical Report TR-LASSY-13-06, Laboratory for Advanced Software Systems, 2013.
5. CMU/SEI. Open source aadl tool environment (osatev2). http://www.aadl.info, 2014.
6. Dionisio De Niz and Peter H Feiler. Aspects in the industry standard aadl. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 15–20. ACM, 2007.
7. Julien Delange, Laurent Pautet, Alain Plantec, Mickaël Kerboeuf, Frank Singhoff, and Fabrice Kordon. Validate, simulate, and implement arinc653 systems using the aadl. In *ACM SIGAda International Conference on Ada*, pages 31–44. ACM, 2009.
8. S. Efftinge. *Xtext reference documentation.* http://www.eclipse.org/gmt/oaw/doc/4.1/r80xtextReference.pdf, 2006.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
10. Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of concrete textual syntax mapping approaches. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 169–184. Springer Berlin Heidelberg.
11. Aeronautical Radio Incorporated. *ARINC Report 664P7-1 Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network.* AEEC, Maryland, USA, 2009.
12. Aeronautical Radio Incorporated. *ARINC Report 653P0 Avionics Application Software Standard Interface, Part 0, Overview of ARINC 653*, 2013.

13. G. Kopetz, Hermann; Bauer. The time-triggered architecture. volume vol.91, no.1, pp.112,126. Proceedings of the IEEE, 2003.
14. Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
15. Holger Krahn, Bernhard Rumpe, and Steven Vlkel. Integrated definition of abstract and concrete syntax for textual languages. In Gregor Engels, Bill Opdyke, DouglasC. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 286–300. Springer Berlin Heidelberg.
16. Michaël Lafaye, David Faura, Marc Gatti, and Laurent Pautet. A new modeling approach for ima platform early validation. In *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 17–20. ACM, 2010.
17. Gilles Lasnier, Laurent Pautet, Jérôme Hugues, and Lutz Wrage. An implementation of the behavior annex in the aadl-toolset osate2. In *IEEE ICECCS*, pages 332–337. IEEE Computer Society, 2011.
18. M.Lauer. Une méthode globale pour la vérification d'exigences temps réel-application á l'avionique modular intgrée, 2012. Thse de Doctorat, Institut National Polytechnique de Toulouse.
19. Roman Obermaisser. *Event-triggered and time-triggered control paradigms*, volume 22. Springer, 2004.
20. Tiyam Robati, Amine El Kouhen, and Abdelouahed Gherbi. Flight management subsystem model using aadl ttethernet extension. http://profs.etsmtl.ca/agherbi/ima.aadl, 2014.
21. W. Steiner. An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks. volume vol., no., pp.375,384. Real-Time Systems Symposium (RTSS), IEEE 31st, 2010.
22. Christopher B Watkins and Randy Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conference, 2007. DASC'07. IEEE/AIAA 26th*, pages 2–A. IEEE, 2007.

A discrete-event simulator for early validation of avionics systems

Denis Buzdalov and Alexey Khoroshilov
{buzdalov,khoroshilov}@ispras.ru

Institute for System Programming of the Russian Academy of Sciences
Moscow, Russia

Abstract. The paper discusses problems arising in development of avionics systems and considers how discrete-event simulation on the base of architecture models at the early stages of avionics design can help to mitigate some of them. A tool for simulation of AADL architecture models augmented by behavioural specifications is presented and its main design decisions are discussed.

Keywords: AADL, discrete-event simulation, early validation

1 Introduction

Nowadays avionics is responsible for control of almost all aspects of aircraft operation. As a result it became a complex system with thousands of sensors and actuators, and hundreds of software components spread across dozens of processor modules. Design and development of such system is a big challenge. And one of the biggest concerns is bugs introduced at the early stages of system design that are usually very expensive to fix in terms of both time and money as long as they often impact a number of components.

Let us demonstrate it on a simple example. A typical prerequisite to guarantee safety of flight is a real-time reaction on hazardous conditions. That leads to a number of safety-critical requirements to avionics subsystems that look like a limitation on time between a moment when a particular sensor reads actual data and a moment when software processes that data and takes appropriate actions, e.g. delivers a control command to an actuator or informs a pilot. If inability to satisfy such requirement is found during integration tests only, it may have catastrophic consequences for the project. For example, if an unexpected latency appears on an overloaded bus, it may require to redesign data flows between components and to reschedule software partitions that leads to redo significant part of safety analysis and other verification activities, i.e. to significant delays and cost increase for the project.

Model-driven system engineering is considered to be the most promising approach that can help to address this concern. The main idea behind it is to represent requirements and architecture decisions in form of models, i.e. in more or less machine-readable form, and then to apply automated verification techniques or even to automatically derive some parts of implementation from these

models. The benefits are manifold. First of all, automated analysis and verification help to identify as much problems as early as possible. But even if changes in system design are introduced, models allow to automate impact analysis and to reduce effort required for repeated verification.

The cornerstone element that enables analysis of various system characteristics is an architecture model of the system under development. One of modelling languages designed for this purpose is Architecture Analysis and Design Language (AADL) [11]. The core of AADL allows to describe an architecture of software-hardware systems. It supports constructions for step-by-step modelling, refinement and integration. Besides the core language, AADL has different extensions, e.g. for constraints definition, fault handling and behavioural modelling. AADL has pretty well-defined semantics that allows different tools to be applied to the same AADL models without significant effort. There are several open source frameworks supporting creation and analysis of AADL models including OSATE [1] and MASIW [7,4].

One way to classify the variety of methods of models analysis is to divide them to *analytical* and *sampling*.

Analytical methods allow to get guaranteed estimations of target characteristics, e.g. the worst-case latency. But the cost of the guarantee is excessively pessimistic estimations. Another drawback of analytical methods is that each of them applicable for a particular tasks only. That means that you usually have to develop a new method to estimate new characteristics or to get estimation in a little bit different conditions.

Sampling methods do not provide a guarantee in getting of worst-case estimations. But their benefit is that they can provide realistic estimations and can be much more easily adopted to new conditions and to new target characteristics.

To apply sampling methods to analysis of architecture models a simulation of the models is required. One of particular kind of simulation — *discrete-event simulation* — is considered in this work. This kind of simulation is naturally suited to the software-hardware systems modelling. In this approach functioning of the modelled system is represented as a sequence of discrete events. Each discrete event is an atomic action of one component's internal state change and interaction with the outer world and other components. All actions of a single discrete event are performed in a single moment of the simulation time.

In the paper we consider possible approaches to model behavioural characteristics of architecture models and discuss design decisions of our AADL simulation engine. Finally we discuss related works and overview possible direction of further improvements.

2 Modelling

In this section we are considering aspects of modelling which are relevant to the discrete-event simulation.

2.1 How Behaviour is Modelled

Approaches of the discrete-event simulation can differ in the ways of how behaviour of model components is modelled.

Depending on the type of the supported behaviour representation, different simulation and analysis tools can reach different quality of analysis and can run into different problems during it.

The AADL core does not contain any constructions to model behaviour of components but the language is easily extendable. That is why let us consider some ways of how behaviour can be modelled and some properties of these variants.

Behaviour model can, for example, be represented as a *randomized events flow* with given probability characteristics and description of how the component reacts to external events.

There is also a class of behaviour representations which can be called imperative. For example, behaviour can be represented as either a *finite state machine (FSM)*, extensions of FSM which work with extended memory state and time, or some other *transition systems*. Transition systems can be combined with randomized events flows, particularly as FSMs with the probabilistic non-determinism resolution.

Also the behaviour can be modelled as a *program model*, when model is a code in some programming language.

AADL has a standardized extension called Behavior Model annex [10]. This annex allows to model behaviour of components as an extended timed FSM interacting with its environment.

Finite state machines (in particular, extended and timed) and specialized transition systems are usually naturally suitable for describing of a behaviour of small components. Also, some of such models are studied well and can be analyzed in some other way except the execution. Such analyses can be used during the whole system analysis. But still, these models are not well-suited to modelling of complicated behaviours (in particular, requiring a lot of internal states and events).

By contrast, program models can be pretty conveniently used for modelling of very complicated behaviour but they usually can be used only for execution.

From the simulation point of view program models have an additional advantage: any simpler model (e.g. FSM and other transition systems) can be translated automatically to a program model. It means that if a simulation system supports program models, simpler model types can be supported automatically.

Randomized events flow which is a really useful representation sometimes, usually also can be translated to a program model automatically.

2.2 Levels of Abstraction

In this section we consider a model creation process in time. Models get their details during a long-term process but analysis of the models have to be performed from the very early stages. That is why simulation have to work with different levels of abstraction.

There are two dimensions of abstraction levels of system models to be considered.

Structural Abstraction The first one is a *structural abstraction*. Structurally abstract models contain components, internal structure of which is still going to be refined in future. For example, such models may define interconnection interfaces approximately or consider some complex subcomponents as black boxes.

For example, it may be known that some not fully specified device D is connected with a processor block P using some data transmission subsystem T (probably, involving some buses and devices) but it is not decided yet how this subsystem should be implemented. This means neither the connection interface of the device D nor the structure of the subsystem T are known.

Behavioural Abstraction Another dimension is a *behavioural abstraction* that depends on how accurately behavioural characteristics are modelled.

The behavioural characteristics include the following aspects that can be described with different accuracy:

- dependencies between input and output;
- influence of a component on the other ones;
- data which components are working with;
- time intervals between events.

A complexity of a behavioural model accurate by both structural and behavioural dimensions is more or less the same as a complexity of a behavioural model abstract by the both dimensions.

If structurally abstract model is built behaviourally accurate, behavioural models of each component can be very complex both by internal state and by interaction with its environment.

2.3 Analysis

Support of analysis of models represented in different abstraction levels is essential for early model verification and validation. In particular, it is really important to analyze structurally abstract and behaviourally accurate models as long as it allows to check single structure refinements and to expose incorrect ones.

To achieve this goal, the way of how behaviour is modelled have to be convenient for describing complex behaviours. It requires a convenient representation of an internal state and operations with it. This means a simulation system have to support program models.

But still, simple behaviours in structurally accurate models have to be defined in a convenient way, e.g. using formalisms based on transition systems. So combination of program models with other representations should be supported as well.

Another important aspect of usability of a simulation system is familiarity of a formalism (or at least its paradigm) to the users.

Considering the requirements discussed above the best candidate for the main formalism is an imperative high-level programming language which has rich libraries of collections, basic algorithms, etc.

3 AADL Simulator in MASIW

MASIW is an AADL-based framework for development and analysis of avionics and other safety critical systems. It contains various tools for development (text and graphical editors, model importers and generators) and analysis (static structure constrains analyzer, static AFDX latency analyzer, AFDX network simulator).

An AADL-model behaviour simulation tool is a welcome addition to such integrated toolset that enables early verification activities based on dynamic analysis.

In this section we consider the most interesting aspects of implementation of such simulation tool.

3.1 Program Models

As we discussed above, support for program behaviour models is an important feature for a simulator to be used for early validation. But there are several problems that should be resolved.

What Program to Consider a Behaviour Model The first problem is how to organize a program model. On the one hand, it seems to be useful to allow all convenient constructs of programming languages and libraries to be available in the program model. On the other hand, arbitrary program code cannot represent a behaviour model of a component because it have to be compatible with model interface of the component.

Representing behaviour of a component in a system, program models have to be able to model interaction with other components. In the context of event-driven simulation, models also have to be able to explicitly work with simulation time and discrete events.

Specific actions (like sending messages to other components) can be performed using special *simulation library* which is used by a program model. Time management can also be organized by such library. For example, there can be library calls modelling long-term computation or a launch of a long-lasting process.

There are other possible approaches to organize interface with simulation engine. For example, we can interpret the standard output of general executable code as outgoing commands of a component for model-specific activities like sending messages to other components and working with the simulation time. Similarly, standard input of a program can be considered as models-specific component input like incoming events and data. This approach is used in some systems in different areas (one of the widely used examples is FUSE [2]) but it

seems that this approach does not really fit for describing behaviour models of components in AADL-models.

As long as we are not limited by any legacy behavioural models we decided to use the following program models representation. We decided not to limit user in the internal state representation. Consequently, there is no limits for the code that works with it.

A simulation library was implemented to manage everything related to discrete-event nature of the simulation. It is intended to be explicitly used in the program model code. The library has a plenty of different calls for time management and for interaction with other components of a system model.

This model representation allows to express everything needed with maximum freedom in behaviour definition.

Interaction with the Simulation Library The question of how an interface of the simulation library should be organized is not trivial as well.

Two fundamentally different approaches were considered.

The first one expects a program model of any component to have a single entry point that handles all simulation events so that only the program model and not the simulation library determines when external events and data can be managed. Simulation library just provides an interface for getting information of new events. This approach can be called *synchronous*.

The synchronous approach has an advantage that simulation of a model can be organized in a very optimal way. If two parts of a model do not interact for some time, they can be simulated independently. Each independent part can have its own current simulation time.

But the synchronous approach has also a disadvantage: it is not possible in the general case to model a situation when some component launches long-lasting process which is provided by another component, and waits for a result of this process. Necessity of this is known from the practice.

Another approach of interaction with a simulation library can be called *asynchronous*. In this case a behaviour program model has several entry points which are in fact callbacks. These callbacks are called at some moment of simulation time when some specific sort of event arises, e.g. incoming events and data, different requests of other components and system events like simulation start.

Some of callbacks can return a value. In particular, such callbacks can model a response of a component to some other component's request and the mentioned above launching of long-lasting process that returns a value.

But still, asynchronous approach has its own shortcomings. First of all, simulation of different parts of a model cannot be performed independently. The reason of this is the following. Consider a component A having had performed some state changes at the simulation time $t_2 > t_1$ and a component B running at the simulation time t_1. Consider now that B has requested some information related to the internal state of A. If simulation engine lets the component A be simulated independently with B (i.e. including the time t_2), then the component B would be unable to get what it needs.

One of possible solutions of the problem is to prohibit any component A to perform any actions at time $t > t_1$ till other components which may request something from A have finished their execution at t_1. But this approach limits abilities to parallelize simulation activities.

We can try to solve the problem by storing states of faster running components (like A in the example above) at the moments earlier than the current one (for $t < t_2$) accessible by other components (like the state of A at t_1 accessible by B). But this approach still has problems. Consider we are running the component A at the t_2 storing all states of A at moments of time between t_1 and t_2. But consider that B at the moment t_1 can request a change of the state of A instead of just reading it. It means that we need to discard all stored states at the moments of time $t > t_1$ and to rerun simulation for A starting at the time t_1. So, this solution requires a lot of space to store all needed states and a lot of overhead for these copies management.

As a result, simulation of asynchronous models can be less effective comparing to synchronous ones regardless how the asynchronous simulation is organized.

Nevertheless, every synchronous behaviour can be represented as an asynchronous one. This means asynchronous simulation libraries are more universal than synchronous ones. Considering limited abilities of synchronous approach, asynchronous one is more preferred.

Table 1. Comparison of simulation library approaches

	synchronous	asynchronous
code representation	linear (like a single function)	a set of callbacks
effectiveness	easy to parallelize	whole model have to be simulated at a single moment of simulation time
abilities	issues to model long-term processes with a returned value	no limits

A comparison of synchronous and asynchronous approaches is presented in the table 1.

Execution Architecture It is important to consider that code execution of a program model that uses a simulation library have to be suspended at the end of discrete events to make other components to able to execute their own events at the same moment of the model time.

One of the simplest ways of organization of such alternating execution is the usage of a multiple threads. One thread is created for each component and they are suspended by a simulation system when a function of the end of discrete event is called. Thread is suspended until new event is raised for the corresponding component.

This approach is practical and pretty easy to implement. But it has some remarkable drawbacks.

One of them is that a behaviour model writer can easily create a deadlock. This situation can be preserved by using of some conventions for the behaviour model code but these conventions cannot be checked automatically by a simulation system.

But the main drawback, as it is seen from practice, is a high load to the threading subsystem of an operating system which is used to run simulation. Models can have tens of thousands of active components that requires the same number of threads. It worked well for Linux-based operating systems, but some other operating systems cannot manage such load. So, straightforward multithread approach leads to the portability issues.

However, there is another approach to organize program models execution that does not have drawbacks mentioned above. This approach is called *continuations* or *coroutines* in computer science [8,9].

The continuations approach allows to execute several program models in a single system thread. It means that program model code can be suspended and then it can be resumed from the very point is was suspended. Such suspension is performed by a simulation library and no special constructs have to be added to a program model.

This approach runs into a problem of correct error tracing because control flow changes vastly. So some effort is required to make error traces and stack traces looking as if control flow is unchanged.

Some modern and progressive programming languages, that use virtual machines for program execution, have the continuations approach built in. Classic languages have libraries implementing this approach but these libraries require an after-compilation program instrumentation.

Instrumentation of library program models is not a hard problem. But instrumentation of user program models can be a problem and it requires special handling of simulation start.

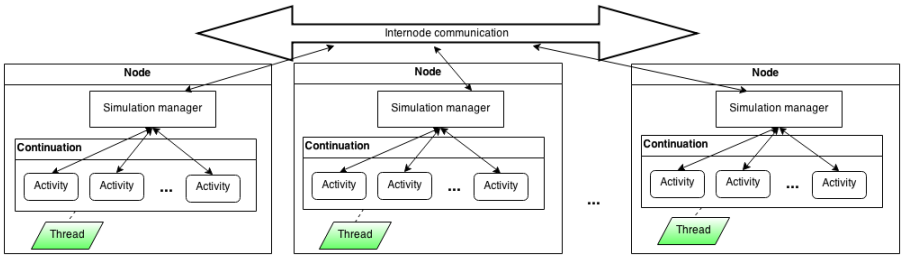


Fig. 1. Continuations approach on multiple nodes

Nevertheless, applying this approach (fig. 1) allows to increase maximal number of model components running on a single node to make it operating sys-

tem independent. It means that more optimal distribution of model components across simulation nodes can be achieved in comparison to the multithreaded multinode approach.

3.2 Behavior Annex

As it was discussed above, support of specialized transition systems as one of allowed types of behaviour model notations is a welcome feature for a simulation tool aimed to analyze models across various stages of development process. AADL has a standardized extension for defining such behaviour models called Behavior Model annex.

AADL Behavior model annex represents behaviour of a component as an extended finite state machine (FSM) of a specific kind. Actions on transitions can contain data state changes, interaction with the outer world and time delays. Transition conditions of such FSMs may depend on data state, external events generated by other components and time events.

Keeping in mind that our program models use simulation library containing operations for communication with environment and time, behavior model annex machines are translated to program models that use the simulation library. This translation is implemented in the MASIW AADL simulator.

Communication actions of the Behavior Model annex are implemented as the simulation library calls. Time-related actions are also mapped to the library calls. State changes (both FSM state and extended data state changes) are implemented naturally in a program model.

3.3 Built-in libraries

MASIW is targeted to the avionics models development and analysis. That is why during simulation we are running into behavioural aspects of different standards widely used in avionics.

For instance, the **ARINC653** standard is widely used for organizing execution of software in the avioncs system. It defines both structural and behavioural aspects of such systems.

Structural information can be represented by special standardized ARINC653 annex of AADL [10] and derived property set.

To ease development of ARINC653-based systems, a standarized behaviours for processors and other components were implemented. They can be used as a part of the behavioural model of a developed system. Moreover, library behaviours can be a base for user-defined behaviours.

AFDX network standard is a very important and widely used standard in avionics systems. AADL standard does not have support for modelling properties related to AFDX networks.

That is why we had to implement our own property set for defining structural aspects of models using AFDX. It have been used in mentioned above MASIW parts: static AFDX latency analyzer and AFDX network simulator.

We have implemented standardized behaviour of AFDX-specific network parts (like AFDX switches and network devices) as library behaviours which can be used in general AADL models simulation in MASIW. Also, some analyzers for these behaviours were implemented such as switches buffers and queues filling, counts of packets for different routes and links, statistics for packets drops and reasons of them and etc.

Such behaviour libraries let the model developer to focus on project features not paying much attention to how to model standard behaviour.

4 Related Works

Marzhin [5] is a proprietary simulator of AADL and AADL Behavior Annex models that mostly targets to analyze schedulability properties. It is based on existing multi-agent simulation kernel and it supports simulation of a subset of AADL and AADL Behavior Annex. The main distinction of MASIW simulator is support not only for Behavior Annex but for program models as well that allows to describe and then to simulate much more complex behaviours. Also, it is pretty easy to configure the MASIW simulator to manage and analyze various properties of a model.

OSATE framework [1] provides several plugins for model development and analysis. One of them called ADeS [12] is dedicated to analysis of behavioural properties using simulation. But unfortunately, its development stopped in 2008 and so this simulator does not support the last version of AADL which really differs from the supported first version.

AADS [13] is a translator of a subset of AADL with Behavior Annex to SCoPE [3] representation. SCoPE implements POSIX-based API that enables it to run appropriate software parts. Also, SystemC [6] is used for hardware simulation. The approach has a lot of benefits. But it is intended to the simulation of pretty accurate models to get accurate estimations. It seems to be not really usable on early steps of the model development.

5 Conclusion

MASIW AADL simulator supports simulation for all stages of the model development using the most appropriate behaviour model for each stage — program models for complex behaviours in structurally abstract models and specialized type of transition system called AADL Behaviour Model annex for other cases. A conclusion from implementation of the simulator is that having the program models support, it is quite natural and easy to implement a support of specialized transition system like AADL Behavior Model annex.

This simulator is integrated to the MASIW framework that supports most steps of the development and analysis process of AADL-models. This integration allowed to perform pretty fast and accurate analysis of avionics models (including models for the early validation).

Chosen behaviour model representation as a program model allows to model errors in models naturally. But support of standardized ways like AADL Error Model annex is a task for the future.

References

1. OSATE 2, https://wiki.sei.cmu.edu/aadl/index.php/Osate_2
2. Filesystem in Userspace, http://fuse.sourceforge.net/
3. SCoPE, http://www.teisa.unican.es/scope
4. Buzdalov, D., Zelenov, S., Kornykhin, E., Petrenko, A., Strakh, A., Ugnenko, A., Khoroshilov, A.: Tools for system design of integrated modular avionics. In: Proceedings of the Institute for System Programming of RAS. vol. 26, pp. 201–230 (2014)
5. Dissaux, P., Marc, O., Rubini, S., Fotsing, C., Gaudel, V., Singhoff, F., Plantec, A., Nguyen-Hong, V., Tran, H.N., et al.: The SMART project: Multi-agent scheduling simulation of real-time architectures. Proceedings of the ERTSS 2014 conference (2014)
6. IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005): IEEE Standard for Standard SystemC Language Reference Manual (Jan 2012)
7. Khoroshilov, A., Albitskiy, D., Koverninskiy, I., Olshanskiy, M., Petrenko, A., Ugnenko, A.: AADL-based toolset for IMA system design and integration. In: SAE 2012 Aerospace Electronics and Avionics Systems Conference. vol. 5, pp. 294–299. SAE Int. (2012)
8. Knuth, D.E.: The Art of Computer Programming vol. 1: Fundamental Algorithms, pp. 193–200. Addison-Wesley, 3 edn. (1997)
9. Reynolds, J.C.: The discoveries of continuations. Lisp and Symbolic Computation 6(3-4), 233–248 (1993)
10. SAE International: Architecture Analysis & Design Language (AADL) Annex Volume 2, SAE International standard AS5506/2 (2011), http://standards.sae.org/as5506/2/
11. SAE International: Architecture Analysis & Design Language (AADL), SAE International standard AS5506B (2012), http://standards.sae.org/as5506b/
12. Tilman, J.F., Schyn, A., Sezestre, R.: Simulation of system architectures with AADL. In: Proceedings of 4th International Congress on Embedded Real-Time Systems. ERTS 2008 (2008)
13. Varona-Gomez, R., Villar, E.: AADS+: AADL simulation including the behavioral annex. In: Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems. pp. 379–384. ICECCS '10, IEEE Computer Society, Washington, DC, USA (2010)

Multi-core Code Generation from Polychronous Programs with Time-Predictable Properties

Zhibin Yang, Jean-Paul Bodeveix, and Mamoun Filali

IRIT-CNRS, Université de Toulouse, France
{Zhibin.Yang,bodeveix,filali}@irit.fr

Abstract. Synchronous programming models capture concurrency in computation quite naturally, especially in its dataflow multi-clock (polychronous) flavor. With the rising importance of multi-core processors in safety-critical embedded systems or cyber-physical systems (CPS), there is a growing need for model-driven generation of multi-threaded code for multi-core systems. This paper proposes a build method of time-predictable system on multi-core, based on synchronous-model development. At the modeling level, the synchronous abstraction allows deterministic time semantics. Thus synchronous programming is a good choice for time-predictable system design. At the compiler level, the verified compiler from the synchronous language SIGNAL to our intermediate representation (S-CGA, a variant of guarded actions) and to multi-threaded code, preserves the time predictability. At the platform level, we propose a time-predictable multi-core architecture model in AADL (Architecture Analysis and Design Language), and then we map the multi-threaded code to this model. Therefore, our method integrates time predictability across several design layers.

Keywords: Synchronous languages, SIGNAL, Guarded actions, Verified compiler, Multi-core, Time predictability, AADL

1 Introduction

Safety-critical embedded systems or cyber-physical systems (CPS) distinguish themselves from general purpose computing systems by several characteristics, such as failure to meet deadlines may cause a catastrophic or at least highly undesirable system failure. Time-predictable system design [1, 21, 20] is concerned with the challenge of building systems in such a way that timing requirements can be guaranteed from the design. This means we can predict the system timing statically. With the widespread advent of multi-core processors in this category of systems, it further aggravates the complexity of timing analysis.

The synchronous abstraction allows deterministic time semantics. Therefore synchronous programming is a good choice for time-predictable system design. There are several synchronous languages, such as ESTEREL [5], LUSTRE [12] and QUARTZ [16] based on the *perfect synchrony* paradigm, and SIGNAL [4] based on the *polychrony* paradigm.

An integration infrastructure for different synchronous languages has gained a lot of interests in recent years [6, 19]. A classical solution is to use an intermediate representation. Guarded commands [10], also called *asynchronous guarded actions* by J. Brandt et al. [6], are a well-established concept for the description of concurrent systems. In the spirit of the guarded commands, J. Brandt et al. propose *synchronous guarded actions* [8] as an intermediate representation for their QUARTZ compiler. As the name suggests, it follows the synchronous model. Hence, the behavior (control flow as well as data flow) is basically described by sets of guarded actions of the form $\langle \gamma \Rightarrow \mathcal{A} \rangle$. The boolean condition γ is called the guard and $\mathcal{A}$ is called the action. To support the integration of synchronous, polychronous and asynchronous models (such as CAOS or SHIM), they propose an extended intermediate representation, that is *clocked guarded actions* [6] where one can declare explicitly a set of clocks. They also show how clocked guarded actions can be used for verification by symbolic model checking (SMV) and simulation by SystemC. [7] presents an embedding of polychronous programs into synchronous ones. The embedding gives us access to the methods and tools that already exist for synchronous specifications.

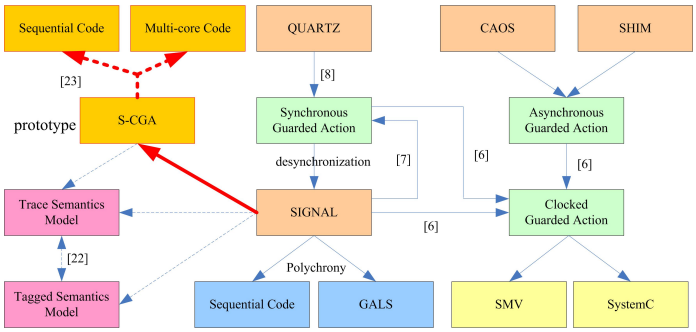


Fig. 1. A global view of the relation between our work and related work

For a safety-critical system, it is required that the compiler must be verified to ensure that the source program semantics is preserved. Our work mainly focuses on the SIGNAL language. We would like to extract a verified SIGNAL compiler from a correctness proof developed within the theorem prover Coq as it has been done in the GENEAUTO project for a part of the SIMULINK compiler. Our intermediate representation is a variant of clocked guarded actions (called S-CGA), and currently the target is multi-core code. In [23], we have already presented the compilation of sequential code and the proof of semantics preservation of the transformation from the kernel SIGNAL to S-CGA. There exist several semantics for SIGNAL, such as denotational semantics based on traces (called trace semantics) [11], denotational semantics based on tags which puts forward a partial order view of time (called tagged model semantics) [11], structural operational semantics defining inductively a set of possible transitions

[4, 11], etc. In [22], we have studied the equivalence between the trace semantics and the tagged model semantics, to assert a determined and precise semantics of the SIGNAL language. The relation between our work and related work is shown in Fig. 1.

The contribution of this paper is to propose a build method of time-predictable system on multi-core, based on synchronous-model development. At the modeling level, synchronous programming is a good choice for time-predictable system design. At the compiler level, the verified compiler from the synchronous language SIGNAL to our intermediate representation (S-CGA, a variant of guarded actions) and thus to multi-threaded code, preserves the time predictability. At the platform level, we propose a time-predictable multi-core architecture model in AADL (Architecture Analysis and Design Language) [15], and then we map the multi-threaded code to this model.

The rest of this paper is structured as follows. Section 2 presents the abstract syntax and the semantics of S-CGA. Section 3 gives the multi-threaded code generation schema from S-CGA. The time-predictable multi-core architecture model and the mapping from multi-threaded code to that model are presented in Section 4. Section 5 gives some concluding remarks.

2 S-CGA

In papers such as [6], clocked guarded actions has been defined as a common representation for synchronous (via synchronous guarded actions), polychronous and asynchronous (via asynchronous guarded actions) models. It has a multi-clocked feature. However, in contrast to the SIGNAL language, clocked guarded actions can evaluate a variable even if its clock does not hold [6] for supporting the asynchronous view. Since we focus on the polychronous view, we introduce S-CGA, which is a variant of clocked guarded actions. S-CGA constrains variable accesses as done by SIGNAL. In this section, we first present the syntax of S-CGA, and then we give the denotational semantics of S-CGA based on the trace model.

S-CGA has the same structure as clocked guarded actions, but they have *different semantics.*

Definition 1 (S-CGA). *A S-CGA system is represented by a set of guarded actions of the form* $\langle \gamma \Rightarrow \mathcal{A} \rangle$ *defined over a set of variables* X*. The Boolean condition* γ *is called the guard and* $\mathcal{A}$ *is called the action. Guarded actions can be of the following forms:*

$$\begin{array}{lll} (1) & \gamma \Rightarrow x = \tau & \text{(immediate)} \\ (2) & \gamma \Rightarrow next(x) = \tau & \text{(delayed)} \\ (3) & \gamma \Rightarrow \ assume(\sigma) & \text{(assumption)} \end{array}$$

where

- *the guard* γ *is a Boolean condition over the variables of* X*, their respective clocks (for a variable* $x \in X$*, we denote its clock* $\hat{x}$*), and their respective initial clocks (denoted* $init(\hat{x})$*),*

– τ *is an expression over* X,
– σ *is a Boolean expression over the variables of* X *and their clocks.*

An immediate assignment $x = \tau$ writes the value of τ immediately to the variable x. The form (1) implicitly imposes that if γ is defined[1] and its value is true, then x is present and τ is defined. Moreover, $init(\hat{x})$ exactly holds the first instant when x is present.

A delayed assignment $next(x) = \tau$ evaluates τ in the given instant but changes the value of the variable x at next time clock $\hat{x}$ ticks.

The form (3) defines a constraint. It determines a Boolean condition which has to hold when γ is defined and true. All the execution traces must satisfy this constraint. Otherwise, they are ignored.

Guarded actions are composed by using the parallel operator $\|$.

An S-CGA example [2] (Example 1) is shown as follows.

$true \Rightarrow assume(\hat{y_1} = \hat{x})$	$\hat{z} \wedge z \Rightarrow s_1 = f(y_1)$
$init(\hat{y_1}) \Rightarrow y_1 = 1$	$\hat{s_2} \Rightarrow s_2 = s_1 + 1$
$\hat{y_1} \Rightarrow next(y_1) = x$	$\hat{s_1} \Rightarrow assume(\hat{s_2})$
$true \Rightarrow assume(\hat{y_2} = \hat{x})$	$\hat{s_3} \Rightarrow assume(\hat{z} \wedge (not\ z))$
$init(\hat{y_2}) \Rightarrow y_2 = 2$	$\hat{z} \wedge (not\ z) \Rightarrow s_3 = f(y_2)$
$\hat{y_2} \Rightarrow next(y_2) = x$	$\hat{s_4} \Rightarrow s_4 = s_3 + 2$
$true \Rightarrow assume(\hat{x} = \hat{z})$	$\hat{s_3} \Rightarrow assume(\hat{s_4})$
$\hat{s_1} \Rightarrow assume(\hat{z} \wedge z)$	

Definition 2 (Trace semantics of S-CGA). *The trace semantics of a S-CGA system is defined as a set of traces, that is* $[\![SCGA]\!] = \{S \mid \forall scga \in SCGA, [\![scga]\!]_S = true\}$. *We have the following semantics rules,*

$$
\begin{aligned}
(1)\quad & [\![\gamma \Rightarrow x = \tau]\!]_S = \\
& \quad \forall i \in \mathbb{N}, \widehat{[\![\gamma]\!]}_{S,i} \wedge [\![\gamma]\!]_{S,i} \\
& \quad \rightarrow (\widehat{[\![x]\!]}_{S,i} \wedge \widehat{[\![\tau]\!]}_{S,i} \wedge [\![x]\!]_{S,i} = [\![\tau]\!]_{S,i})
\end{aligned}
$$

$$
\begin{aligned}
(2)\quad & [\![\gamma \Rightarrow next(x) = \tau]\!]_S = \\
& \quad \forall i_1 < i_2 \in \mathbb{N}, \\
& \quad ((\forall i' \in \mathbb{N},\ i_1 < i' < i_2 \rightarrow \neg\widehat{[\![x]\!]}_{S,i'}) \ \wedge \widehat{[\![\gamma]\!]}_{S,i_1} \wedge [\![\gamma]\!]_{S,i_1}) \\
& \quad \rightarrow (\widehat{[\![x]\!]}_{S,i_1} \wedge \widehat{[\![\tau]\!]}_{S,i_1} \wedge (\widehat{[\![x]\!]}_{S,i_2} \rightarrow [\![x]\!]_{S,i_2} = [\![\tau]\!]_{S,i_1}))
\end{aligned}
$$

$$
\begin{aligned}
(3)\quad & [\![\gamma \Rightarrow\ assume(\sigma)]\!]_S = \\
& \quad \forall i \in \mathbb{N}, \widehat{[\![\gamma]\!]}_{S,i} \wedge [\![\gamma]\!]_{S,i} \rightarrow \widehat{[\![\sigma]\!]}_{S,i} \wedge [\![\sigma]\!]_{S,i}
\end{aligned}
$$

– Rule (1): when γ is present, and the value of γ is true, x and τ are both present, and the value of x is that of τ.
– Rule (2): when γ is present and the value of γ is true at instant i_1, x and τ are present at i_1, and if i_2 is the next instant where x is present, then the value of x at i_2 is that of τ at instant i_1.

[1] An expression is said to be defined if all the variables it contains are present.
[2] If two guarded actions update the same variables, the guards must be exclusive.

– Rule (3): when γ is present, and the value of γ is true, σ is present and true.

The semantics of S-CGA composition is defined as $[\![scga_1 \parallel scga_2]\!]_S = [\![scga_1]\!]_S \wedge [\![scga_2]\!]_S$.

In [23], we have already presented the translation rules from the kernel SIGNAL to S-CGA, and give the proof of the semantics preservation in Coq.

3 From S-CGA to Multi-threaded Code

The SIGNAL compilation process contains one major analysis called *clock calculus* from which *code generation* directly follows. Moreover, the clock calculus contains several steps, such as the synchronization of each process, i.e., an equation system over clocks; the resolution of the system of clock equations; the construction of a clock hierarchy on which the automatic code generation strongly relies. Our goal here is to adapt the clock calculus to S-CGA.

Based on the semantics of S-CGA, we can get the equation system over clocks. The general rules are given as follows.

S-CGA	Clock Equations
$\gamma \Rightarrow x = \tau$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow next(x) = \tau$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow assume(\sigma)$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{\sigma} \wedge \sigma$
	$init(\hat{x}) \rightarrow \hat{x}\ (\forall x \in X)$

As a first step, we just consider the *endochrony* property [3], namely we can construct a clock hierarchy based on the resolution of the system of clock equations. The clock hierarchy of Example 1 (with three clock equivalence classes C0, C1, and C2) is shown in Fig. 2. In the figure, for instance *clk_x* denotes $\hat{x}$.

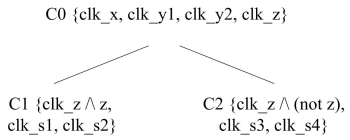


Fig. 2. Clock hierarchy

Moreover, we construct the data-dependency graph (DDG, as shown in Fig. 3) based on the variables reading and writing.

Finally, the multi-threaded code generation is based on both the clock hierarchy and the data dependency graph. First, we map the guarded actions to threads (i.e. partitions, as shown in Fig. 3). As presented in Fig. 4, we would like to treat the partition methods generally, this means different partition methods (such as the vertical way [2] for a concurrent execution, the horizontal way [3]

[3] The weak endochrony [14] property will be considered in the future.

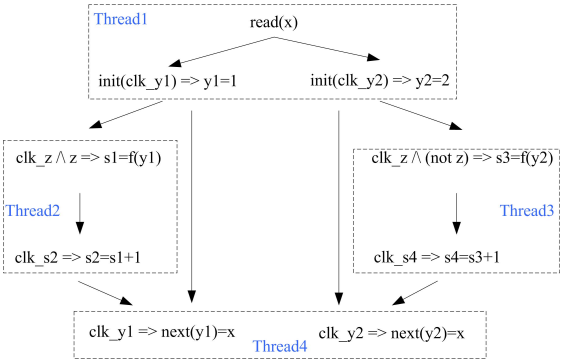


Fig. 3. Data dependency graph

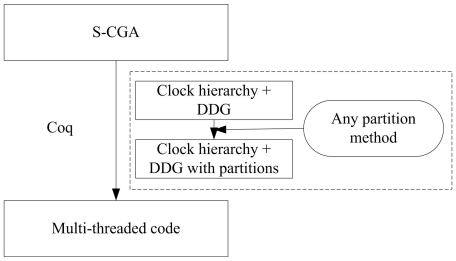


Fig. 4. The proof idea

for a pipelined execution, etc) don't affect the proof (here we don't consider performance). Second, in each thread, we organize the guarded actions based on the clock hierarchy. For example, the two guards in Thread2 belong to the same clock equivalence class, so they are merged inside the same control condition in the generated code. Third, we add wait/notify synchronization among the threads. A code fragment of Thread2 is given as follows.

```
/*Thread 2*/
void step()
{
  wait(Thread1);
  if(C1){
     s1 = f(y1);
     s2 = s1 + 1;}
  notify(Thread4);
}
```

4 Mapping Multi-threaded Code to Multi-core

To allow for static prediction of the system timing, we need time-predictable processor architectures, thus we know all the architecture details such as the pipeline and the memory hierarchy to analyze the execution time of programs.

Furthermore, the mapping from multi-threaded code to multi-core architectures should be also static and deterministic.

4.1 A time-predictable multi-core architecture model

With the advent of multi-core architectures, interference between threads on shared resources further complicates analysis. There are some recommendations from R. Wilhelm et al. [21, 20], i.e., the better way is to reduce the time interference: (1) pipeline with static branch prediction and with in-order execution; (2) separation of caches (instruction and data caches); (3) LRU (Least Recently Used) cache replacement policy; and (4) access of main memory via a TDMA (Time Division Multiple Access) scheme. In the EC funded project T-CREST [4], M. Schoeberl et al. [18, 17] propose a new form of organization for the instruction cache, named *method cache* (MC), and split data caches (including *stack cache* (SC), *static data cache* (SDC), *constants data cache* (CDC), and *heap allocated data cache* (HC)), to increase the time predictability and to tighten the WCET. The method cache stores complete methods and cache misses occur only on method invocation and return. They split the data cache for different data areas, thus data cache analysis can be performed individually for the different areas. In our work, heap is avoided to be used because we don't use dynamic memory allocation in our multi-threaded code.

Based on these existing work, we would like to model a time-predictable multi-core architecture in AADL. AADL is an SAE (Society of Automotive Engineers) architecture description language standard for embedded real-time systems, and supports several kinds of system analysis such as schedulability analysis. Moreover, we have already worked on the semantics of different AADL subsets such as [24]. So we envision how to validate semantically the mapping from the language level to the architecture level.

Our multi-core architecture model is illustrated in Fig. 5. Inside the core, we consider static branch prediction and in-order execution in the pipeline. A simplified instruction set (*get_instruction*, *compute*, *write_data*, and *read_data*) is used. As a first step, we just consider a first level cache (i.e. without L2 and L3). Each core is associated with a method cache, a stack cache, a static data cache, and a constants data cache. However, the same principle of cache splitting can be applied to L2 and L3 caches. The extension of the timing analysis for a cache hierarchy is straight forward. Moreover, TDMA-based resource arbitration allocates statically-computed slots to the cores.

As proposed by [9], a core is associated with an AADL *processor* component and a multi-core processor with an AADL *system* component containing multiple AADL processor subcomponents, each one representing a separate core. This modeling approach provides flexibility: an AADL system can contain other components to represent cache, and shared bus, etc. For that purpose, we define specific modeling patterns with new properties (such as *Multi_Core_Properties*). A part of AADL specification is given in Fig. 6.

[4] Time-predictable Multi-Core Architecture for Embedded Systems

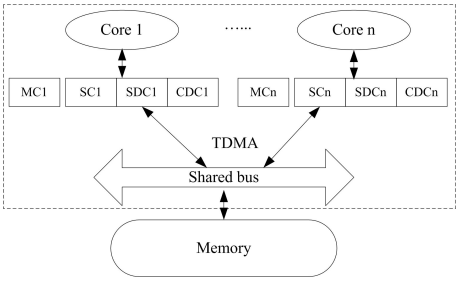


Fig. 5. A time-predictable multi-core architecture model

```
processor core
features
 MC: requires bus access cache_bus;
 SC: requires bus access cache_bus;
 SDC: requires bus access cache_bus;
 CDC: requires bus access cache_bus;
end core;

processor implementation core.impl
properties
 Multi_Core_Properties::Branch_Prediction => Static;
 Multi_Core_Properties::Execution_Order => In_Order;
end core.impl;

system multicore
features
 ExtMem: provides bus access shared_bus.impl;
end multicore;

system implementation multicore.impl
subcomponents
 Core1: processor core.impl{Multi_Core_Properties::Core_Id => 1;};
 Core2: processor core.impl{Multi_Core_Properties::Core_Id => 2;};
 Cache_MC1: memory method_cache.impl{Multi_Core_Properties::MC_Id => 1;};
 Cache_MC2: memory method_cache.impl{Multi_Core_Properties::MC_Id => 2;};
 ...
 SBus: bus shared_bus.impl;
 C2CBus1: bus cache_bus;
 C2CBus2: bus cache_bus;
connections
 bus access C2CBus1 -> Core1.MC;
 bus access C2CBus2 -> Core2.MC;
 bus access SBus -> Cache_MC1.Cache_Bus;
 bus access SBus -> Cache_MC2.Cache_Bus;
 ...
end multicore.impl;
```

Fig. 6. A part of the AADL specification of the time-predictable multi-core architecture

4.2 The mapping method

To preserve the time predictability, we consider static mapping and scheduling. Take the example shown in the last section. It generates a configuration file (such as *num_of_threads=4*) in multi-threaded code generation. Moreover, we have a manual configuration file for the time-predictable multi-core architecture

model, for example *num_of_cores=4*. Thus, we can generate a static mapping and scheduling, for instance:

- Thread1 $\mapsto$ Core1, Thread2 $\mapsto$ Core2, Thread3 $\mapsto$ Core3, and Thread4 $\mapsto$ Core4.
- Thread1: notify(Thread2), notify(Thread3);
  Thread2: wait(Thread1), notify(Thread4);
  Thread3: wait(Thread1), notify(Thread4);
  Thread4: wait(Thread2), wait(Thread3).

Based on the simplified instruction set (considered in the architecture model), the multi-core code can be generated. Thanks to the mechanizations such as method cache, split data caches, TDMA and static scheduling, the execution time of the multi-core code can be bounded.

5 Conclusion and Future Work

With the widespread advent of multi-core processors in safety-critical embedded systems or cyber-physical systems (CPS), it further aggravates the complexity of timing analysis. This paper proposes a build method of time-predictable system on multi-core, based on synchronous-model development. Our method integrates time predictability across several design layers, i.e., synchronous programming, verified compiler, and time-predictable multi-core architecture model. Interaction among cores might also arm software isolation layers, such as the one defined in ARINC653. Thanks to the existing work such as [9] and [13] on AADL modeling on multi-core architectures and their association with ARINC653, we would like to associate our work with partitioned architectures in the future.

Acknowledgments: This work was supported by the RTRA STAE Foundation in France (http://www.fondation-stae.net/).

References

1. P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, Mar. 2014.
2. D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. DATE '10, pages 949–952, 2010.
3. D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Generating software pipelines for OpenMP. In M. Dietrich, editor, *MBMV*, pages 11–20. Fraunhofer Verlag, 2010.
4. A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
5. F. Boussinot and R. de Simone. The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

6. J. Brandt, M. Gemünde, K. Schneider, S. K. Shukla, and J.-P. Talpin. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. *Design Automation for Embedded Systems*, pages 1–35, 2012.
7. J. Brandt, M. Gemunde, K. Schneider, S. K. Shukla, and J.-P. Talpin. Embedding polychrony into synchrony. *IEEE Trans. Software Eng.*, 39(7):917–929, 2013.
8. J. Brandt and K. Schneider. Separate translation of synchronous programs to guarded actions. *Internal Report 382/11, Department of Computer Science, University of Kaiserslautern*, 2011.
9. J. Delange and P. Feiler. Design and analysis of multi-core architecture for cyber-physical systems. In *5th Embedded Real Time Software and Systems*, ERTS'14, February 2014.
10. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
11. A. Gamatié. *Designing embedded systems with the SIGNAL programming language*. Springer, 2010.
12. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
13. J. Hugues. AADLib, a library of reusable AADL models. In *SAE Aerotech 2013 Congress & Exhibition (Montreal, Canada)*, September 2013.
14. D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.
15. SAE. AS5506A: Architecture Analysis and Design Language (AADL) Version 2.0. 2009.
16. K. Schneider. The synchronous programming language quartz. *Internal report, Department of Computer Science, University of Kaiserslautern, Germany*, 2010.
17. M. Schoeberl. A time predictable instruction cache for a Java processor. In R. Meersman, Z. Tari, and A. Corsaro, editors, *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, volume 3292 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 2004.
18. M. Schoeberl, B. Huber, and W. Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, 2013.
19. J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla. Constructive polychronous systems. In *Logical Foundations of Computer Science*, volume 7734 of *Lecture Notes in Computer Science*, pages 335–349. Springer, 2013.
20. L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Syst.*, 28(2-3):157–177, Nov. 2004.
21. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem: Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.
22. Z. Yang, J.-P. Bodeveix, and M. Filali. A comparative study of two formal semantics of the SIGNAL language. *Frontiers of Computer Science*, 7(5):673–693, 2013.
23. Z. Yang, J.-P. Bodeveix, M. Filali, H. Kai, and D. Ma. A verified transformation: From polychronous programs to a variant of clocked guarded actions. In *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, SCOPES '14, pages 128–137. ACM, 2014.
24. Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin. From AADL to timed abstract state machines: A verified model transformation. *Journal of Systems and Software*, 93:42–68, 2014.

Modeling Shared-Memory Multiprocessor Systems with AADL

Stéphane Rubini[1], Pierre Dissaux[2], and Frank Singhoff[1]

[1] Université de Bretagne Occidentale, UMR 6285, Lab-STICC, Brest, France
[2] Ellidiss Technlogies, Brest, France
{stephane.rubini,frank.singhoff}@univ-brest.fr
pierre.dissaux@ellidiss.com

Abstract. Multiprocessor chips are now commonly supplied by IC manufacturers. Real-time applications based on this kind of execution platforms are difficult to develop for many reasons: complexity, design space, unpredictability, ... The allocation of the multiple hardware resources to the software entities could have a great impact on the final result.
Then, if the interest to represent the set of hardware computing resources inside an architectural model is obvious, we argue that other hardware elements like shared buses or memory hierarchies must be included in the models if analyze of the timing behavior is expected to be performed. This article gives guidelines to represent, with AADL, shared-memory multiprocessing systems and their memory hierarchies while keeping a high-level model of the hardware architecture.

Keywords: Multiprocessor, Architecture Description Language, Memory Hierarchy

Introduction

The growth potential for computing power supplied by general purpose monoprocessor (GPU) systems, is quite decreasing. New gains are related to complex hardware structures and to very advanced fabrication technologies. Especially in the embedded system context, the additional power consumption to paid for increasing the instruction rate is high.

Today, the main answer to that problem is to multiply the number of processing units in VLSI chips. Less aggressive internal core designs allow for increasing the hardware efficiency and reducing the thermal problems (hot spots).

Moreover, as the execution units are multiple, it becomes possible to specialize some of them for dedicated usages. As an example, System-On-Chips like TI OMAPs include multiples GPUs, a DSP and an image processing unit.

These Programmable Heterogeneous Multiprocessors (PHMs) are now execution platforms that the designers must consider when they develop new products. The focus should not be only on the individual processing units, but on the whole hardware system. An early knowledge of some non-functional details of the execution target might be a condition to lead a project to completion.

A meaningful performance analysis cannot be conducted without considering a realistic model of the platform, and the data flow. The challenge is to eliminate low-level complexity, while the functional or non-functional behaviors remain close to the reality of the execution environment. The complexity of both the hardware execution platform and the software application requires that designers develop high-level models of their systems.

At the same time, system design space exploration requires to apply separation-of-concerns principles and to distinguish the application model from the architecture platform one. This approach requires that a deployment step maps software entities onto the hardware resources, statically or dynamically like for instance in the case of multi-processor scheduling decisions.

The aim of the paper is to propose a level of abstraction of the hardware for complex shared-memory multi-processor platforms. The section 1 focuses on the modeling of the processing resources based only on the functional point of view. The usefulness of such models are tested for the allocation of software entities on processors through deployment processes. The section 2 develops these descriptions to include memory hierarchy models, as a major structuring entities by considering the information flow into the system. In the next section, we apply those modeling guidelines to a real board. Hence, examples of tools that work or could work from the modeling level that we propose are given. Finally, after the description of related works, we conclude on some uncovered features of the hardware platform that could enrich our modeling guidelines.

1 Resource allocations

When a hardware platform provides multiple execution units, the design process must include an allocation or a placement strategy. The system model defines what software entities are assigned to hardware resources. Such assignments may be done by the mean of a component-containment hierarchy[1]. However AADL addresses the placement of software on a hardware resources through property values. Such binding properties form a deployment or an allocation layer in the model (Fig. 1).

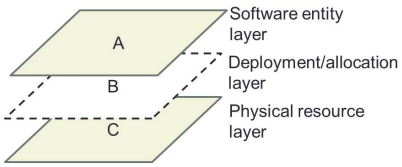


Fig. 1. Layered model

The next two paragraphs illustrate the usage of binding properties for allocating tasks and logical partitions to processors.

Processing unit modeling and allocation In multi-processing systems, the processing units may be implemented as a separated chip (a processor), or grouped into a single chip (a core). Finally, the data path of a core or a processor may be shared to execute concurrently several execution threads (physical threads)[2].

A way to model the system is to enumerate, at the same level, all the processing units without consideration of their actual implementation. The model of the Fig. 2 represents the available processing units of a PHM as arrays of AADL processors which group together those compatible with the same instruction set.

```
processor isa1 end isa1;
processor implementation isa1.proc end isa1.proc;
processor implementation isa1.core end isa1.core;
processor implementation isa1.phys_thread end isa1.phys_thread;
processor isa2 end isa2;
system mpHardSystem end mpHardSystem;
system implementation mpHardSystem.impl
subcomponents
  PU_isa1 : processor isa1[N];
  PU_isa2 : processor isa2[M];
end mpHardSystem.impl;
```

Fig. 2. Hardware layer (layer C of Fig. 1): set of processing units. N and M parameters are constant fixed in the model. `isa2` processing unit implementations are not exposed.

The ability of this model to express the allocation of the hardware execution resources to the software tasks is shown Fig. 3.

```
Allowed_Processor_Binding => (reference(exec.PU_isa1))
    applies to app.thrs;
Allowed_Processor_Binding => (reference(exec.PU_isa2))
    applies to app.dedicatedThr;
Actual_Processor_Binding => (reference(exec.PU_isa1[1]),
                             reference(exec.PU_isa1[4]))
    applies to app.thrs[2];
```

Fig. 3. Software placement (deployment layer B). `exec` is a component of type `mpHardSystem`. `app`, `thrs` and `dedicatedThr` are respectively a process, an array of threads and a singular thread. `thrs[2]` may be scheduled on processor units 1 and 4.

As the processing units of an array are homogeneous, all tasks, which are abstraction of executable codes for an ISA, can be allocated on all of them. The `Allowed_Processor_Binding` property expresses this in the model. Then,

thee `Actual_Processor_Binding` properties allocate threads onto one or a set of processors.

Process and partition allocations The same basic principle of allocation schemes can be applied to processes to define the visibility space of data in the context of partitioned systems. On the model of the Fig. 4, a process `part` may be used by two processing units following the operational mode: in normal mode, only `PU_isa1[0]` has the right to access it, whereas `PU_isa1[1]` is activated in an escape state.

```
system implementation main.redundant
subcomponents
 part : process partition.impl;
 exec : system  mpHardSystem.impl;
modes
 normal    : initial mode;
 redundant : mode;
properties
 Allowed_Processor_Binding => (
  reference(exec.procUnits[0]), reference(exec.PU_isa[1])
  ) applies to part;
 Actual_Processor_Binding => (reference(exec.PU_isa1[0]))
    in modes (normal) applies to part;
 Actual_Processor_Binding => (reference(exec.PU_isa1[1]))
    in modes (redundant) applies to part;
end main.redundant;
```

Fig. 4. Partition allocations (adapted from the OSATE github examples/core-examples/multi-core/simple.aadl)
.

In [3], similar allocation scheme is used to model that an AADL *virtual processor* can be hosted by several processing units. An implementation requirement is that the context can migrate from one processing unit to the other one.

Discussion These examples, i.e. thread allocations in the scheduling field and partition allocations in the Time-Space Partitioned field, are nearly similar, and the modeling approach is homogeneous. The deployment layer represents spatial allocations of the software entities. Temporal intervals where the resources are actually used are not directly specified; property values assigned to hardware resources may reference what are the rules to determine the dynamic sharing of the resource.

At the hardware layer level, the architecture models enumerate the processing units as a "flat" structure; they abstract multi-processing architectures as a set of available implementation-agnostic computing resources.

But a major problem with tightly coupled systems like shared-memory multiprocessors comes from the unexpressed sharing of resources at the level of memory hierarchy. Private cache memories locally separate the information flows near the processing units, but some parts of the memory hierarchy remain shared between processors.

The programming model of shared memory multiprocessor platforms allows communications of data or synchronizations between tasks through memory locations accesses. However, such a functional explicit sharing is generally secondary with respect to the sharing of the main memory banks and buses which provides all the memory words that the program execution requires.

For instance, [4] reports the effect of this resource sharing on a quad core system (Intel Xeon) for programs of the SPEC2006 benchmark: when a core executes a synthetic workload, another program running on the other core may experience an impressive slowdown (up to a slowdown of 200%).

So, an AADL model of shared memory multiprocessor systems must represent the resources shared by the processing units, especially if the software specification does not express this sharing as own.

2 Modeling of the memory hierarchy

The memory hierarchies can be complex subsystems, but they mostly expose, to the software, the interface of a simple memory[3]. Hence, designers must take attention to memory hierarchy as its non-functional characteristics are significant.

The guidelines we propose are based on the modeling of the memory hierarchy as a tree. The terminal nodes represent the processing units and the nodes abstract the different sub-parts of the memory system. A node of the memory hierarchy is characterized by the visibility of the memory words it contains; a node is shared by the same set of processing entities. Data exchanges within the memory hierarchy follow the paths of the tree.

The following rules may be applied to build the model:

- The memory entities used by a same set of processing units are declared in a `system` component, as sub-components. If the memory entity is unique, a single `memory` component may be substituted to the `system` component.
- A `system` component groups, as sub-components, a memory system, and the processors and upper level memory systems which share the access to this memory system.
- For the sake of simplicity, levels of memory hierarchy connected to only one processor, i.e. processor's private caches, may be modeled as sub-components of this `processor` component.
- The memory system associated to the root node contains at least a memory component representing the main memory.

[3] We consider that the program or the operating system are aware of memory coherency problems.

When required for analyzing purposes, processors and composite memory components may detail their internal structure. AADL bus connection features can be used for that purpose. Information about the behavior of the each memory entities are given by AADL properties or AADL memory classifiers, for instance for distinguishing the main memory and the cache levels.

The Fig. 5 shows the modeling of the tree with a hierarchy of AADL components. The memories associated to the M_y sub-system are used or shared by P_j and P_k processors. M_x is the root memory system, including all the data that could be addressed in the physical address space. The M_x memory is shared by the P_i processor and M_y memory sub-system.

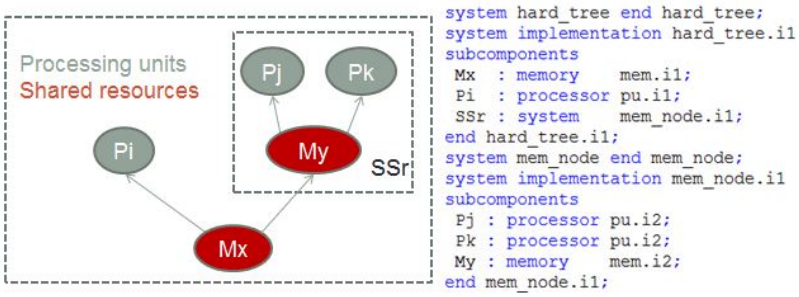


Fig. 5. "Sharing" tree and its AADL representation.

3 Example

As an example, we apply the guidelines that we propose on the processor of a VPX board from the company *InterfaceConcept*. The board VPX3a includes an Intel *core i7*, a FPGA Xilinx Kintex7 and an IO Bridge. The *core i7 2566LE* processor contains two cores, with an optional activation of multi-threading capabilities. Three levels of caches constitute the memory hierarchy; level 1 is composed of two separated caches for instructions and data, level 2 and 3 are unified ones. Only the level 3 of cache is shared between cores.

The Fig. 6 shows an high level model of the hardware architecture for the VPX3a board, with emphasis on the processing units and the memory hierarchy. We assume that the hyper-threading is enabled on the core number two.

The Fig. 7 and 8 show the structure of the memory hierarchy nodes. AADL bus connections have been used to model the internal hierarchy of the implementation `L1I_L1D_L2` of `mem_system`. A similar modeling method can be applied to the other memory systems.

Notice that the cache sharing may change during the design process, even on a given execution platform. The activation of the multi-threading on a core or

```
system implementation exec.IC_INT_VPX3a
subcomponents
  core1 : processor Intel64.core_single_thread;
  core2 : system    Intel64.core_dual_thread;
  mem   : memory    mem_system.L3_main;
end exec.IC_INT_VPx3a;
system implementation Intel64.core_dual_thread;
subcomponents
  pth1 : processor Intel64.physical_thread;
  pth2 : processor Intel64.physical_thread;
  mem  : memory    mem_system.L1I_L1D_L2;
end Intel64.core_dual_thread;
```

Fig. 6. Model of the processor core i7 2566LE. The root subsystem `mem` is used directly by the processor *core1* and communicates with the subsystem *core2*. *core2* contains two physical threads which share the caches L1 and L2.

```
memory implementation mem_system.L3_main
subcomponents
  L3cache : memory cache_system.L3;
  main    : memory shared_memory.main;
end mem_system.l3_main;
processor implementation Intel64.core_single_thread
subcomponents
  L1Icache  : memory cache_system.L1I;
  L1Dcache  : memory cache_system.L1D;
  L2cache   : memory cache_system.L2;
end Intel64.core_single_thread;
```

Fig. 7. Structured memory subsystems and processor's private memory resources. The L2 cache is private to a core when hyper-threading is not activated. The L3 cache and the main memory are shared by all the cores, and then are grouped into a unique node.

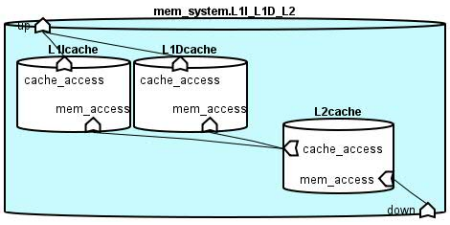


Fig. 8. Internal structure of the L1I_L1D_L2 composite memory system.

the operating system enabling of a level of cache are examples of modifications that a designer could do to explore their impact on the system performance. If such situations may exist, the component hierarchy can contain "potentially shared" nodes in the tree, currently used by only one processing unit.

4 Exploitation of the models

As shared-memory multiprocessor platforms become more complex and widely used, designers of real-time systems need to have access to analysis tools in order to assess their choices or dimension their systems. Such tools must work from an architectural model of the execution platform; we think that the modeling guidelines presented in this article set up a right level of abstraction for this purpose. These AADL models may be used directly as a tool entry or as a source for a model transform whether the tool uses a DSL language.

In the sequel, we give examples of tools which are or could able to handle models compliant with the guidelines proposed in this article.

Scheduling analysis *AADL Inspector* from Ellidiss Technologies is a lightweight framework allowing to apply different analysis on AADL models. One of the analysis tools available with *AADL Inspector* is Cheddar[5], a scheduling analysis tool which deals with global multiprocessor scheduling [6]. *AADL inspector* transforms the significant AADL entities (components and properties) into Cheddar-ADL components, in order to control the scheduling analysis [7]. Today, the model transform does not deal with caches, but Cheddar includes some algorithms to evaluate the cache related preemption cost when instruction caches are defined in an architecture model [8].

Resource allocations The placement of software entities on hardware resources may be complex if the number of tasks or virtual processors is high. The use of optimization methods are sometimes necessary to perform these allocations. For instance, Cheddar implements basic partitioning algorithms such as RM-Best-Fit [9] to automatically allocate tasks to processors.

From an AADL model, `Allowed_Processor_Binding` can be used to supply the set of candidate resources and allocation targets (e.g. processors). The results of partitioning algorithms can then be expressed on the analyzed AADL model by updating its `Actual_Processor_Binding` properties. *AADL Inspector* will integrate such a function in its future releases.

Timing analysis In [10], the timing analysis begins by considering the processor individually and isolated from the other ones. The tool conducts a first computation of WCET, completed by a cache analysis. A multilevel hierarchy of caches is considered at this stage; the hierarchy ends with the first level of shared cache.

Then, the analysis assumes that all the possible conflicts happen with the other processors for the shared cache usage. From this pessimistic analysis, the worst case response times of all the tasks are computed, and the interferences

previously considered are kept if the task's running times intersect. This process is recurrently performed upto a fix point is reached.

These examples of tools and analysis show the interest of modeling multi-processing platform and their memory hierarchy, as many analysis can be conducted from that kind of models.

5 Related works

We describe in this section other modeling approaches used to guide performance analysis on shared-memory multiprocessor systems.

In [1], Paul proposes a software-on-hardware performance modeling environment called MESH that founds the modeling and the simulation of single-chip multiprocessors on a layered approach. A software and a hardware layer respectively include the application threads and the hardware resources. An intermediate layer represents the dynamic mapping of logical threads onto physical resources through the scheduler decisions and associates thread's logical events to physical times. MESH models can include shared resources other than execution units; the simulation kernel applies time penalties of application threads when access contentions are detected.

In the context of shared-memory multiprocessors, [11] abstracts the essential features of the memory hierarchy in a formal model. A 3-tuple $\langle p_l, \alpha_l, \sigma_l \rangle$ characterizes a level l of memory caches, where p_l is the number of cores sharing the level l caches, α_l the number of level l caches, and σ_l the size of one level l cache. From the rules of a peddle game, the authors establish lower bounds on memory traffic between different levels of caches for computations represented as a directed acyclic graph. Next, they base optimizations on this metric to improve the implementation of some parallel algorithms.

The modeling of multicore execution platforms with *Simulink* has been proposed by [12] to optimize the partitioning of tasks in soft real-time systems. The *Simulink* model represents the cores, the core communication costs, and the task set. But, the approach does not take into account the memory hierarchy in the performance analysis; the behavior issued from the sharing of the caches cannot be inferred from this model.

Conclusion and future works

This article presents an approach for modeling shared-memory multiprocessor platforms. Basic entities that we have emphasized in this high-level model, are the processing units, whereas the way they are implemented, and the different levels of the memory hierarchy. We expect the architecture of the memory subsystem is a key feature to perform timing analysis on multi-processing platforms.

The memory hierarchy is not the only challenge in the timing analysis on multiprocessor platforms. Another topic is the multiplication of interconnection buses, with various characteristics in term of throughput and latency. Multiprocessor SoCs use the capabilities of integrated circuit to implement a lot of buses

to connect the different functions available on the chip. One goal devoted to the architectural models will be to document this structure, and guide tools and designers to identify potential bottlenecks.

Another problem is to deal with PHMs. SoCs mix different types of processors, with different speeds or capabilities. How AADL models can represent all these types of information must be investigated.

Acknowledgments This work is done in the context of the SMART project. SMART and Cheddar are supported by the *Conseil Régional de Bretagne*, *Bpifrance*, *Conseil Général du Finistère* and *BMO*. Cheddar is also supported by *Ellidiss Technologies*, *EGIDE/PESSOA n. 27380SA* and *Thales TCS*.

References

1. J. M. Paul, D. E. Thomas, and A. S. Cassidy, "High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors," *ACM Trans. on Design Automation of Electronic Systems*, vol. 10, no. 3, pp. 431–461, 2005.
2. S. Rubini, C. Fotsing, P. Dissaux, F. Singhoff, and H. N. Tran, "Scheduling analysis from architectural models of embedded multi-processor systems," *ACM SIGBED Review*, vol. 11, no. 1, 2014.
3. J. Delange and P. Feiler, "Design and analysis of multi-core architectures for cyber-physical systems," in *Proceedings of the 7th European Congress Embedded Real Time Software and Systems (ERTSS)*, Toulouse, France, Feb. 2014.
4. V. Babka, "Cache sharing sensitivity of SPEC CPU2006," Distributed Systems Research Group, Department of Software Engineering, Tech. Rep., 2009.
5. F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a Flexible Real-Time Scheduling Framework," *ACM SIGAda Ada Letters*, vol. 24, no. 4, Dec. 2004.
6. R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011.
7. P. Dissaux, O. Marc, S. Rubini, C. Fotsing, V. Gaudel, F. Singhoff, and H. N. Tran, "The SMART project: Multi-agent scheduling simulation of real-time architectures," in *Proceedings of the 7th European Congress Embedded Real Time Software and System (ERTSS)*, Toulouse, France, Feb. 2014.
8. H. N. Tran, F. Singhoff, S. Rubini, and J. Boukhobza, "Instruction cache in hard real-time systems: modeling and integration in scheduling analysis tools with AADL," in *Proceedings of the 12th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 14)*, Milan, Italy, August 2014.
9. Y. Oh and S. H. Son, "Tight performance bounds of heuristics for a real-time scheduling problem," *Technical Report CS93-24, University of Virginia.*, 1993.
10. S. Chattopadhyay, A. Roychoudhury, and T. Mitra, "Modeling shared cache and bus in multi-cores for timing analysis," in *Proceedings of the 13th ACM International Workshop on Software & Compilers for Embedded Systems*, 2010, pp. 6–15.
11. J. E. Savage and M. Zubair, "A unified model for multicore architectures," in *Proceedings of the 1st ACM International Forum on Next-generation multicore/-manycore technologies*, 2008, pp. 9–20.
12. J. Feljan, J. Carlson, and T. Seceleanu, "Towards a model-based approach for allocating tasks to multicore processors," in *Proceedings of the 38th EUROMICRO Conference on the Software Engineering and Advanced Applications (SEAA)*, Sep. 2012, pp. 117–124.

Executable AADL

Real Time Simulation of AADL Models

Pierre Dissaux[1], Olivier Marc[2]

[1]Ellidiss Technologies, Brest, France.
[2]Virtualys, Brest, France.
pierre.dissaux@ellidiss.com
olivier.marc@virtualys.com

Abstract. The Architecture Analysis and Design Language (AADL) standard [2] defines a default runtime semantic for software intensive Real Time systems. This includes support for multi tasking, network distributed architectures and Time and Space Partitionning systems. A proper implementation of the AADL runtime thus allows for the virtual execution of a system at a model level and contributes to the early verification of critical software applications.
This paper describes an implementation of the AADL runtime by the Marzhin Multi Agent simulator that is embedded in the AADL Inspector tool [5].

Keywords. AADL, Simulation, Multi Agent

Introduction

The Architecture Analysis and Design Language (AADL) standard defines a default runtime semantic for software intensive Real Time systems. This includes support for multi tasking, network distributed architectures and Time and Space Partitionning systems (TSP). A proper implementation the AADL runtime thus allows for the virtual execution of a system at a model level and contributes to the early verification of critical software applications in the development life-cycle.

This paper firstly summarizes the definition of the default AADL runtime, then describes one of its implementations that has been performed to develop the Marzhin Multi Agent simulator, and finally explains how it can be used in practice within the AADL Inspector tool.

1 The AADL Runtime

The AADL Language is an international standard of the SAE (AS-5506B) [1]. The standard defines in particular a default execution model that specifies the way the various components interact at run-time. This enables a precise timing analysis and simulation of AADL models.

A typical AADL model is composed of one or several execution resources (Processors) that can communicate via Buses. The software application is composed of one or several memory address spaces (Processes) that contain concurrent Threads and shared Data. Various inter-threads communication paradigms are supported.

1.1 Processors

In AADL, the Processor represents the association of a hardware computation resource and a scheduler. It must declare a Scheduling_Protocol property whose value corresponds to one of those that are actually supported by the analysis, simulation or code generator. Typically supported Scheduling_Protocols are:

- Rate Monotonic protocol (RM), based on the period of the Threads.
- Deadline Monotonic protocol (DM), based on the deadline of the Threads.
- POSIX 1003 (HPF), based on the predefined priority of the Threads.
- ARINC 653, for the static scheduling of partition slots.

In the case of a partitioned system, the Processor computation resource is shared between several Virtual Processors, each of them being associated with a set of Threads located in an AADL Process. Virtual Processors must also define their own Scheduling_Protocol property. This is typically what happens when the ARINC 653 Annex of the AADL standard is used.

1.2 Threads

The default behavior of AADL Threads is specified in the standard by a state-transition automaton.

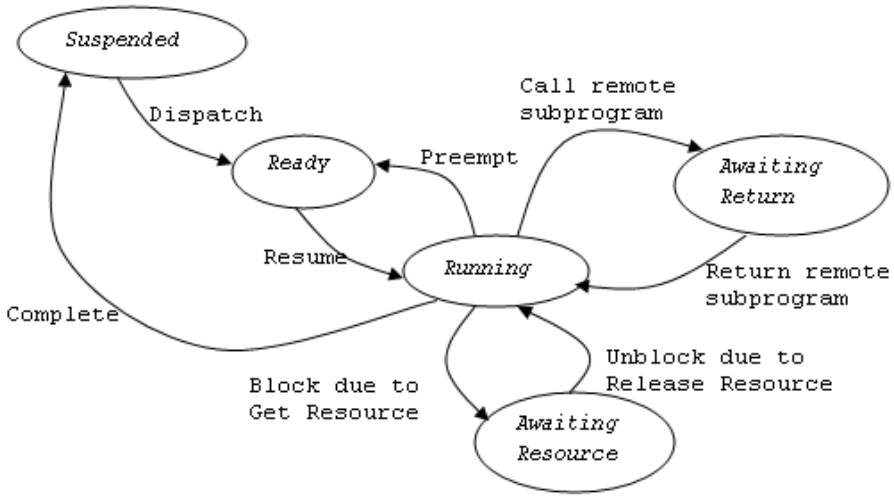


Fig. 1. Runtime states for AADL threads

A Thread must have a Dispatch_Protocol property that defines when it is ready to execute. Supported protocols are:

- Periodic: the thread is periodically triggered by a system clock.
- Aperiodic: the thread is triggered upon arrival of an event on one of its ports.
- Sporadic: same as Aperiodic with a minimum inter-arrival time.
- Timed: same as Aperiodic with an additional timeout event.
- Hybrid: the thread is triggered by event ports and the system clock.
- Background: the thread is triggered when the execution resource is free.

Thread interfaces contain Features that are used to implement communication channels. They can be:

- Data Ports: allows for point to point data exchange
- Event Ports or Event Data Ports: allows for events and message exchange
- Access to shared Data: allows for multi-points data exchange with concurrency control.
- Access to remote Subprograms: allows for remote subprogram calls.

1.3 Shared Data

One particular way to exchange information between Threads is to let them have access to the same shared data. Shared data are represented in AADL by Data subcomponents to which Threads can have access through Data Access Connections.

It is possible to specify critical sections thanks to the AADL Behavior Annex. In order to ensure mutual exclusion of all the threads accessing a given shared data component, a Concurrency_Control_Protocol property can be set. A typical value for this property is Priority_Ceiling_Protocol (PCP).

1.4 AADL Behavior Annex

The core definition of AADL deals with the architectural description of the system. It specifies which components are instantiated and how they are connected and bound together. The functional activity of Threads or Subprograms is summarized by a Compute_Execution_Time property that must be given with its Min and Max values. The Max value of this property thus corresponds to the usual WCET (Worst Case Execution Time) that is used for scheduling analysis.

However, in order to perform precise timing analysis or simulations, it is necessary to provide a more detailed description of the functional behavior of Threads and Subprograms. The AADL Behavior Annex is an action language that can be used to provide a simplified representation of the sequential source code structure (pseudo-code).

Examples of actions that can be defined with the AADL Behavior Annex are:

- `p!` : sending an event on port p (Put_Value and Send_Output)
- `d!<` : entering a critical section on shared data access d (Get_Resource)
- `d!>` : leaving a critical section on shared data access d (Release_Resource)

- `computation(a..b)` : use of the processor for a duration between the minimum duration a and the maximum duration b.

2 The Marzhin Simulator

2.1 Principles of the Marzhin Simulation.

Marzhin is a simulation engine that is based on a multi-agent kernel which implies a random order of activation of the execution units. Each agent can contain one or more execution units that are invoked randomly during a simulation cycle. The agents can be specified independently which highly facilitates the initial development and the maintenance of the simulator. At the execution time, all the agents interact together to exhibit a global behavior.

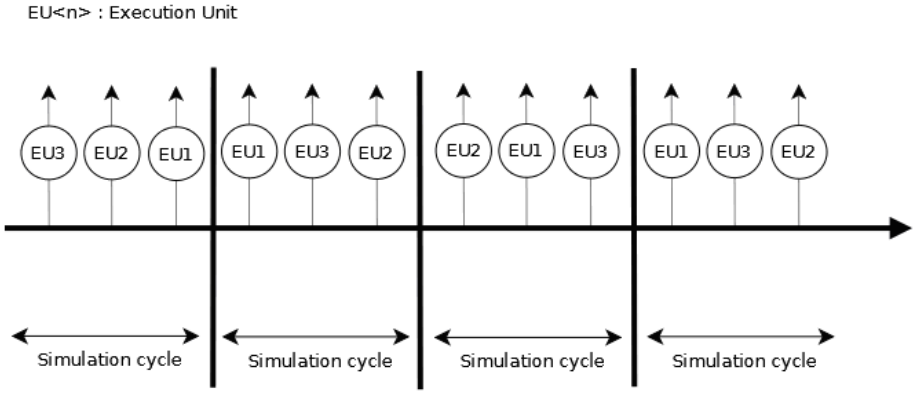


Fig. 2. Marzhin random execution principle

Each AADL entity that is managed by Marzhin is modeled by a specialized agent containing the appropriate execution units to reflect the AADL runtime semantics.

2.2 The Marzhin Data Model.

The Marzhin agents have been defined so that they usually match the corresponding AADL entities. However, in order to well distinguish them from their AADL equivalent, the name of the Marzhin entities has been capitalized in the next paragraphs:

- THREAD: It has the Dispatch_Protocol, DispatchOffset, DispatchJitter, Period, Priority, Deadline and Quantum properties. ThePriority is generally determined by the Scheduling_Protocol property of the PROCESS containing the THREAD. It maintains a list of instructions to be executed. They can be for example: event sending or conditional branches. Their different execution states are the same as those of the AADL Threads as described in section 1.2.

- PROCESS: It represents the address space partitionning and the scheduler. In particular, it thus implements the Scheduling_Protocol that is specified by the AADL Processor.
- PROCESSOR : it contains PROCESSes and manages their scheduling in case of a multi-partition system.

2.3 Marzhin Simulation Cycle.

In the case of the execution of THREADs in a PROCESS, the Marzhin simulation cycles run as follows:

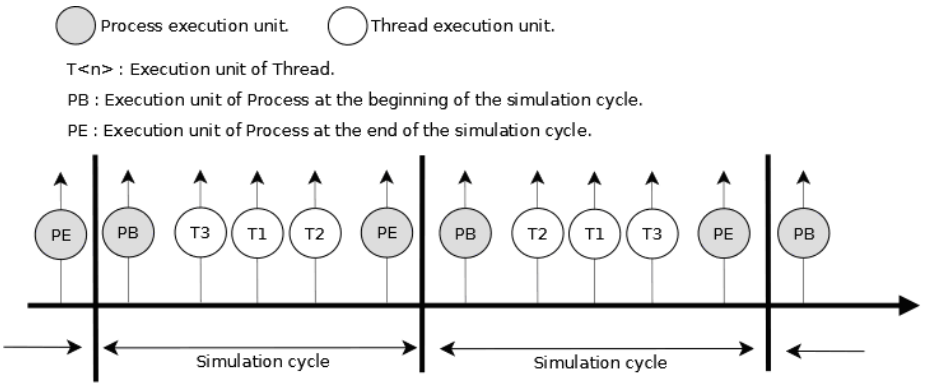


Fig. 3. Marzhin simulation cycle

1. An execution unit starts the simulation cycle of the PROCESS (PB). This allows for updating the priority of each THREAD at each simulation cycle if needed.
2. Execution in a random order of the election process for all the THREADS (T1, T2, T3) in order to update their current internal state and determine the highest priority THREAD that will be executed.
3. An execution unit ends the simulation cycle of the PROCESS (PE) and actually executes the current instruction of the selected THREAD.

In the case of a PROCESSOR containing several PROCESSes, the execution is defined according to the partition slots. If a partition is not active, all the execution units of involved entities (PROCESSs, THREADs ...) are disabled and are not taken into account in the simulation cycle. Only the execution units of the active partition will be activated during the cycle.

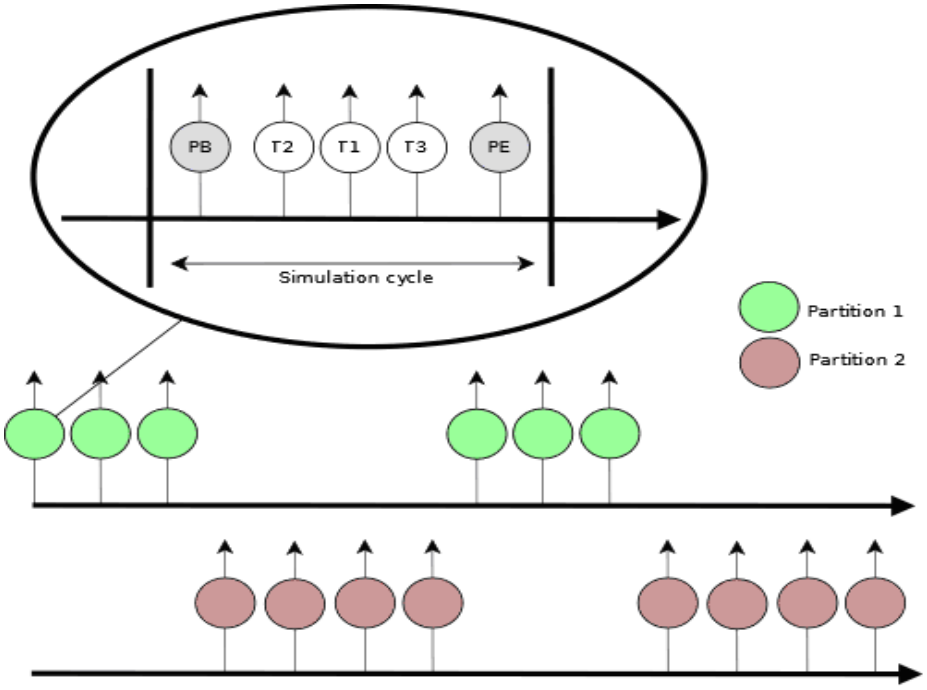


Fig. 4. Simulation of a multi-partition system

2.4 About Determinism in Marzhin.

Despites the intrinsic randomness of the Marzhin simulator, a deterministic behavior is observed most of the times, thanks to the rigorous management of the THREAD priorities. However, in some situations, it becomes possible to introduce a certain level of non-determinism that can be useful for analysis purposes.

In the example below, randomness occurs with a Rate Monotonic scheduler when several threads have the same period and therefore have the same priority:

Simulation configuration:

```
process1 : RATE_MONOTONIC_PROTOCOL
thread1  : DispatchProtocol=PERIODIC Period=10 WCET=3
thread2  : DispatchProtocol=PERIODIC Period=10 WCET=3
thread3  : DispatchProtocol=PERIODIC Period=10 WCET=3
```

Simulation trace:

```
THREAD process1.thread3  __|_|___|._||__|....|__|____|.
THREAD process1.thread2  _|_|_|....|_____||..._|____||..
THREAD process1.thread1  |_____||..___||___|.__|_||....
```

Caption:

```
. : THREAD_STATE_SUSPENDED
| : THREAD_STATE_RUNNING
_ : THREAD_STATE_READY
```

During the simulation cycle 0, the random routine selected thread1 whereas it is thread2 in cycle 1, and so on. It is however possible to control this non-determinism thanks to the Quantum and Dispatch_Order attributes. Quantum specifies the minimum amount of time the currently selected THREAD will remain active without being prempted and Dispatch_Order indicates how the current THREAD is selected within the list (FIRST, LAST or RANDOM). The same example with a Quantum set at 3 and a Dispatch_Order set at FIRST gives the following simulation trace:

```
THREAD process1.thread3  ______|||._____|||._____
THREAD process1.thread2  ___|||...___|||...___||
THREAD process1.thread1  |||.......|||.......|||..
```

The non-determinism of Marzhin can also be beneficial to manage the Global Asynchronism of the simulation environment. It is thus possible to inject events or update data values in incoming ports connected to remote input devices such as the operator keyboard, a dedicated dialog box or an active widget in a 3D virtual reality simulation.

The following example shows how an event can dynamically influence the behavior of the simulation. The periodic THREAD thread1 sends an event to the sporadic THREAD thread2. Such an event could also come from external interface of the simulator:

```
process1 : RATE_MONOTONIC_PROTOCOL
thread1  : DispatchProtocol=PERIODIC Period=10 WCET=5
thread2  : DispatchProtocol=SPORADIC Period=4  WCET=3

EVT IN process1.thread2.evt .....................11......
THREAD process1.thread2     .....................|||....
THREAD process1.thread1     .|||||.....|||||.....|___||||
```

Caption:

```
1 : number of events in the incoming port queue.
```

3 Virtual Execution of AADL Models

3.1 AADL Inspector

AADL Inspector is a model processing framework composed of an AADL toolbox and a customizable set of plugins. The AADL toolbox includes an AADL parser and the LMP (Logic Model Processing) model processing environment [4] that is based

on the use of the prolog language. The LMP engine is used to perform queries on the AADL declarative and instance models, to implement static model checkers and to develop model transformations.

For Real Time analysis, two plugins are currently embedded in AADL Inspector: Cheddar [1] that implements feasibility tests and a static simulator, and Marzhin for dynamic simulation. The static simulator graphically reflects the deterministic outcome of the scheduling analysis, whereas the dynamic simulator exhibits the behavior of the multi-agent engine execution. The result of both simulators is displayed graphically in an advanced time lines viewer.

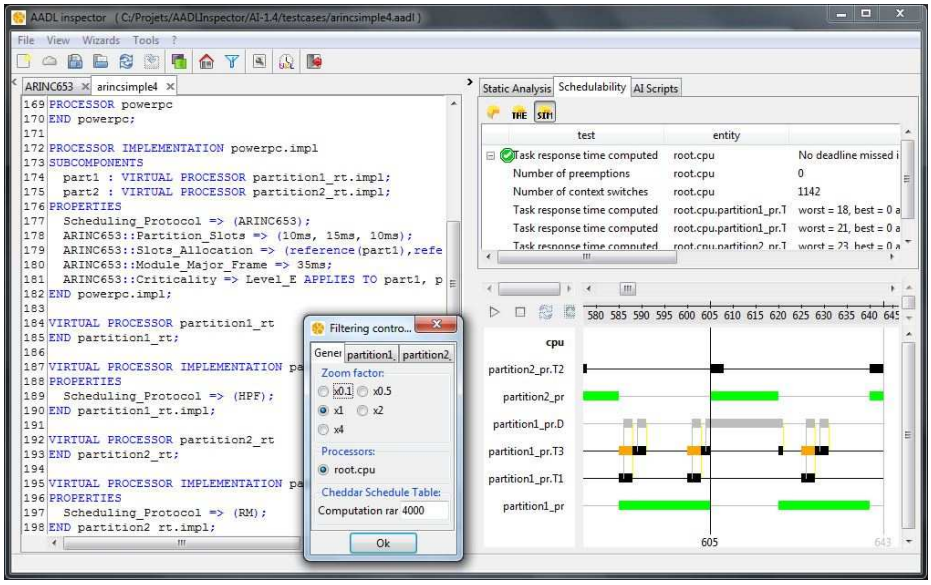


Fig. 5. AADL Inspector 1.4

Thanks to AADL Inspector, it is thus possible to load a complete AADL project distributed on several files containing textual declarative statements, to analyse it in order to build the corresponding instance model, to perform the proper model transformation so that it can be processed by Marzhin, and to pilot its virtual execution through a control panel.

3.2 Executing AADL models

Such a virtual execution of AADL models can efficiently complements the use of more formal real time analysis tools such as Cheddar, as it does not require the input model to satisfy restricted assumptions. It thus significantly extends the scope of model driven real time analysis, especially in the direction of non-periodic activities.

Another use of virtual execution is to perform architecture trade-off studies by providing an immediate feedback showing the coarse grain dynamic behavior of the system during the design phases.

Finally, the specific technical approach that has been chosen for the implementation of Marzhin enables an easy interaction with an asynchronous environment, such as a human operator or a virtual reality simulation.

This approach can be operated early in the development process of the system to support system and software real-time design activities, before the software coding phases. Although the AADL Behavior Annex is used to describe the concurrent aspects of the system behavior, purely procedural actions are still expressed by their computation time. Further work would be required to investigate the ways to enrich this approach with automatic code generation capabilities.

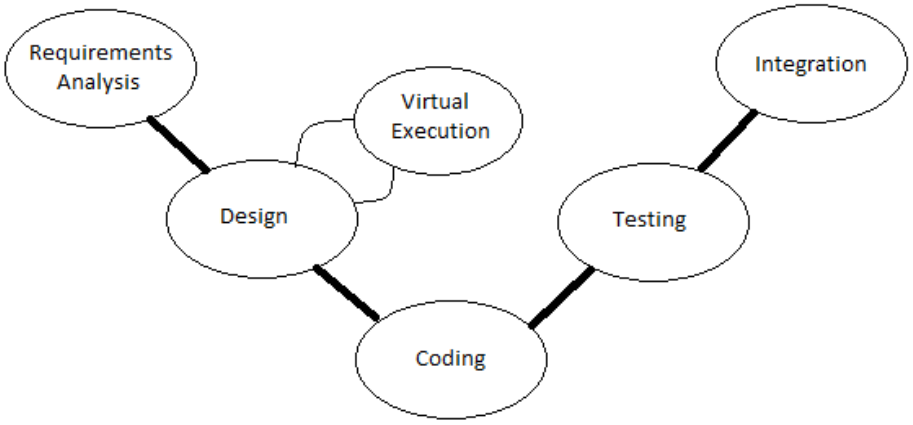


Fig. 6. Use of Virtual Execution in the development life cycle

Conclusion and Future Work

The current implementation of the Marzhin simulator that is available as a part of the AADL Inspector tool already supports a comprehensive subset of the AADL runtime semantics that enables virtual execution of models for the purpose of Real Time analysis, exploration of design solutions and early demonstration of the behavior of a future system.

This work is partly realized in the context of the SMART project [3] in collaboration with the University of Brest and with the financial support of the Council of Brittany, the Council of Finistère, BMO and BPI France.

The future improvements that are foreseen for this activity concern a more complete implementation of the AADL Behavior Annex, improved support of distributed systems and investigations around the possible benefit of the approach for system safety analysis with a proper use of the AADL Error Annex. An additional topic could be studying the possible implications for automatic code generation.

References

1. F. Singhoff, J. Legrand, L. Nana, L. Marcé. “Cheddar: a Flexible Real-Time Scheduling Framework”, ACM SIGAda Ada Letters, 24(4):1-8, ACM Press. 2004
2. SAE International. “Architecture Analysis and Design Language (AADL)”, AS5506B. 2012
3. P. Dissaux, O. Marc, S. Rubini , C. Fotsing, V. Gaudel, F. Singhoff, A. Plantec, Vương Nguyễn-Hồng, Hải Nam Trần. “The SMART Project: Multi-Agent Scheduling Simulation of Real-time Architectures”, Proceedings ERTS conference. 2014.
4. P. Dissaux, P. Farail. “Model Verification: Return of experience”, Proceedings ERTS conference. 2014.
5. Ellidiss Technologies. AADL Inspector site: http://www.ellidiss.fr

Automatic Derivation of AADL Product Architectures in Software Product Line Development

Javier González-Huerta, Silvia Abrahão, Emilio Insfran

ISSI Research Group, Universitat Politècnica de València
Camino de Vera, s/n, 46022, Valencia, Spain
{jagonzalez, sabrahao, einsfran}@dsic.upv.es

Abstract. Software Product Line development involves the explicit management of variability that has to be encompassed by the software artifacts, in particular by the software architecture. Architectural variability has to be not only supported by the architecture but also explicitly represented. The Common Variability Language (CVL) allows to represent such variability independently of the Architecture Description Language (ADL) and to support the resolution of this variability for the automatic derivation of AADL product architectures. This paper presents a multimodel approach to represent the relationships between the external variability, represented by a feature model, and the architectural variability, represented by the CVL model, for the automatic derivation of AADL product architectures through model transformations. These transformations take into account functional and non-functional requirements.

Keywords: AADL; Software Product Line Development; Architecture Derivation; Variability Representation.

1 Introduction

Software Product Line (SPL) development is aimed to support the construction of a set of software products sharing a common and managed set of features, which are developed from a common set of core assets in a prescribed way [1]. Thus, in SPL development variability must be defined, represented, exploited and implemented [2]. The external variability (relevant to customers), usually represented by a feature model [3], should be realized by the internal variability (relevant to developers) of the software assets used to build up each individual software product [4].

Software Architecture is a key asset in SPL development and plays a dual role: on the one hand the product line architecture (PLA) should provide variation mechanisms that help to achieve a set of explicitly allowed variations and, on the other hand, the product architecture (PA) is derived from the PLA by exercising its built-in architectural variation points [1].

In order to enable the automatic resolution of the PLA variation points is required not only that the architectural description languages provide variation mechanisms, but also to explicitly represent how the different variants realize the external variability usually represented in feature models.

Although AADL [5] incorporates different variation mechanisms that allow describing variability in system families [6], the explicit representation of the variation points and its variants is also required to cope with the problem of product configuration and architecture derivation.

To tackle this problem, in previous works, we presented the quality-driven product architecture derivation, evaluation and improvement (QuaDAI) method [7], which uses a multimodel to guide the software architect in the derivation, evaluation and improvement of product architectures in a model-driven software product line development process.

The Common Variability Language (CVL) [8] is a language that allows the specification of variability over any EMF-based, and supports the resolution of the variability and automatic derivation of resolved models. CVL incorporates its own variability mechanisms (e.g., fragment substitution) that can be used to extend those provided by the ADLs.

In this paper, we extend the multimodel approach to incorporate the explicit representation of the architectural variability, by using CVL, and to establish relationships among architectural variants and variation points with: i) the features that represent the SPL external variability; ii) the non-functional requirements and iii) the quality attributes. Once the application engineer has selected the features and non-functional requirements (NFRs) and the priorities of the quality attributes that together conform the product configuration, the relationships defined in the multimodel allow us to automatically derive the product AADL specification from the PLA.

The remainder of the paper is structured as follows. Section 2 discusses existing approaches that deal with the explicit representation of architectural variability and the derivation of product architectures in SPL development by using CVL. Section 3 presents our approach for the derivation of AADL product architectures by introducing the explicit representation of the architectural variability with CVL. Finally, Section 4 drafts our conclusions and final remarks.

2 Related work

AADL incorporates different architectural variation mechanisms that support the development of system families (e.g., multiples implementation of a system specification, component extension or the configuration support through alternative source code files) [6]. However, in a real SPL scenario is difficult to manage the derivation of the architectural specification of a product (PA), especially when the SPL allows a wide range of variability. To cope with the problem, in the last years, several approaches had been presented that support the representation of architectural variability and the derivation of product architectures in SPL development by using CVL (e.g., [9], [10], [11]).

Nascimento et al. [10] present an approach for defining product line architectures using CVL. They apply the Feature-Architecture Mapping Method (FArM) to filter the feature models in order to consider only the architectural-related features. These features will form the CVL specification that will allow obtaining the COSMOS* architectural models. They do not define relationships between the external variability model

(features model) and the architectural variability expressed in CVL and thus the derivation of the product architecture taking as input the configuration is not supported. They explicitly omit the non-functional requirements when applying the FArM method.

Svendsen et al. [11] present the applicability of CVL for obtaining the product models for a Train Control SPL that are defined using a DSL. They only consider the explicit definition of the internal variability and consequently, the configuration should be made directly over the CVL specification of the internal variability.

Combemale et al. [9] present an approach to specify and resolve variability on Reusable Aspect Models (RAM), a set of interrelated design models. They use CVL to resolve the variability on each model and then compose the corresponding reusable aspects by using the RAM weaver. They also consider just the internal variability, and the configuration should be made over the CVL specification.

In summary, none of the aforementioned approaches establish relationships among the SPL external variability and the architectural variability, even though some of them acknowledge that is a convenient practice in variability management [9]. Establishing connections between the SPL external variability, expressed by means of feature models, the non-functional requirements, represented in the quality model, and the architectural variability, represented by using CVL allows us:

i) To configure the product by using the feature model and the quality viewpoint;
ii) To check its consistency by using the constraints defined on the features model, on the quality viewpoint and on the multimodel;
iii) To solve the architectural variability automatically by using model transformations.

3 A Multimodel Approach for the Derivation of AADL Product Architectures

QuaDAI is an approach for the derivation, evaluation and improvement of product architecture that defines an artifact (the multimodel) and a process consisting of a set of activities conducted by model transformations. QuaDAI relies on a multimodel [12] that allows the explicit representation of different viewpoints of a software product line and the relationships among them. In this section, we focus on the representation of the architectural variability, its resolution and the derivation of the software architecture of the product under development.

3.1 Illustrating Example

The approach is illustrated through the use of a running example: a SPL from the automotive domain that comprises the safety critical embedded software systems responsible for controlling a car. This SPL is an extension of the example introduced in [13], and was modified in order to apply, among others the variation points described in [14].

This SPL comprises several features such as Antilock Braking System, Traction Control System, Stability Control System or Cruise Control System. The Cruise Control System feature incorporates variability. This variability is resolved depending on other selections made on the features model (i.e., the selection of the cruise control together with the park assistant implies the positive resolution of an extended version of the cruise control[1]). Fig. 1 shows an excerpt of the features model that represents the SPL external variability.

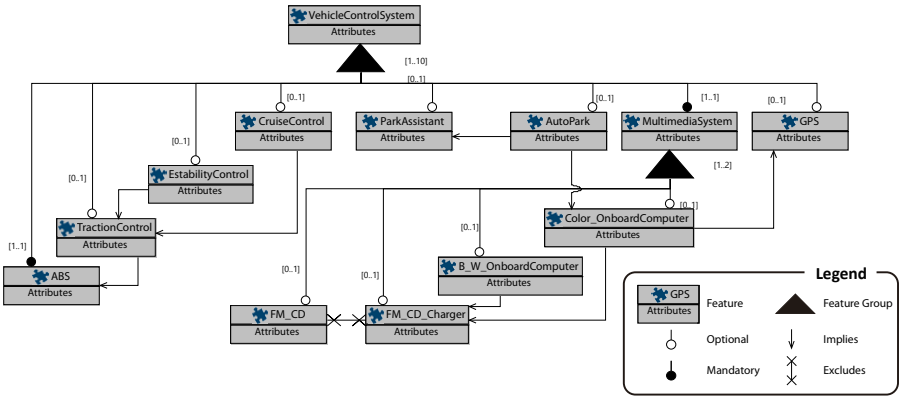


Fig. 1. SPL External Variability

3.2 A Multimodel for Representing Architectural Variability

In QuaDAI, a multimodel permits the explicit representation of relationships among entities in different viewpoints. A multimodel is a set of interrelated models that represent the different viewpoints of a particular system. A viewpoint is an abstraction that yields the specification of the whole system restricted to a particular set of concerns, and it is created with a specific purpose in mind. In any given viewpoint it is possible to produce a model of the system that contains only the objects that are visible from that viewpoint [16]. Such a model is known as a viewpoint model, or view of the system from that viewpoint. The multimodel permits the definition of relationships among model elements in those viewpoints, capturing the missing information that the separation of concerns could lead to [12].

The problem of representing and automatically resolving the architectural variability taking into account functional and non-functional requirements requires (at least) three viewpoints:

- **The variability viewpoint** represents the SPL external variability expressing the commonalities and variability within the product line. Its main element is the feature, which is a user-visible aspect or characteristic of a system [3] and it is expressed by means of a variant [15] of the cardinality

[1] The whole specification of the example is available at http://users.dsic.upv.es/~jagonzalez/CarCarSPL/links.html

based feature model (shown in Fig. 1).

- **The architectural viewpoint** represents the architectural variability of the Product Line architecture that realizes the external variability of the SPL expressed in the variability viewpoint. It is expressed by means of the Common Variability Language (CVL) and its main element is the Variability Specification (VSpec). We only represent in the multimodel the PLA architectural variability, the PLA is represented in an AADL base model, which is referenced by the CVL specification. A base model, under the CVL terminology, is a model on which variability is defined using CVL [8]. The base model is not part of CVL and can be an instance of any metamodel defined via MOF [8]. Fig. 2 shows an example of the CVL variability specification on an AADL base model, allowing to express whether some AADL elements should exist or not in the resolved model (e.g., the ABS) and to configure some internal values (e.g., the range value assignment).
- **The quality viewpoint** represents the hierarchical decomposition of quality into sub-characteristics, quality attributes, metrics and the impacts and constraints among quality attributes. Is expressed by means of a quality model for software product lines [17], which extends the ISO/IEC 25010 (SQuaRE) [18] and allows the representation of NFRs as constraints affecting characteristics, sub-characteristics and quality attributes.

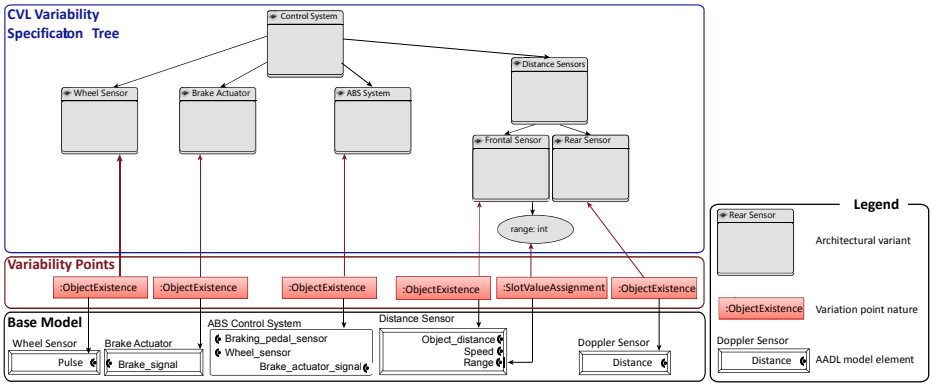


Fig. 2. CVL Variability Specification on an AADL Base Model

The multimodel also allows the definition of relationships among elements on each viewpoints with different semantics as *is_realized_by* [19] or *impact* relationships [12]. An excerpt of these relationships is shown in Fig. 3.

We can describe in the multimodel: i) how the ABS feature *is_realized_by* a set of VSpecs (e.g., the WheelRotationSensor); ii) how the user_safety NFR *is_realized_by* a set of features (e.g., the ABS or the Stability Control); iii) how the selection of a given feature VSpec impacts positive or negatively on a quality attribute; or iv) how the positive resolution of a given VSpec impacts positive or negatively on a quality attribute. These relationships are used to check the consistency of the configuration and to decide

which variation points should be resolved positively in the CVL resolution model driving the derivation of the AADL product architecture.

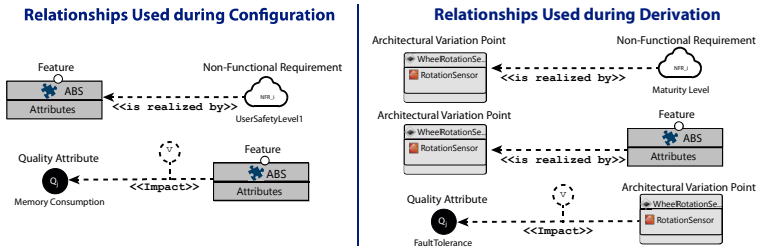


Fig. 3. Multimodel Relationships

3.3 Automating the Derivation of Product Architectures

The QuaDAI derivation process for obtaining AADL product architectures based on the functional and non-functional requirements comprises two main activates: the *product configuration* and the *architecture instantiation*. Fig. 4 shows an excerpt of the specification of the activities with the main input and output artifacts using the Software & Systems Process Engineering Meta-Model (SPEM) [20].

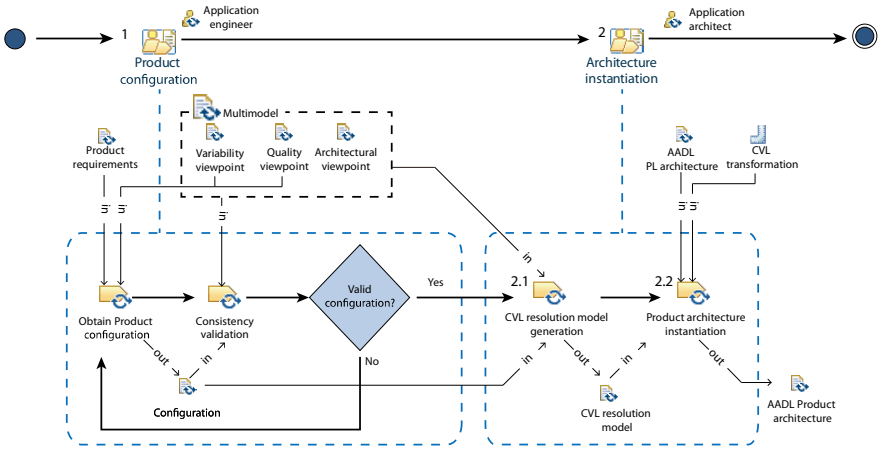


Fig. 4. SPEM specification of the QuaDAI Derivation Process

In the *product configuration* activity, the application engineer selects the features and NFRs that the product must fulfill and establishes the quality attributes priorities in the *obtain product configuration* task. Those priorities will be used during the derivation to choose from a set of architectural variants that having the same functionality differ in their quality attribute levels. In the *consistency validation* activity, first we check the variability viewpoint consistency (i.e., whether the selection of features fulfills the constraints defined in the feature model) and the quality viewpoint consistency (i.e., whether the priorities of the quality attributes defined in the configuration satisfy

the impact and constraint relationships among them defined in the quality viewpoint). Once the intra-viewpoint consistency is satisfied we check the inter-viewpoint consistency (i.e., whether the configuration satisfy the *is_realized_by* and *impact* relationships defined among elements on different viewpoints). The multimodel relationships had been formalized and operationalized in OCL and are checked at runtime by using the OCLTools validator [21]. This consistency checking mechanism allows us to assure that the product configuration meets the SPL constraints facilitating the architecture instantiation activity that focus on the resolution of the PLA architectural variability.

In the *architecture instantiation* activity, the application architect generates the AADL product architecture by means of two model transformation activities. The first transformation, *CVL resolution model generation*, takes as input a valid product configuration and the multimodel (i.e., the relationships between the architectural viewpoint with variability and the quality viewpoint) and, through a Query View Transformation (QVT) [22] model transformation, generates a CVL resolution model. With the multimodel relationships, the QVT transformation decides which architectural variants have to be positively resolved in each variation point. Finally, the *product architecture instantiation* activity, through a CVL transformation, takes as input the CVL resolution model and generates the AADL product architecture. This AADL product architecture represents the resolution of the PLA architectural variability taking into account not only the functional requirements and non-functional requirements selected in the configuration. The use of CVL alleviates part of the computational complexity since it allows us, at design time, to describe the PLA architectural variants and, in derivation time, to only focus on the resolution of the PLA architectural variability. Fig. 5 shows an outline of the AADL architecture derivation with the CVL resolution model generation and the AADL Product architecture instantiation.

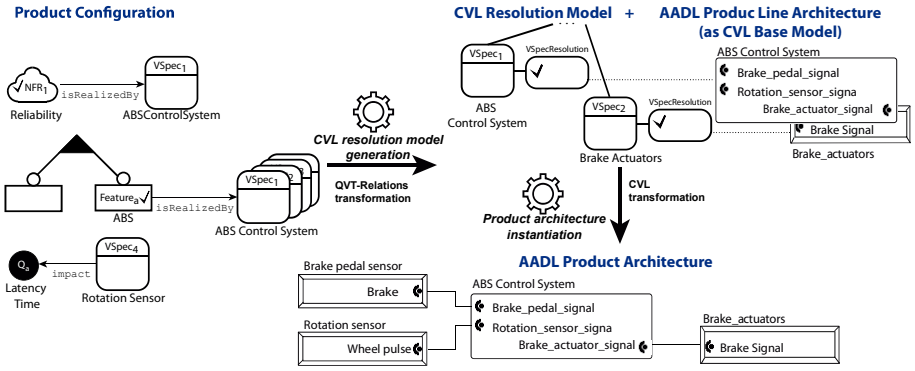


Fig. 5. AADL Product Architecture Instantiation

3.4 Tool Support

The approach is supported by a prototype[2] that gives support to the configuration, consistency checking and generation of the CVL resolution model. The prototype allows importing feature models and CVL specifications defined using third party tools and to establish the relationships among them described in the paper so as to automate the AADL product architecture derivation.

The variability viewpoint consistency checking is carried out by transforming the feature model and the features selection to the FAMA [23] metamodel and by calling the FAMA validator. The quality viewpoint and the inter-viewpoint consistency checking are carried out through OCL constraints checked at runtime by the OCLTools validator. The tool is also capable of calling the QVT transformation that generates the CVL resolution model.

Fig. 6(a) shows the call to the CVL resolution creation functionality. Fig. 6(b) shows the resulting CVL resolution model when for a configuration formed by the feature configuration *features*={`Vehicle Control System, ABS, TractionControl, StabilityControl and FM_CD`} (see Fig. 1) and the NFRs configuration *nfrs*={`EuroNCAP`[3]}.

Finally, with the integration of the CVL supporting tool [24] the CVL transformation can be called so as to generate the resulting AADL product architecture.

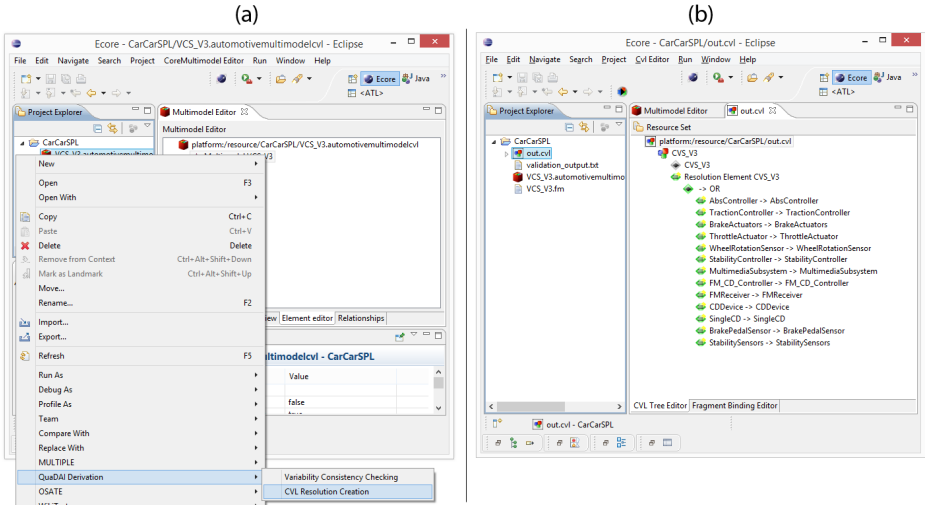


Fig. 6. Generation of the CVL Resolution Model with the Prototype

[2] The prototype is available for download at: http://users.dsic.upv.es/~jagonzalez/CarCarSPL/index.html

[3] EuroNCAP is a voluntary EU vehicle safety rating system. In our example, the EuroNCAP NFR is realized by the ABS, the Traction Control, and the Stability Control features.

4 Conclusions and Further Works

In this paper, we have presented an approach to explicitly represent architectural variability on AADL architectural models by using CVL. We include the architectural variability in a multimodel in which we also represent the SPL external variability in a feature model, and the non-functional requirements in a quality model. In this multimodel, we can establish relationships among elements on the CVL model, the feature model and the quality model. This information is used to drive the model transformation that resolves the architectural variability and obtains the AADL product architecture. The approach is supported by a tool with which the user can edit a product configuration, check its consistency and automatically derive the CVL resolution model. The CVL resolution models allow us to obtain the AADL product architecture by using the CVL supporting tool.

In this work, we apply model-driven engineering principles to provide a feasible solution to an open, complex, error-prone and time-consuming problem in the software product line development community, which is the derivation of product architectures takin into account functional and non-functional requirements.

As further work, we plan to empirically validate the approach through controlled experiments and case studies. We plan also to analyze how to incorporate more powerful CVL variation mechanisms (i.e., the use of VInterfaces that can be used to configure CVL configuration units) and its possible use in combination with the AADL syntax. Finally, although the approach has been initially defined for working together with AADL, the use of CVL isolates the approach from the architectural description language and we want to analyze its applicability to other MOF-based ADLs.

Acknowledgements. This research is supported by the Value@Cloud project (MICINN TIN2013-46300-R) and the ValI+D fellowship program (ACIF/2011/235).

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional (2001).
2. Van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer Berlin Heidelberg (2007).
3. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. , CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute, Carnegie Melon University (1990).
4. Pohl, K., Böckle, G., van der Linden, F.: Software product line engineering. Springer, Berlin (2005).
5. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison Wesley (2013).
6. Feiler, P.H.: Modeling of System Families. , CMU/SEI-2007-TN-047, Software Engineering Institute, Carnegie Mellon University (2007).
7. González-Huerta, J., Insfrán, E., Abrahão, S.: Defining and Validating a Multimodel Approach for Product Architecture Derivation and Improvement. 16th International

Conference on Model-Driven Engineering Languajes and Systems. pp. 388–404. , Miami, USA (2013).
8. Object Management Group: Common Variability Language (CVL) OMG Revised Submission. (2012).
9. Combemale, B., Barais, O., Alam, O., Kienzle, J.: Using CVL to operationalize product line development with reusable aspect models. Workshop on Variability Modeling Made Useful for Everyone. pp. 9–14. , Innsbruck, Austria (2012).
10. Nascimento, A.S., Rubira, C.M.F., Burrows, R., Castor, F.: A Model-Driven Infrastructure for Developing Product Line Architectures Using CVL. 7th Brazilian Symposium on Software Components, Architectures and Reuse. pp. 119–128. , Brasilia, Brazil (2013).
11. Svendsen, A., Zhang, X.: Developing a software product line for train control: a case study of CVL. 14th Software Product Line Conference. pp. 106–120. , Jeju Island, South Korea (2010).
12. González-Huerta, J., Insfran, E., Abrahão, S.: A Multimodel for Integrating Quality Assessment in Model-Driven Engineering. 8th International Conference on the Quality of Information and Communications Technology. pp. 251–254. , Lisbon, Portugal (2012).
13. Hudak, J., Feiler, P.H.: Developing AADL Models for Control Systems : A Practitioner ' s Guide. , CMU/SEI-2007-TR-014, Software Engineering Institute, Carnegie Mellon University (2007).
14. Shiraishi, S.: An AADL-based approach to variability modeling of automotive control systems. 13h Model Driven Engineering Languages and Systems. pp. 346–360. , Oslo, Norway (2010).
15. Gómez, A., Ramos, I.: Cardinality-Based Feature Modeling and Model-Driven Engineering : Fitting them Together. International Workshop on Variability Modelling of Software-Intensive Systems. pp. 61–68. , Linz, Austria (2010).
16. Barkmeyer, E.J., Feeney, A.B., Denno, P., Flater, D.W., Libes, D.E., Steves, M.P., Wallace, E.K.: Concepts for Automating Systems Integration. , NISTIR 6928, U.S. Department of Commerce (2003).
17. González-Huerta, J., Insfran, E., Abrahão, S., McGregor, J.D.: Non-functional requirements in model-driven software product line engineering. Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages. pp. 1–6. , Innsbruck, Austria (2012).
18. ISO/IEC: ISO/IEC 25000:2005 Software Engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE, (2005).
19. Janota, M., Botterweck, G.: Formal approach to integrating feature and architecture models. 11th Conference on Fundamenteal Approaches to Software Engineering. pp. 31–45. , Budapest, Hungary (2008).
20. Object Management Group: Software & Systems Process Engineering Meta-Model Specification (SPEM) v2.0. (2008).
21. Eclipse: Eclipse OCL, http://projects.eclipse.org/projects/modeling.mdt.ocl.
22. Object Management Group: Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification. (2008).
23. ISA Research Group: Fama Tool Suite, http://www.isa.us.es/fama/.
24. Haugen, Ø., Moller-Pedersen, B., Olsen, G.K., Svendsen, A., Fleurey, F., Zhang, X.: Consolidated CVL language and tool. , MoSiS Project, D.2.1.4., SINTEF, Univeristy of Oslo (2010).

Towards an Architecture-Centric Approach dedicated to Model-Based Virtual Integration for Embedded Software Systems

Huafeng Yu[1], Jean-Pierre Talpin[2], Sandeep Shukla[3], Prachi Joshi[3], and Shinichi Shiraishi[1]

[1] TOYOTA InfoTechnology Center, U.S.A.
465 N Bernardo Avenue, Mountain View, CA 94043, U.S.A.
huafeng.yu@us.toyota-itc.com
[2] INRIA Rennes - Bretagne Atlantique,
Campus de Beaulieu, 263 Avenue Général Leclerc, 35042 Rennes, France
[3] Virginia Polytechnic Institute and State University
Falls Church Campus, 7054 Haycock Rd., Falls Church, VA 22043, USA

Abstract. Current embedded systems are increasingly more complex and heterogeneous, but they are expected to be more safe, reliable and adaptive. In consideration of all these aspects, their design is always a great challenge. Developing these systems with conventional design approaches and programming methods turns out to be difficult. In this paper, we mainly present the informative background and the general idea of an ongoing yet young research project, including the model-based design and an architecture-centric approach, to address previous challenges. Our idea adopts a formal-methods-based model integration approach, dedicated to architecture-centric virtual integration for embedded software systems, in an early design phase. We thus expect to improve and enhance *Correct By Construction* in the design. The considered formal methods consist of timing specification, design by contracts, and semantics interoperability for models to be integrated in the system. The application domains of our approach include automotive and avionic systems.

Keywords: Virtual integration, model-based design, AADL, timing specification, design by contract, semantics interoperability

1 Introduction

Current embedded systems are increasingly more complex and heterogeneous, but they are expected to be more safe, reliable and adaptive [8] [16]. In consideration of all these aspects, their design is always a great challenge. Complexity in the design and implementation is a common issue for current avionic and automotive systems. In the current system design, verification and validation (V&V) is also a key concern, particularly for safety-critical systems. These systems generally require great V&V effort to avoid unexpected system behavior.

Moreover, the design is expected to be validated as early as possible due to the huge cost of correction in the late-phase implementation. Design validation in an early phase has become one of the key solutions to reduce the overall V&V cost.

In this paper, we mainly present the informative background and the general idea of an ongoing yet young research project, including the model-based design and an architecture-centric approach, to address previous challenges. Our idea adopts an formal-methods-based model integration approach, dedicated to architecture-centric virtual integration for embedded software systems, in an early design phase. By applying formal methods in an early design phase, we expect to improve and enhance *correct by construction*. The formal methods to be considered consist of timing specification, design by contracts, and semantics interoperability for models to be integrated in the system. The application domain of our approach include avionic and automotive systems.

2 Research Challenges

High-level modeling has been widely adopted as a promising solution to address the system complexity issue [33]. High-level modeling languages, such as UML[27], SysML[1] and MARTE[26], have been widely adopted, thanks to its standardization for modeling. AUTOSAR[2] and EAST-ADL[9] are domain-specific languages for automotive systems. AADL[32] (Architecture Analysis and Design Language) is an SAE standard dedicated to architecture description and modeling for avionic and automotive systems. AADL provides an industry standard, textual and graphic notation with precise semantics to model applications and execution platforms and is supported by commercial and open source tool solutions—including Open Source AADL Tool Environment (OSATE) [28]. Matlab/Simulink[21] is a dataflow language for modeling, simulating and analyzing dynamic systems. Modelica[23] is an object-oriented modeling language for component-based complex systems. These high-level languages enables domain specific modeling and analysis of complex embedded systems. SCADE [12] is an integrated design environment dedicated to rigorous design of safety-critical systems[4].

Multi-paradigm modeling

All the languages mentioned previously are considered as candidate languages in high-level modeling for embedded systems. Multi-languages can be used in the same design because of system modeling from different views, for example, software, architecture, etc.; and different purposes, such as analysis, verification, and evaluation. Furthermore, different languages may adopt different formalism, e.g., state machines, dataflow, communicating sequential processes, differential equations, as backstage support. So the first challenge at the modeling language level is how to harmonize multiple paradigm modeling [24] [25] in the same

design, particularly, when we consider a reliable integration followed by using formal techniques for analysis and V&V at the system level.

An avionic co-modeling example. Co-modeling for the system-level design has been explored in [37] [36], where AADL was used to model the architecture part and Simulink was used to model the behavior part of an avionic case study, called simplified Airbus A350 doors management system. However, semantic difference of the two models makes the integration problematic. In order to have a clear and unambiguous integration, a formal model of computation (MoC), called Polychrony [17], was adopted as an intermediate model. This MoC is based on the synchronous/polychronous timing semantics. The later formal analysis, verification, and scheduling were mainly performed on the basis of the same MoC.

Integration frameworks

In Polychrony, the integration is performed at the polychronous MoC level[36]. Polychrony provides model transformations from AADL and Simulink (via GeneAuto[35]) to the polychronous MoC. In order to keep the semantics coherent, both AADL and Simulink models adopt the polychronous semantics. Based on the same polychronous semantics, the composed model can used for analysis, verification, and simulation or be translated into other formal models for formal verification and scheduling. So in this integration scheme, the core polychronous model provides formal semantics support and its environment provides tool connection. Model-based system integration has also been discussed in [34] with regard to cyber-physical systems, [6] for tool integration platform, [31] based on SOA (Service of Architecture), [10] for heterogeneous models integration, and [29] for real-time software engineering. AUTOSAR[2] aims at component-level integration for automotive systems. System Architecture Virtual Integration (SAVI) program [30] [13] aims at creating an architecture-centric model repository to support analysis of virtually integrated system models related to performance, safety, and reliability, and so on. It also enables to discover system-level faults at the early design phase, thus reduce risk, cost, and development time.

3 A Model-Based Architecture-Centric Virtual Integration Framework

Based on the previous exploration of design issues and the state of the art of solutions in research, we find an architecture-centric model-based integration framework is necessary for the next-generation design of automotive software systems. The framework is expected to provide the following advantages: reliable model integration, fast and early-phase design validation, architecture optimization enabling, easy access to current matured software development tools and environment, etc. With this objective in mind, we first propose a model-based

architecture-centric virtual integration approach, in the framework of model-based systems engineering [11], for the design of next-generation automotive systems. This approach is involved in mostly *correct by construction* technologies, rather than a posteriori *Verification & Validation* in the implementation phase. We adopt different modeling languages with regards to different views of the system, for example, AADL for architecture modeling and Simulink for behavioral modeling, etc. The main research topics in the project include: timing specification [5], design by contracts and semantics interoperability for the purpose of a reliable model integration, which are explained in the following subsections.

Timing specification

With all the concerns in the embedded system design, timing is one of the most significant ones. In general, the timing issue becomes more explicit when architecture is considered and the system is integrated, due to the gap between software and architecture design. In our project, we consider high-level, formalized timing constraints to be defined, observed and analyzed based on software architecture, specified in AADL. From this point of view, an architecture centric approach is adopted for the model integration in our project. Considering abstraction in the system design, we advocate the modeling of synchrony and time as software and hardware events, which are related to synchronization in an architecture specification. Compared to real time, synchronous logical time, applied on both software and architecture, provides an algebraic framework in which both event-driven and time-triggered execution policies can be specified.

In the framework of our project, we define the semantics and algebra with regard to logical timing constraints and specification, and support the submission of a timing-related annex to the SAE standard AADL[32]. This annex will define a synchronous and timed specification framework to formally model time domains pertaining to the design of embedded architectures, including the specifications of automotive software architectures. The behavior annex of AADL are considered as the vehicle to implement this model, together with a timing annex (TA), as a mean to represent abstractions of these behavior annexes using clock constraints and regular expressions.

Design by contract

Design by contract [22] [15] is also adopted in our approach in the project. Contracts play a significant role in the safe and reliable model integration in our approach. We first analyze high-level requirements from automotive or avionic systems, from which *formalizable* requirements are then extracted according to the technical formalizability and verifiability. These requirements are expressed in formal languages so that they can be used to build the contracts for the integration of models that implement corresponding functionality. The contracts are expected to consider different criteria for safety, performance, cost, timing constraints, and so on. A mathematical framework will then be built to define the

composition of these models, together with the contracts on them, in a formal way. The contracts and their associated models will be checked by modeling checking technologies [14] [19] [7] .

Semantics interoperability

One of the main issues in the composition of models is semantics difference between heterogeneous models and different formalism. One of the feasible solutions to this issue is to have a common model as the intermediate formal model, and all other models are translated into the common model. An example can be found in [37]. The intermediate model provides the formal semantics, based on which, expected properties of the original models and their integration are checked. However, this requires a semantics preservation in the model translation, which is not practical in most cases. Another solution is related to formal semantics interoperability. Some work can be found in [3] [20], [18]. Our current research topic is focusing on the study of differences between the models, which can lead to issues in the model translations, from the point of view of model semantics, particularly timing semantics and operational semantics. The expected result of this research is intended to provide a foundation of the previous two research topics.

4 Conclusion

In this position paper, we have presented several important issues in current system design related to embedded systems, such as multi-paradigm modeling, integration framework, and formal semantics issues. A brief survey of corresponding research topics was also presented. We, hence, propose a model-based architecture-centric integration approach, considering timing specification, design by contract and semantics interoperability as main topics of research. Based on these research, a model-based integration framework is expected to be built, which is dedicated to model-based systems engineering for next-generation automotive systems.

Acknowledgment

The authors appreciate the valuable advices from Ryo Ito and Kazuhiro Kajio (Toyota Motor Corporation).

References

1. Systems Modeling Language (SysML). http://www.sysml.org/specs.
2. AUTOSAR (AUTomotive Open System ARchitecture). http://www.autosar.org/.
3. A. Benveniste, B. Caillaud, L.P. Carloni, P. Caspi, and A.L. Sangiovanni-Vincentelli. Composing Heterogeneous Reactive Systems. *ACM Transactions on Embedded Computing Systems*, 7(4), 2008.

4. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 2003.
5. L. Besnard, E. Borde, P. Dissaux, T. Gautier, P. Le Guernic, and J.-P. Talpin. Logically timed specifications in the aadl : a synchronous model of computation and communication (recommendations to the sae committee on aadl. Technical Report 446, INRIA, 2014.
6. M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments. *Proceedings of the IEEE*, 98:526–545, 2010.
7. Darren Cofer, Andrew Gacek, Steven Miller, Michael W Whalen, Brian LaValley, and Lui Sha. Compositional Verification of Architectural Models. In *NASA Formal Methods*, 2012.
8. DARPA. Adaptive Vehicle Make (AVM) Project. http://www.darpa.mil/Our_Work/TTO/Programs.
9. EAST-ADL. http://www.east-adl.info.
10. J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity - the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
11. J.A. Estefan. Survey of Model-Based Systems Engineering (MBSE) Methodologies. Technical report, INCOSE MBSE Initiative, 2008.
12. Esterel Technologies. SCADE Suite. http://www.esterel-technologies.com/products/scade-suite/.
13. P. Feiler, J. Hansson, D. de Niz, and L. Wrage. System Architecture Virtual Integration: An Industrial Case Study. Technical report, Software Engineering Institute, Nov. 2009. CMU/SEI-2009-TR-017.
14. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 441–444, Berlin, Heidelberg, 2006. Springer-Verlag.
15. J.-M. Jézéquel and B. Meyer. Design by Contract: The Lessons of Ariane. *Computer*, 30:129–130, 1997.
16. Xiaoqing Jin, Jyotirmoy Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Challenges of Applying Formal Methods to Automotive Control Systems. In *NSF National Workshop on Transportation Cyber-Physical Systems*, 2014.
17. P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12:261–304, 2002.
18. E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 2006.
19. A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *Runtime Verification*, 2010.
20. D. Mathaikutty, H. Patel, S. Shukla, and A. Jantsch. Modelling Environment for Heterogeneous Systems based on MoCs. In *Forum on specification and Design Languages (FDL)*, pages 291–303, 2005.
21. MathWorks. The MathWorks: Matlab/Simulink. http://www.mathworks.com/products/simulink/.
22. B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, Oct 1992.
23. Modelica and the Modelica Association. https://www.modelica.org.

24. P. J. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *SIMULATION: Transactions of the Society for Modeling and Simulation International*, 80(9):433–450, 2004.
25. K.D. Müller-Glaser, G. Frick, E. Sax, and M. Kühl. Multiparadigm Modeling in Embedded Systems Design. *IEEE Transactions on Control Systems Technology*, 12(2):279–292, 2004.
26. Object Management Group (OMG). The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. http://www.omg.org/spec/MARTE/1.1/PDF, June 2011.
27. OMG. Unified modeling language (uml). www.uml.org/.
28. OSATE. OSATE V2 Project. https://wiki.sei.cmu.edu/aadl/index.php/Osate_2.
29. Maxime Perrotin, Eric Conquet, Julien Delange, André Schiele, and Thanassis Tsiodras. TASTE: A Real-Time Software Engineering Tool-Chain Overview, Status, and Future. In *SDL 2011: Integrating System and Software Modeling*, 2012. Lecture Notes in Computer Science Volume 7083, pp 26-37.
30. D. Redman, D. Ward, J. Chilenski, and G. Pollari. Virtual integration for improved system design,. In *The First Analytic Virtual Integration of Cyber-Physical Systems Workshop in conjunction with RTSS*, 2010.
31. A. Rossignol. The Reference Technology Platform. In *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Springer, 2013.
32. SAE Aerospace (Society of Automotive Engineers). Aerospace Standard AS5506A: Architecture Analysis and Design Language (AADL) . *SAE AS5506A*, 2009.
33. D.C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39:25–31, 2006.
34. J. Sztipanovits, X. D. Koutsoukos, G. Karsai, N. Kottenstette, P.J. Antsaklis, V. Gupta, B. Goodwine, J.S. Baras, and S. Wang. Toward a Science of Cyber-Physical System Integration. *Proceedings of the IEEE*, 100(1):29–44, 2012.
35. A. Toom, T. Naks, M. Pantel, M. Gandriau, and I. Wati. Gene-Auto: An Automatic Code Generator for a Safe Subset of SimuLink/StateFlow and Scicos. In *European Conference on Embedded Real-Time Software (ERTS'08)*, 2008.
36. H. Yu, Y. Ma, T. Gautier, L. Besnard, J.-P. Talpin, and P. Le Guernic. Polychronous Modeling, Analysis, Verification and Simulation for Timed Software Architectures. *Journal of Systems Architecture (JSA)*, 59(10):1157–1170, 2013.
37. H. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P. Le Guernic, A. Toom, and O. Laurent. System-level Co-simulation of Integrated Avionics Using Polychrony. In *ACM Symposium on Applied Computing (SAC'11)*, 2011.