Front.Comput.Sci.
DOI

RESEARCH ARTICLE

Exploring Software Architectures in AADL via POLYCHRONY and SYNDEX

Huafeng YU, Yue MA, Thierry GAUTIER (✉)[1], Loïc BESNARD (✉)[2]
Jean-Pierre TALPIN, Paul LE GUERNIC(✉)[1], Yves SOREL(✉)[3]

1 INRIA Rennes, 263, av. du Général Leclerc, 35042 Rennes, France
2 IRISA/CNRS, 263, av. du Général Leclerc, 35042 Rennes, France
3 INRIA Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France

Abstract Architecture Analysis & Design Language (AADL) has been increasingly adopted in the design of embedded systems, and corresponding scheduling and formal verification have been well studied. However, little work takes code distribution and co-simulation into account, particularly considering clock constraints, for distributed multiprocessor systems. In this paper, we present our approach to handle the previous concerns, together with the associated tool chain, AADL-POLYCHRONY-SYNDEX. First, in order to avoid semantics ambiguities of AADL, the polychronous/multiclock semantics of AADL, based on a polychronous model of computation, is considered. Clock synthesis is then carried out in POLYCHRONY, which bridges the gap between the polychronous semantics and the synchronous semantics of SYNDEX. The same timing semantics is always preserved in order to ensure the correctness of the transformations between different formalism. Code distribution and corresponding scheduling is carried out on the obtained SYNDEX model in the last step, which enables the exploration of architectures originally specified in AADL. Our contribution provides a fast yet efficient architecture exploration approach for the design of distributed real-time and embedded systems. An avionic case study is used here to illustrate our approach.

Keywords POLYCHRONY, AADL, SYNDEX, scheduling, distribution, architecture exploration

Received month dd, yyyy; accepted month dd, yyyy

E-mail: {*firstname*}.{*lastname*}@*inria.fr* and *loic.besnard@irisa.fr*

1 Introduction

Architecture Analysis & Design Language (AADL) [1] is gradually adopted for high-level system co-modeling in embedded systems due to issues of system complexity, time to market, validation, etc. It permits the fast yet expressive modeling of a system. Early-phase analysis and validation can be therefore rapidly performed [2–8]. AADL provides a fast design entry, however, there are still some critical challenges, such as unambiguous semantics, architecture exploration, code distribution, timing analysis, and co-simulation. To address these issues, expressive formal models and complete tool chains are required.

In this paper, our approach to address previous issues is presented, including the formal timing modeling, timing analysis, clock synthesis, architecture exploration and the associated toolchain AADL-POLYCHRONY-SYNDEX. First, two polychronous/synchronous languages [9], i.e., POLYCHRONY/SIGNAL [10] and SYNDEX [11], are adopted to provide support for formal timing modeling based on a polychronous Model of Computation (MoC). We review the multiclock timing semantics of AADL, derived from AADL software and execution platform components. These components generally have the multiclock nature, in consequence they are modeled with our polychronous MoC. In this way, users are not suffered to find and/or build the fastest clock in the system, which distinguishes from [12], [3], [11], [13]. According to this principle, AADL models are transformed into SIGNAL

models. To bridge the gap between the polychronous semantics of SIGNAL and synchronous semantics of SYNDEX, clock synthesis in POLYCHRONY [14], the design environment dedicated to SIGNAL, is applied. The translation from SIGNAL to SYNDEX is integrated in POLYCHRONY. Finally, SYNDEX models are used to perform distribution, scheduling, and architecture exploration.

The main advantages of our approach is: 1) a formal model is adopted to connect the three languages, and it helps to preserve the semantics coherence and correct code generation in the transformations; 2) the formal model and methods used in the transformation are transparent to AADL practitioners, and it is fast and efficient to have the illustrative results for architecture exploration; 3) it provides the possibility for one of the three languages to take advantage of the functionalities provided by the other two languages.

A tool chain has been developed, which includes model transformations between the three languages, considering both semantics and syntactic aspects. A tutorial avionic case study is used in this paper to show the effectiveness of our contribution. This compact yet typical and general case study has been developed in the framework of the CESAR project [15].

Outline. Section 2 briefly introduces the three languages: AADL, POLYCHRONY/SIGNAL, SYNDEX. Section 3 presents our main contribution: the timing modeling, translations between different formalism, clock synthesis, architecture exploration, and exemplifying the whole work with a case study. Some related works are summarized in Section 4, and conclusion is drawn in Section 5.

2 Background

SIGNAL and POLYCHRONY

Synchronous languages are dedicated to the trusted design of synchronous reactive embedded systems [9]. Among these languages, the SIGNAL language stands out as it enables to describe systems with multiclock relations [10], and to support *refinement* [16].

SIGNAL handles unbounded series of values in the domain D_x, where $x = (x_t)_{t\in\mathbb{N}}$, called *signals*, implicitly indexed by discrete time. At any *instant*, a *signal* is either present and holds a value v in D_x; or absent and holds an extra value. The set of instants when a *signal* x is present represents its *clock*. In other words, SIGNAL is based on a synchronized dataflow model, called polychronous MoC. Two *signals* are said to be synchronous if they are both present (or both absent) at the same instants (they have the same clock). In comparison, a polychronous system is composed of a set of equations on signals, which are not necessary synchronous.

The abstract representation of a SIGNAL program is a Data Control Graph (DCG), composed of a clock hierarchy and a conditioned precedence graph. Clocks, which represent program controls, are structured in a hierarchy in the form of a forest (set of clock trees). In a hierarchy, child clocks are always included in the parent clock (decided by the inclusion relation of their instants).

SIGNAL is associated with the POLYCHRONY design environment [14], which provides a formal framework for back-end semantic-preserving transformation, scheduling, code generation, formal analysis and verification, architecture exploitation, and distribution [17]. However, SIGNAL is less expressive in the specification of execution platform and real-time characteristics. This is the main reason why AADL is connected as a front-end to express these information. The polychronous semantics of SIGNAL makes it more approximate to AADL timing semantics than other pure synchronous or asynchronous models.

AADL

As an architecture description language, based on a component modeling approach, AADL describes the structure of systems as an assembly of software components allocated on execution platform components together with timing semantics. The purpose of a model in AADL is to describe the execution characteristics of the system.

In the following, an industrial case study of simplified doors and slides control system (SDSCS) in the avionic generic pilot application, proposed by Airbus in the frame of CESAR project, is used to illustrate the basic components of an AADL model. In this case study (in Figure 1), a typical safety critical system takes charge of the management of passenger doors. It includes different components modeling the hardware and software, and allowing them to communicate and to control the doors.

In the system *door_manager*, two subsystems *door1* and *door2*, are managed by two processes *doors_process1* and *doors_process2* (in Figure 1). The processor *CPIOM1* (resp. *CPIOM2*) is responsible for scheduling and executing threads in process *doors_process1* (resp. *doors_process2*). The devices, e.g., *LGS*, *DPS*, etc., interface with external environment of the system. All the communication between the devices and processors is through the bus *AFDX1*.

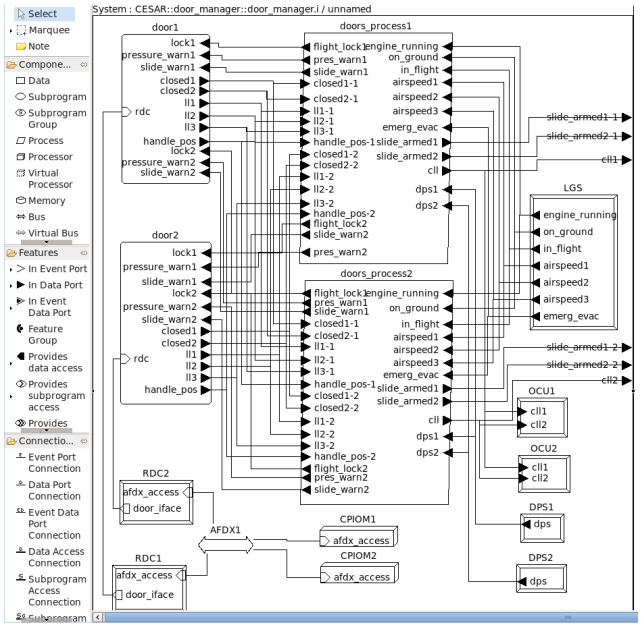

Fig. 1 An overview of the SDSCS case study in AADL

SynDEx

SynDEx [11] is a system level Computer-Aided Design (CAD) software, based on the *Algorithm Architecture Adequation* (AAA) methodology, for rapid prototyping and optimizing the implementation of real-time embedded applications on multicomponent architectures [18].

Algorithm Model. An algorithm graph is a conditioned factorized dataflow graph repeated infinitely according to every reaction of the application, according to the synchronous model. A dataflow graph is a directed acyclic graph (DAG) where vertices are application functions and edges are dependences between functions, defining a partial order on the operation execution. A conditioned vertex is a hierarchical vertex which contains several sub-dataflow graphs. According to the value of its specific input, called *condition*, only one of the possible sub-dataflow graphs will be executed during the considered reaction. A factorized vertex is also a hierarchical vertex which contains a repetition of N times of one sub-dataflow graph. These specific vertices are the equivalent in the dataflow model of `If...Then...Else...` and `For i= 1 to N Then...` in the control flow model.

Architecture model. A multiprocessor target architecture is modeled as a non-oriented hypergraph of operators, e.g., processors (graph vertices), connected through bidirectional communication media, e.g., bus (non-oriented graph edges). Each operator is a finite state machine (programmable, with instruction and data memories), which executes sequentially functions of the algorithm. Each communication medium executes sequentially communication functions.

Implementation model. Given a pair of algorithm and architecture graphs, the algorithm graph is transformed according to the architecture graph in order to obtain an implementation graph [19]. This transformation corresponds to the allocation and a scheduling of the algorithm graph. The allocation as well as the schedule is optimized through a multiprocessor real-time schedulability analysis and a resource analysis.

Code generation. SynDEx also permits the generation of dedicated distributed real-time executives with optional real-time performance measurement. Dedicated executives, which induce minimal overhead, are built from processor-dependent executive kernels (provided for processors such as PIC18F2680, ADSP21060 and i80386, and communication media like TCP/IP, RS232, CAN).

3 From AADL to Signal and SynDEx

3.1 From AADL to Signal

An AADL model describes the architecture and execution characteristics of an application system in terms of its constituent software and execution platform components and their interactions. Such characteristics depend on the hardware executing the software, where the timing properties and execution binding properties are associated. In this section, we mainly handling timing and binding properties of AADL components with regard to our polychronous MoC. Syntactic aspects in the transformation are only briefly described. The timing modeling of AADL applications in the framework of Polychrony is based on these properties.

Modeling AADL components

Each AADL component and its interface, e.g., process, thread, device, etc., is modeled as a Signal process and its corresponding interface. Assembly of AADL components, considering connections and binding, is represented by a parallel composition of corresponding translated Signal processes [10]. So this translation is considered as a n-to-n syntactic translation.

Timing properties of components

AADL supports an input-compute-output model of communication and execution for threads and port-based communication (Figure 2). The inputs of a thread received from

other components are frozen at a specified point, represented by *Input_Time* property, by default the *dispatch* time, during thread execution and made available to the thread for access. From that point on, its content is not affected by the arrival of new values for the remainder of the current execution until an explicit request for input, e.g., the two new arrival values 2 and 3 (in Figure 2) will not be processed until the next *Input_Time*. Similarly, the output is made available to other components at time specified by *Output_Time* property, for data ports by default at *complete* or *deadline* time depending on the associated port connection communication type.

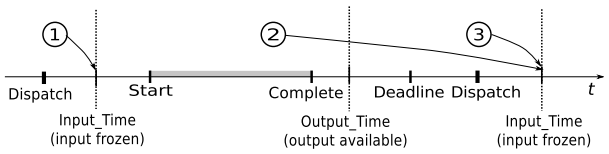

Fig. 2 A time model of the execution of a thread

The key idea for modeling the AADL computing latency and communication delay in SIGNAL is to keep the ideal view of instantaneous computations and communications moving computing latency and communication delays to specific *memory* processes, that introduces delay and well suited synchronizations [6]. As a consequence, various properties result in explicit synchronization signals.

The values of timing properties, e.g., *Input_Time*, *Output_Time*, *period*, *deadline*, etc., are preserved in the *pragmas* of generated SIGNAL programs with the list of the designations of objects with which they are associated.

Processor and its affine-scheduling

An AADL model is not complete and executable if the processor-level scheduling is not resolved. A scheduler is therefore expected to be integrated so that a complete model is used for the following validation, distribution and simulation. A scheduling based on affine clock systems [20] is thus developed for each AADL processor. A particular case of affine relations is affine sampling relation, expressed as $y = \{d \cdot t + \phi \mid t \in x\}$, of a reference discrete time x (d, t, ϕ are integers): y is a subsampling of positive phase ϕ and strictly positive period d on x. Affine clock relations yield an expressive calculus for the specification and the analysis of time-triggered systems. The scheduling based on the affine clocks can be easily and seamlessly connected to POLYCHRONY for formal analysis.

Binding

For a complete system specification, the application component instances must be executed by the appropriate execution platform components. The decisions and methods, related to combine the components of a system to produce a physical system implementation, are collectively called *binding* in AADL.

A process is bound to the processor specified by the *Actual_Processor_Binding* property. Support for process/threads execution may be embedded in the processor hardware, or it may require software that implements processor functionality. Such software must be bound to a memory component that is accessible to the processor via the *Actual_Memory_ Binding* property. The interactions among these execution platform components are enabled through a bus via the *Actual_Connection_Binding* property.

Binding properties are declared in the system implementation that contains in its containment hierarchy both the components to be bound and the execution platform components that are the target of the binding. These binding information are kept in the generated SIGNAL program by specific *pragmas*.

3.2 Timing analysis and clock synthesis

The previous translation from AADL to SIGNAL concentrates on the timing modeling of software architectures in AADL. This modeling enables a formal timing analysis and clock synthesis, based on the polychronous MoC.

Timing analysis

Timing analysis mainly refers to analysis of clock relations based on clock hierarchy. DCG represents a program with all dependences set. The following DCG levels are distinguished: 1) The **DCGBasic** level is the most general level. 2) The **DCGPoly** level is the subtype of DCGBasic such that the clock hierarchy in the DCG is the result of the clock calculus. Specific clocks such as *tick* are created, but the clock hierarchy, in the general case, has several roots. 3) The **DCGEndo** level is the subtype of DCGPoly such that the clock hierarchy in the DCG is a tree (it is provided with a single root which is *tick*). The program is endochronous. 4) The **DCGBool** level is the subtype of DCGEndo such that all clock expressions are Boolean extractions. Clocks are represented as Boolean signals. 5) The **DCGFlat** level is the subtype of DCGBool such that the clock hierarchy in the DCG is flat: only two levels are kept and each Boolean clock signal is defined on

the root clock *tick*.

Clock synthesis

In a clock tree with only one root, the simulation clock (the fastest clock in the system) is based on the root clock (i.e., the *tick*). In this case, the system is endochronous and it is possible to build the unique deterministic behavior in the code generation. But if there is no common root for all the trees, i.e., there is no fastest clock, the system is polychronous, and non-deterministic concurrency is thus introduced. In an AADL multi-processor specification, it is generally hard to find the fastest clock, as each component may have its own activation clock or frequency. Code generation considering the deterministic behavior is therefore difficult, even impossible. To tackle this issue, independent clocks, particularly the root clocks in different trees, are required to be synchronized. This synchronization, called *endochronization*, can be performed in an ad-hoc way by the compiler, or in a specific way in a manual manner. More details can be found in [17]. A more general clock synchronization method via controller synthesis is also possible [21]. Endochronization leads to the passage from the **DCGPoly** level to the **DCGEndo** level.

3.3 From SIGNAL to SYNDEX

Both SYNDEX and SIGNAL belong to the family of synchronous/polychronous languages. However, there are still several differences to consider in the translation. First, SIGNAL is based on a polychronous model while SYNDEX is based on the synchronous model. Clock synthesis is therefore needed to handle multiclocked specifications before the translation. Secondly, system representations of SIGNAL and SYNDEX, based on graphs, can be at different abstraction levels. Transformation between these levels is thus required so that the translation is performed at the same level. Finally, execution platform and real-time characteristics, preserved in SIGNAL pragmas, are translated into SYNDEX. In this section, we briefly present our contribution to resolve the previous issues.

Appropriate translation level

Different levels are appropriate for different compiling targets. For example, code generation benefits from the clock hierarchy based on the DCGBool representation; DCGFlat representation is more close to the input format of some provers, since the clock signals are *always* present. The reference level for the translation from SIGNAL to SYNDEX is expected to be the DCGBool level, which enables the optimized code generation. However, this would require SYNDEX to consider the clock hierarchy defined in the SIGNAL code generation. To avoid this problem, we have chosen the DCGFlat instead of DCGBool level. The technical difficulty of code generation from a non-flat hierarchy disappears at the DCGFlat level where all clocks are set according to the master clock.

Translation

Once and the translation level is chosen, it remains to define the structure of the SYNDEX code translated from a SIGNAL program. The correspondence between the two representations is defined in two aspects: 1) The first one is the clock hierarchy, which plays the semantic role and serves as a structural *backbone*: each clock in the hierarchy is associated with a SYNDEX algorithm. This algorithm combines the calculations to be performed according to this clock, which is represented by a SYNDEX *condition*. 2) The second one is related to syntax: it consists of all the SIGNAL processes that are preserved as structuring units (SYNDEX algorithms, in this case) in the generated SYNDEX code. By default, SIGNAL processes are expanded during compiling: those that are kept should be explicitly marked (via associated *pragmas*). In the current state, they should be endochronous. Thus, using these presented units, it is possible to find, in SYNDEX, the necessary elements of syntactic structure, which allows one to associate specific information: allocation constraints, temporal information, etc.

By following these general principles, translating from SIGNAL to SYNDEX is performed by applying a set of transformations on abstract graph models (unlike the model transformation from AADL to SIGNAL in the framework of EMF [22] in the previous section).

In the transformation of ASME2SSME, **execution platform**, **timing properties**, and **binding** specified in AADL are kept in the *pragma* part of SIGNAL programs. In the translation from SIGNAL to SYNDEX, the information of execution platform, related to processing and communication units, are taken from the *pragma* and are translated into an architecture graph in SYNDEX. Timing properties, such as *period*, *deadline*, *computation time*, are taken from the *pragmas* and finally set to corresponding SYNDEX nodes. The binding information of software and hardware are translated into *groups* and allocation *constraints* in SYNDEX, i.e., all the software components allocated on the same processing unit are put into the same group, and then a constraint is specified in SYN-

DEx so that this group is allocated on the same processing unit. As the translation is simple and direct, no more detail is given here.

3.4 A toolchain AADL-Polychrony-SynDEx

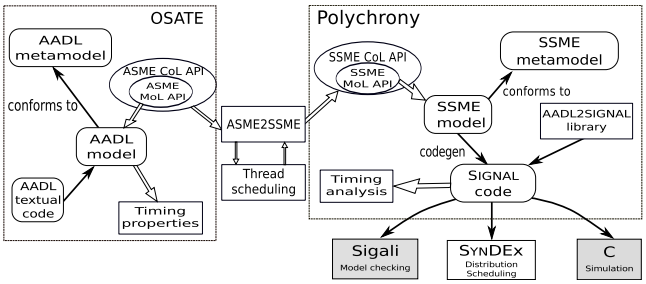

Fig. 3 The AADL-Polychrony-SynDEx tool chain

A toolchain (Figure 3) for modeling, scheduling, timing analysis, and verification of AADL models in the polychronous MoC has been developed in Eclipse Modeling Framework (EMF). The AADL model with timing properties, which conforms to the AADL metamodel, is captured in the OSATE toolkit [23]. A model transformation **ASME2SSME** allows one to perform analysis on ASME models (AADL Syntax Model under Eclipse) and generate corresponding Signal SSME models (Signal Syntax Model under Eclipse). The Signal code is then generated.

3.5 Architecture exploration with SynDEx

SynDEx enables the allocation of functions onto the resources considering timing requirements and minimization of resources usage. It allows users to explore manually and/or automatically the design space solutions using optimization heuristics. Exploration is mainly carried out through timing analyses and simulations. The results of these predict the real-time behaviour of the application functions executed on the various resources, i.e., processors, integrated circuits and communication media. This approach conforms to the typical hardware/software codesign and architecture exploration process [24, 25]. Finally, for the software part of the application, code is automatically generated as a dedicated real-time executive, or as a configuration file for a resident real-time operating system such as Osek [26] or RTlinux. This approach will improve the design safety provided by formal models, and decrease the development cycle thanks to timing simulation and automatic code generation.

3.6 A case study

SDSCS is a generic simplified version of the system that manages passenger doors on Airbus series aircraft. As a safety-critical system, in addition to the fulfillment of safety objectives, high-level modeling and component-based development are also expected for fast and efficient design.

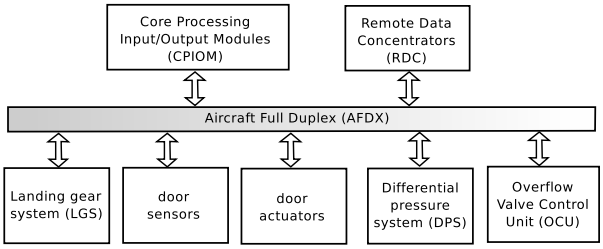

Fig. 4 A simple example of the SDSCS architecture.

In this example, each passenger door has a software handler, which achieves the following tasks: 1) monitor door status via door sensors; 2) control flight lock actuators; 3) manage the residual pressure of the cabin; 4) inhibit cabin pressurization if any external door is not closed, latched and locked. The four tasks are implemented with simple logic that determines the status of monitors, actuators, etc., according to the sensor readings. In addition to sensors and actuators, SDSCS is equipped with other hardware components, e.g., CPIOMs (Core Processing Input/Output Modules) and RDCs (Remote Data Concentrators) are connected via the AFDX (Aircraft Full DupleX) network (Figure 4). Sensors and actuators are also connected to RDCs and communicate with other systems via AFDX.

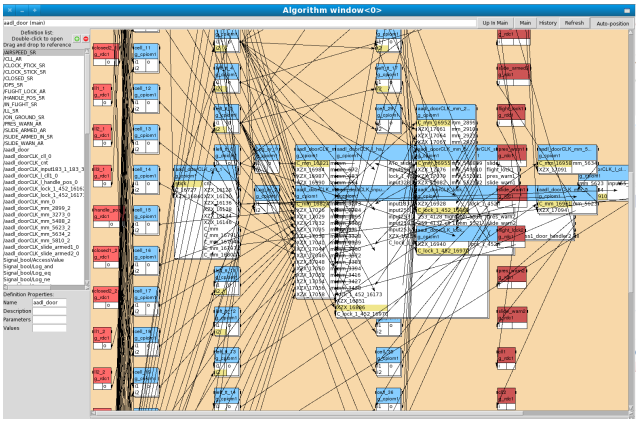

Fig. 5 A graph of SynDEx algorithm: obtained from SDSCS case study

An overview of SDSCS modeled in AADL is shown in Figure 1. The whole system is presented as an AADL system. The two doors, modeled as subsystems, are controlled by two

processes *doors_process1* and *doors_process2*. Each process contains three threads to perform the management of doors. All the threads and devices are periodic and share he same period. The process *doors_process1* (resp. *doors_process2*) is executed on a processor *CPIOM1* (resp. *CPIOM2*). The bus *AFDX* connects the processors, *CPIOM1* and *CPIOM2*, and devices that model the sensors and actuators, such as *DPS*, *OCU*, etc.

The case study was successfully transformed with the tool chain described in Figure 3. From an .aadl file (AADL textual file), we first get .aaxl (AADL model file) with OSATE. Aided by high-level and low-level APIs, ASME2SSME is able to read the AADL models. While performing the model transformation towards the SSME model, a thread-level scheduler is also integrated. The SSME model is transformed into SIGNAL code, from which static analysis can be performed. The SIGNAL code can be transformed into .sdx file (SYNDEX code), as well as other format for model checking and simulation. From the SYNDEX tool, the .sdx file is red, and the adequation can be carried out. Figure 5 illustrates the SYNDEX algorithm graph describing a partial order on the execution of the functions. This algorithm is translated from the AADL functional part of SDSCS. Figure 6 shows the SYNDEX architecture graph, which is translated from the AADL architectural part of SDSCS. The two processing units (CPIOM1 and CPIOM2), two concentrators (RDC1 and RDC2), and the communication media (AFDX1) can be easily found in the figure.

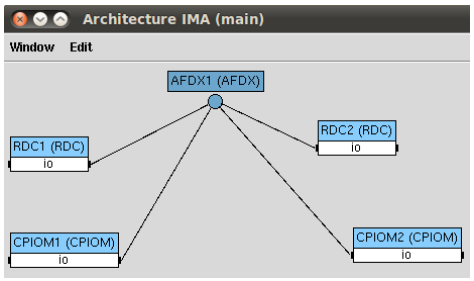

Fig. 6 A graph of SYNDEX architecture: obtained from SDSCS case study

Figure 7 illustrates partial adequation results in SYNDEX: algorithm (Figure 5) is mapped onto the architecture (Figure 6). There are five columns in the figure, which represent the five architectural components. The horizontal lines in the columns represent the computation or communication allocated on the corresponding architectural components. In this case study, the algorithm has more than 150 nodes and the architecture has 5 nodes in SYNDEX. The adequation takes about 15 minutes 35 seconds in average. With this tool chain, it is easy to change the configuration of the execution platform and binding. For example, the number of the processing units can be changed, the type of processing units and communication media can be easily changed. The influence of these changes is finally illustrative in SYNDEX. In addition, generation of dedicated distributed real-time executives with optional real-time performance measurement is also possible for different processors and communication media [11]. The names of AADL components are always kept in all the transformations in order to enable traceability. Hence, our approach provides a fast yet efficient architecture exploration for the design of distributed real-time and embedded systems.

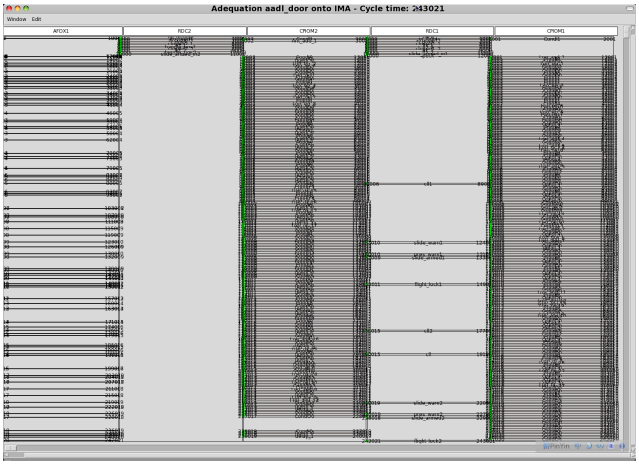

Fig. 7 A graph of SYNDEX adequation result: algorithm is mapped onto the architecture

4 Related work

Many contributions on schedulability analysis and scheduling tools exist for AADL specifications, such as [2], [27], and [28]. We have chosen SYNDEX. The main reason is that SYNDEX and SIGNAL have similar synchronous semantics; this proximity reduces the difficulties in the translation. In addition, the scheduler generated by SYNDEX can be integrated, in a manual manner currently, into the original SIGNAL programs for formal verification purpose.

AADL has been connected to many formal models for analysis and validation. The AADL2Sync project [12] and the Compass Approach [3] provide complete tool chains from modeling to validation, but they are generally based on the synchronous semantics, which is not approximate to AADL timing semantics. AADL2BIP [5] allows simulation of AADL models, as well as application of particular verifica-

tion techniques, i.e., state exploration and component-based deadlock detection. The AADL2Fiacre project [29] mainly concentrates on model transformation and code generation, in other words, formal analysis and verification are performed externally with other tools and models. The previous projects do not address code distribution and real-time scheduling in a general sense. The Ocarina project [7] considers code generation for distributed real-time and embedded systems, but using formal models and semantics is not reported in the work.

5 Conclusion

In this paper, we present our proposed approach to address architecture exploration based on AADL, Polychrony, and SynDEx: Software architecture is specified in AADL at a high level of abstraction; Polychrony provides the polychronous model of computation, formal analysis, clock synthesis, and transformations to bridge between AADL and SynDEx; SynDEx is finally used for the code distribution and real-time scheduling. With a reduced design cost of embedded systems, this tool chain makes it possible to perform architecture exploration at the earliest design stage. An avionic case study is used to illustrate our approach.

In a longer term, as a perspective, the translation between Signal and SynDEx is expected to be bidirectional, i.e., a particular distribution and scheduling determined by SynDEx is automatically synthesized in the Signal programs for the purpose of formal verification, performance evaluation, and other analyses.

References

1. SAE Aerospace (Society of Automotive Engineers). Aerospace Standard AS5506A: Architecture Analysis and Design Language (AADL) . *SAE AS5506A*, 2009.
2. F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with AADL. In *ACM SIGAda international conference on ADA (SigAda'05)*, 2005.
3. M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, and M. Roveri. Safety, Dependability, and Performance Analysis of Extended AADL Models. *The Computer Journal*, 54(5):754–775, 2011.
4. P.H. Feiler and J. Hansson. Flow Latency Analysis with the Architecture Analysis and Design Language (AADL). Technical report, CMU, 2007.
5. M.Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Models in Software Engineering. chapter Translating AADL into BIP - Application to the Verification of Real-Time Systems, pages 5–19. Springer-Verlag, 2009.
6. Y. Ma, H. Yu, T. Gautier, J.-P. Talpin, L. Besnard, and P. Le Guernic. System Synthesis from AADL using Polychrony. In *Electronic System Level Synthesis Conference*, 2011.
7. J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems (TECS)*, 2008.
8. Z. Yang, K. Hu, D. Ma, and L. Pi. Towards a formal semantics for the AADL behavior annex. In *Design, Automation and Test in Europe (DATE)*, pages 1166–1171, 2009.
9. 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.
10. 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.
11. Y. Sorel. SynDEx: System-Level CAD Software for Optimizing Distributed Real-Time Embedded Systems. *ERCIM News*, 59:68–69, 2004.
12. E. Jahier, N. Halbwachs, and P. Raymond. Synchronous Modeling and Validation of Priority Inheritance Schedulers. In *Fundamental Approaches to Software Engineering (FASE'09)*, 2009.
13. A. Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ENTCS, Edinburgh, UK, April 2005. Elsevier Science, New-York.
14. The Polychrony Toolset. `http://www.irisa.fr/espresso/Polychrony/`.
15. Cost-efficient methods and processes for safety relevant embedded systems (CESAR project). `http://www.cesarproject.eu/`.
16. J.-P. Talpin, P. Le Guernic, S.K. Shukla, F. Doucet, and R. Gupta. Formal Refinement Checking in a System-level Design Methodology. *Fundamenta Informaticae*, 62(2):243–273, 2004.
17. L. Besnard, T. Gautier, P. Le Guernic, and J.-P. Talpin. Compilation of polychronous data flow equations. In Sandeep Shukla and Jean-Pierre Talpin, editors, *Correct-by-Construction Embedded Software Synthesis: Formal Frameworks, Methodologies, and Tools*, 2010.
18. O. Kermia and Y. Sorel. A Rapid Heuristic for Scheduling Non-Preemptive Dependent Periodic Tasks onto Multiprocessor. In *International Conference on Parallel and Distributed Computing Systems (PDCS)*, 2007.
19. T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *MEMOCODE'03*, 2003.
20. I.M. Smarandache, T. Gautier, and P. Le Guernic. Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints. In *World Congress on Formal Methods*, 1999.
21. H. Yu, J.P. Talpin, L. Besnard, T. Gautier, H. Marchand, and P. Le Guernic. Polychronous Controller Synthesis from MARTE CCSL Timing Specifications. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'11)*, pages

21–30, 2011.

22. Eclipse modeling framework project (emf). `http://www.eclipse.org/modeling/emf/`.
23. OSATE V2 Project. `http://gforge.enseeiht.fr/projects/osate2/`.
24. A.D. Pimentel, C. Erbas, and S. Polstra. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Trans. Computers*, 55(2):99–112, 2006.
25. M. Gries. Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.*, 38(2):131–183, December 2004.
26. Osek. `http://www.osek-vdx.org/`.
27. O. Sokolsky, I. Lee, and D. Clark. Schedulability Analysis of AADL models. In *20th International Parallel and Distributed Processing Symposium (IPDPS'06)*, 2006.
28. S. Gui, L. Luo, Y. Li, and L. Wang. Formal Schedulability Analysis and Simulation for AADL. In *International Conference on Embedded Software and Systems (ICESS)*, 2008.
29. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS'08*, 2008.