Rigorous Component-Based System Design Using the BIP Framework

Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, Joseph Sifakis

Rigorous Component-based System Design Using the BIP Framework

Ananda Basu Saddek Bensalem Marius Bozga Jacques Combaz
Mohamad Jaber Thanh-Hung Nguyen Joseph Sifakis

February 8, 2011

Abstract

Rigorous system design requires the use of a single powerful component framework allowing the representation of the designed system at different levels of detail, from application software to its implementation. The use of a single framework allows to maintain the overall coherency and correctness by comparing different architectural solutions and their properties.

In this paper, we present the BIP (Behavior, Interaction, Priority) component framework which encompasses an expressive notion of composition for heterogeneous components by combining interactions and priorities. This allows description at different levels of abstraction from application software to mixed hardware/software systems. Then, we introduce a rigorous design flow that uses BIP as a unifying semantic model to derive from an application software, a model of the target architecture and a mapping, a correct implementation. Correctness of implementation is ensured by application of source-to-source transformations in BIP which preserve correctness of essential design properties. The design is fully automated and supported by a toolset including a compiler, the D-Finder verification tool and model transformers. We illustrate the use of BIP as a modeling formalism as well as crucial aspects of the design flow for ensuring correctness, through an autonomous robot case study.

1 Introduction

System design is the process leading to a mixed hardware/software system meeting given specifications. It involves the development of application software taking into account features of an execution platform. The latter is defined by its architecture involving a set of processors equipped with hardware-dependent software such as operating systems as well as primitives for coordination of the computation and interaction with the external environment.

System design radically differs from pure software design in that it must take into account not only functional but also extra-functional specifications regarding the use of resources of the execution platform such as time, memory and energy. Meeting extra-functional specifications is essential for the design of embedded systems. It requires evaluation of the impact of design choices on the overall behavior of the system. It also implies a deep understanding of the interaction between application software and the underlying execution platform. We currently lack approaches for modeling mixed

hardware/software systems. There are no rigorous techniques for deriving global models of a given system from models of its application software and its execution platform.

We call rigorous a design flow which allows guaranteeing essential system properties. Most of the existing rigorous design flows privilege a unique programming model together with an associated compilation chain adapted for a given execution model. For example, synchronous system design relies on synchronous programming models and usually targets hardware or sequential implementations on single processors [1]. Alternatively, real-time programming based on scheduling theory for periodic tasks, targets dedicated real-time multitasking platforms [2].

A rigorous design flow should be characterized by the following:

- It should be *model-based*, that is all the software and system descriptions used along the design flow should be based on a single semantic model. This is essential for maintaining the overall coherency of the flow by guaranteeing that a description at step n meets essential properties of a description at step $n+1$.
- It should be *component-based*, that is, it provides primitives for building composite components as the composition of simpler components. Existing theoretical frameworks for composition are based on a single operator e.g., product of automata, function call. Poor expressiveness of these frameworks may lead to complicated designs: achieving a given coordination between components often requires additional components to manage their interaction.
- It should rely on tractable theory for guaranteeing *correctness by construction* to avoid as much as possible monolithic a posteriori verification.

BIP (Behavior, Interaction, Priority) is a component framework intended to rigorous system design. It allows the construction of composite hierarchically structured components from atomic components characterized by their behavior and their interface. Components are composed by layered application of interactions and of priorities. Interactions express synchronization constraints between actions of the composed components while priorities are used to filter amongst possible interactions and to steer system evolution so as to meet performance requirements e.g. to express scheduling policies. Interactions are described in BIP as the combination of two types of protocols: rendez-vous to express strong symmetric synchronization and broadcast to express triggered asymmetric synchronization. The combination of interactions and priorities confers BIP expressiveness not matched by any other existing formalism [3]. It defines a clean and abstract concept of architecture separate from behavior. Architecture in BIP is a first class concept with well-defined semantics that can be analyzed and transformed. BIP relies on rigorous operational semantics that has been implemented by three Execution Engines for centralized, distributed and real-time execution. It is used as a unifying semantic model in a rigorous system design flow. Rigorousness is ensured by two kinds of tools: 1) D-Finder a verification tool for checking safety properties and deadlock-freedom in particular; 2) source-to-source transformers that allow progressive refinement of the application to get a correct implementation.

BIP drastically differs from existing component frameworks for software engineering. These often use multithreaded programming and point-to-point interaction mechanisms such as function call for coordination between components while in BIP the execution of atomic components is inherently concurrent and their coordination is expressed in terms of high level mechanisms such as protocols and scheduling policies. BIP can be considered as an ADL (Architecture Description Language) or

as a coordination language as it focuses on the organization of computation between components. As other existing ADL such as ACME [4] and Darwin [5], BIP uses the concept of connector to express coordination between components. Nonetheless, connectors in BIP are stateless. There is a clear distinction between architecture which involves connectors and priorities and behavior. Another significant difference is that BIP is intended to system modeling as it directly encompasses timing and resource management aspects. It differs from other system modeling formalisms which either seek generality at the detriment of rigorousness, such as SysML [6] and AADL [7] or have a limited scope as they are based on specific models of computation such as Ptolemy [8].

The paper is organized as follows. Section 2 presents the BIP component framework including the language and the associated toolset. Section 3 highlights the BIP rigorous system design flow and presents the main steps for deriving correct implementations from a given application software and a target platform. Section 4 presents experimental results from the DALA autonomous robot case study. Section 5 concludes and discusses futures work directions.

2 The BIP Component Framework

The BIP framework allows building complex systems by coordinating the behavior of a set of *atomic components*. Coordination in BIP uses *connectors*, to specify possible interaction patterns between components, and *priorities*, to select amongst possible interactions.

Atomic components are finite-state automata or Petri nets that are extended with arbitrary data and ports. Ports are action names, and may be associated with data. They are used for interaction with other components. States denote control locations at which the components await for interaction. A transition is an execution step, labeled by a port, from a control location to another. It has associated a guard and an action, that are respectively a boolean condition and a function defined on local data. In BIP complex data and their transformations are written in C/C++.

A transition can be executed if its guard evaluates to true and some *interaction* involving its port is enabled. The execution is an atomic sequence of two microsteps: (i) execution of the interaction involving the port, which is a synchronization between several components, with possible exchange of data, followed by (ii) execution of the action associated with the transition.

Example 1 *Figure 1 shows two atomic components,* Service-Controller *and* Activity *of the DALA robot controller presented in section 4.* Activity *wraps the long-time computation of some specific applicative function.* Service-Controller *provides execution control (i.e., triggering, canceling, error control, etc) over the associated* Activity *component. For sake of simplicity, the figure presents only the skeleton control behavior (i.e., ports and transitions) whereas the data and associated code is omitted. For example,* Activity *is initialized (*start *transition) and then it executes its associated functions (*exec, internal_exec *transitions). The execution may finish normally (*finish *transition), may fail (*fail *transition) or may be interrupted (*inter *transition).*

Composite components are defined by assembling constituent components (atomic or composite) using *connectors*. Connectors relate ports of interacting components. They represent sets of interactions, that is, non-empty sets of ports that have to be jointly executed. Within a connector, an interaction can take place in two situations: either all involved ports are ready to participate

(strong synchronization), or a port triggers the interaction without waiting for other ports (broadcast). The set of valid interactions within connectors are formally defined using algebraic expressions on ports using a binary *fusion* operator and a unary *typing* operator [9]. Typing associates with connector-ends (ports or connectors) synchronization types: *trigger* (active port, initiates broadcast) or *synchron* (passive port). Moreover, with every interaction of a connector are associated a guard and a data transfer function. An interaction may be executed only when its guard is true. Its execution consists in computing the data transfer function and notifying the components involved in the interaction.

Finally, *priorities* are used to choose between simultaneously enabled interactions within a BIP component. These are rules, each consisting of an ordered pair of interactions associated with a condition. When the condition holds and both interactions of the corresponding pair are enabled, only the one with higher-priority can be executed.

Example 2 *Figure 1 presents the* Service *composite component obtained by the composition of* Activity *and* Service-Controller *through six connectors. These enforce strong synchronizations of several actions and allow the* Service-Controller *to initiate and follow the computation performed within* Activity*. Priorities are used to privilege the execution of* fail *interaction, that is error handling, over* finish *and* exec *interactions, which correspond to normal behavior. The example also illustrates the principle of encapsulation used in BIP: the* Service *component is further composed with the* ServiceProxy *component by using the ports available on its interface, which are explicitely re-directed either to ports of subcomponents or to inner connectors. The* trigger-request *connector between* ServiceProxy *and* Service *illustrates a broadcast initiated by the* trigger *port, that is,* trigger *actions are either executed alone, or synchronized with* request *actions, whenever enabled.*

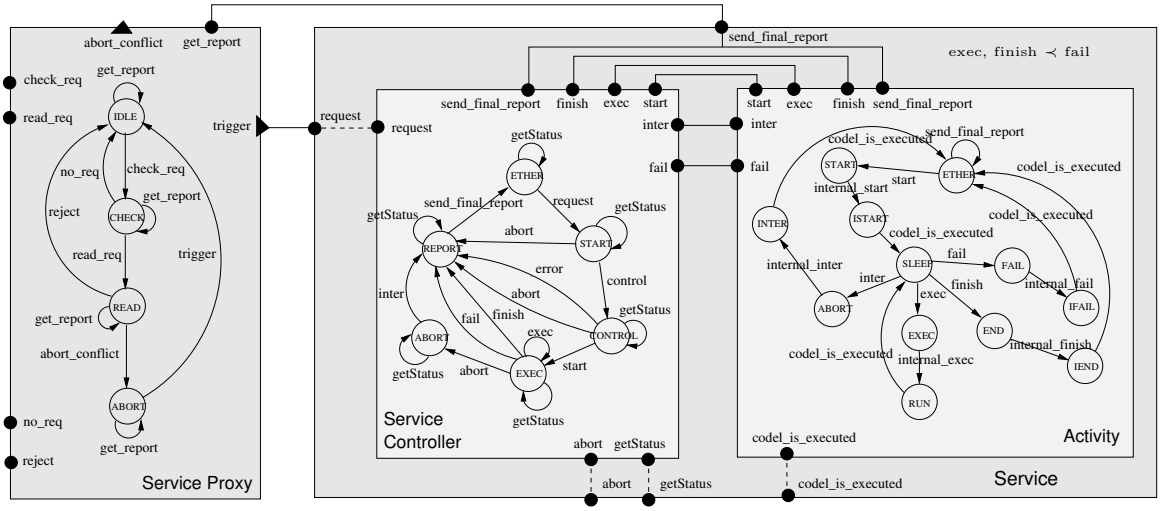

Figure 1: BIP Components: a service of the DALA Robot

The combined use of connectors and priorites confers BIP an unpaired expressive power for describing component coordination. This has been confirmed in practice by a number of succesful

experiments on direct modeling or componentizing existing software in BIP [10].

The BIP framework is supported by a concrete modeling language. The BIP language leverages on C++ style variables and data type declarations, expressions and statements, and provides additional structural syntactic constructs for defining component behavior, describing connectors and priorities. Moreover, it provides constructs for dealing with parametric and hierarchical descriptions as well as for expressing timing constraints associated with behavior.

3 The BIP Design Flow

We present a rigorous system design flow where BIP is used as a unifying semantic model to ensure consistency between the different design steps.

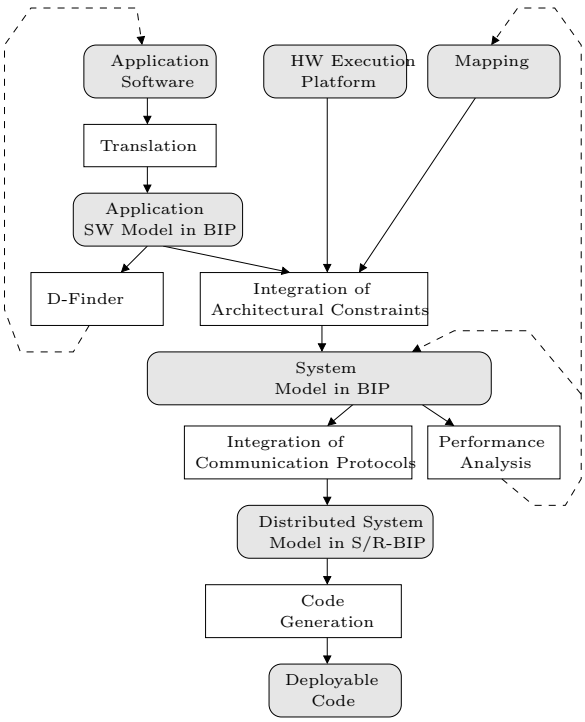

Figure 2: BIP Design Flow

The design flow involves four distinct steps as depicted in Figure 2. They consist in translating the application software into a BIP model and deriving progressively an implementation by application of source-to-source transformations. These transformations are correct-by-construction as the obtained BIP models are observationally equivalent. In particular, they preserve safety properties of the application software. Furthermore, the D-Finder verification tool is used to check essential safety properties of the application software. We describe below D-Finder and main transformations of the design flow.

The BIP design flow is entirely supported by the BIP language and an extensible toolset (figure 3). This includes translators from various programming models, verification tools, source-to-source transformers and C/C++ code generators for BIP models.

Translating Application Software into BIP

The first step in the design flow requires the generation of a BIP model for the application software. We have developed a general method for generating BIP models from languages with well-defined operational semantics. It involves the following three steps for a given application software written in a language $\mathcal{L}$:

1. Translation of atomic components of the source language into BIP components. The translation focuses on the definition of adequate interfaces. It encapsulates and reuses data structures and functions of the application software,
2. Translation of coordination mechanisms between components of the application software into connectors and priorities in the target BIP model,
3. Generation of a BIP component modeling the operational semantics of $\mathcal{L}$. This component plays the role of an engine coordinating the execution of the application software components.

We have developed BIP model generators for several programming models used by embedded system developpers (*Source2source transformers* in figure 3). The generated models preserve the structure and their size is linear with respect to the size of the initial programs. They are easy to understand by developers in source languages.

Compositional Verification by using D-Finder

A compositional verification method for BIP based on computation of invariants is presented in [11]. The method consists in computing increasingly stronger invariants for composite components as conjunctions of local invariants for atomic components and interaction invariants characterizing the composition glue. Local component invariants are generated by static analysis of atomic components. Interaction invariants are generated from abstractions of the composite component to be verified.

The method has been recently improved to take advantage of the incrementality of the design process. Incremental system design proceeds by adding new interactions to existing sets of components. Each time an interaction is added, it is possible to verify whether the resulting system violates a given property and discover design errors as soon as they appear. The incremental verification technique [12] uses sufficient conditions ensuring the preservation of invariants when new interactions are added along the component construction process. If these conditions are not satisfied, new invariants are generated by reusing invariants of the interacting components. Reusing invariants reduces considerably the verification effort.

Compositional verification techniques have been implemented in the D-Finder tool [13] for checking deadlock-freedom of systems described in BIP. Experimental results on classical benchmarks show that D-Finder can be exponentially faster than existing monolithic verification tools, like NuSMV.

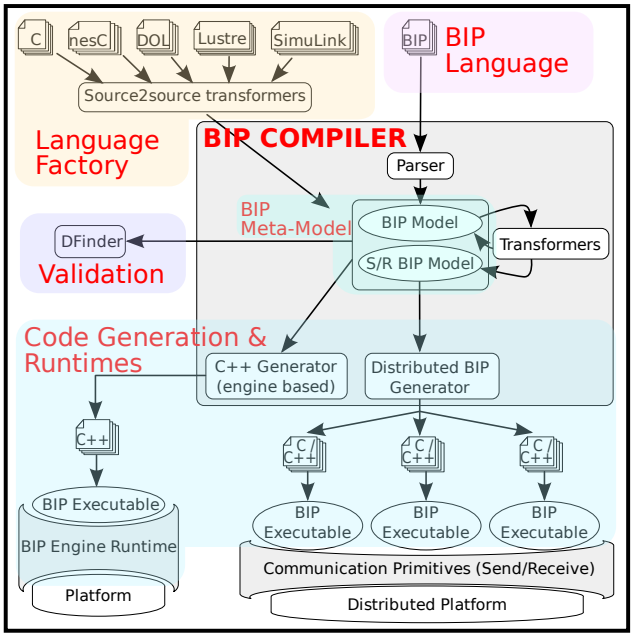

Figure 3: BIP Toolset

D-Finder has been also successful for the verification of complex software applications, as reported in section 4.

Generating Implementations

The BIP toolset offers several compilation chains, targetting different execution platforms. The implementation of BIP on single-core platforms is realized by using *Engines*, that is, dedicated middleware for execution of the C++ code auto-generated from BIP descriptions. BIP provides currently two Engines, respectively for real-time single-thread and multi-thread execution. For multi-thread execution, each atomic component is assigned to a thread, the Engine being a thread itself. Communication takes place only between atomic components and the Engine, and never directly between different atomic components.

To generate distributed implementations from BIP models it is necessary to transform them into S/R-BIP models [14]. These are a subclass of models where multi-party interaction is replaced by protocols using S/R (Send/Receive) primitives. From S/R-BIP models and a mapping of atomic components into processing elements of a platform it is possible to generate efficient C/C++ or MPI-code.

The method in [14] uses the following sequence of correct-by-construction transformations, that preserve *observational equivalence*:

1. Given a user-defined partition of its interactions, a BIP system model is transformed into a S/R-BIP system model such that (i) atomicity of transitions in the original model is broken by separating interaction and computation, and (ii) multi-party interactions of the source model are replaced by protocols using send/receive primitives. Moreover, the target S/R-BIP model is structured in three layers:

   (a) The *component layer* consists of the atomic components in the original model where each port involved in strong interactions is replaced by a pair of corresponding Send/Receive ports.

   (b) The *interaction protocol* layer consists of a set of components, each managing a class of interactions of the partition. This protocol detects enabledness of interactions and executes them after resolving conflicts either locally or assisted by the third layer.

   (c) The *conflict resolution protocol* layer resolves conflicts requested by the interaction protocol layer. This protocol resolves a *committee coordination problem* [15] using one distributed algorithm amongst (i) fully centralized, (ii) token-ring, and (iii) dining philosophers [16, 15].

2. We generate from the obtained 3-layer S/R-BIP model and a mapping of its atomic components on processors, either a MPI program, or a set of plain C/C++ programs that use TCP/IP communication. The generation consists in statically composing atomic components running on the same processor to obtain a single observationally equivalent component, and consequently reduced coordination overhead at runtime.

4 Case Study: DALA Robot Controller

We applied the rigorous design flow to develop in BIP a new version of the functional layer of the DALA robot controller from an existing version developed by using the GeNoM framework [17]. A preliminary result on this work presented in [10] summarizes our latest developements, including the complete modeling of the functional layer, its functional verification as well as the synthesis of a software controller which is correct-by-construction.

The design of the functional layer of the robot in BIP involved the following steps:

1. Hierarchical decomposition into components. The overall architecture can be represented as a tree. Its root is the functional layer and the leaves correspond to atomic components. The grammar below shows how the designed system can be obtained as the incremental composition of components:

$$
\begin{array}{rcl}
\textit{Functional Layer} & ::= & (\ \textit{Module}\ )^{+} \\
\textit{Module} & ::= & (\ \textit{Service}\ )^{+}\ .\ (\ \textit{Execution-Task}\ )^{+}\ .\ (\ \mathsf{Poster}\ )^{+} \\
\textit{Service} & ::= & (\ \mathsf{Service\text{-}Controller}\ )\ .\ (\ \mathsf{Activity}\ ) \\
\textit{Execution-Task} & ::= & (\ \mathsf{Timer}\ )\ .\ (\ \mathsf{Scheduler\text{-}Activity}\ )\ .\ (\ \mathsf{Execution\text{-}Controller})
\end{array}
$$

2. Description of the behavior of each atomic component. We provided in section 2 an abstraction for the Service-Controller and Activity components. In addition, the functional layer includes Poster components used to store and communicate data associated with different modules.

The Timer components trigger periodic, time-dependent computations. The Scheduler-Activity and Execution-Controller components provide execution control at module level i.e., scheduling for the different running services within every module.

3. Description of composite components as the composition of atomic components by using only interactions and priorities. This is possible because BIP is expressive enough for describing any kind of coordination by using only architectural constraints.

The entire functional layer contains 8 distinct modules. Their functionalities are (1) collecting data from the laser sensors (LaserRF), (2) generating an obstacle map (Aspect), (3) navigating using the near diagram approach (NDD), (4) managing the low level robot wheel controller (RFLEX), (5) emulating the communication with an orbiter (Antenna), (6) providing power and energy for the robot (Battery), (7) heating the robot in a low temperature environment (Heating) and (8) controlling the movement of two cameras (Platine).

Details about the characteristics of the software componentized in BIP are available in table 1. For example, the NDD module interconnects 27 atomic components (totalizing 152 control locations, 16 booleans and 11 integer variables) using 117 connectors. The whole functional layer uses 248 atomic components and 1068 connectors.

The BIP model of the functional layer has been formally verified for deadlock-freedom and other safety properties (e.g., data freshness) using D-Finder. We have been capable of checking safety and deadlock-freedom properties for all the modules. We have successively detected (and corrected) two deadlocks within Antenna and NDD, respectively. We also succeded to verify deadlock-freedom for composition of three modules (LaserRF, Aspect and NDD), and data freshness property between two modules (Aspect and NDD).

Table 1 provides results for checking deadlock-freedom of individual modules.

module	components	control locations	interactions	states	time (minutes)
SICK	43	213	202	$2^{20} \times 3^{29} \times 34$	1:22
Aspect	29	160	117	$2^{17} \times 3^{23}$	0:39
NDD	27	152	117	$2^{22} \times 3^{14} \times 5$	8:16
RFLEX	56	308	227	$2^{34} \times 3^{35} \times 1045$	9:39
Antenna	20	97	73	$2^{12} \times 3^{9} \times 13$	0:14
Battery	30	176	138	$2^{22} \times 3^{17} \times 5$	0:26
Heating	26	149	116	$2^{17} \times 3^{14} \times 145$	0:17
Platine	37	174	151	$2^{19} \times 3^{22} \times 35$	0:59

Table 1: Deadlock-freedom checking results for DALA modules

The BIP model has been also used to synthesize the execution controller that encodes and enforces safety properties, thereby facilitating the development of safe and dependable robotic architectures. In the initial version of this software a centralized hand-written controller (R2C) was used to ensure the proper execution of the services and to enforce the safety constraints on module interactions. In the BIP model, the proper execution order and the safety properties are already enforced by connectors and priorities. As an example, consider the requirement which states that the robot can navigate using the GoTo service of the NDD module only if the module POM has already executed successfully its Run service (which updates poster Pos). Such a requirement is enforced by adding a

connector between port trigger of the Goto service and port status of the Run service, and guarded by the status value.

Finally, we have run experiments with the code generated automatically from the BIP model on the DALA rover, and demonstrated via fault injections that the BIP engine successfully stops the robot from reaching undesired/unsafe states.

5 Discussion and Future Work

The paper presents the BIP component framework and an associated system design flow. BIP is unique for both its expressiveness and its rigorous semantics. In contrast to other formalisms, mathematical foundation on a minimal set of concepts and structuring principles does not hamper its effective use for modeling complex real-life systems.

BIP can model in a natural and direct manner various types of synchronization. Using less expressive frameworks e.g. based on a single composition operator, often leads to intractable models. For instance, BIP directly encompasses multiparty interaction between components. Modeling multiparty interaction in frameworks supporting only point-to-point interaction e.g. function call, requires the use of protocols. This leads to overly complex models with complicated coordination structure. We have compared descriptions of the DALA functional level application software written in the GeNoM object-oriented formalism based on C++ and corresponding models written in BIP after componentization. In contrast to object-oriented software, BIP models are easy to understand and analyse as the composition of integrated features. Furthermore, explicit use of automata in behavior ensures robustness of modules. They enforce the right order of execution of functions independently of their context of use.

Clear separation between architecture and behavior in BIP allows compositional and incremental analysis. This is advantageously exploited by D-Finder which separately analyses behavior of atomic components and extracts interaction invariants characterizing architectural constraints.

BIP supports a rigorous system design flow. A key idea is the application of correctness-preserving source-to-source transformations to progressively refine the application software model by taking into account hardware architecture constraints as well as mechanisms used for the coordination between processors in a distributed implementation. Verification is used to check essential properties as early as possible in the design flow. To avoid complexity limitations, the verification process is incremental and compositional. When the validity of a property is established for a model, the property holds for all the models obtained by transformation. The complexity of the transformations is linear with the size of the transformed models.

Using BIP as a unifying modeling framework allows to maintain the overall coherency of the design flow by comparing different architectural solutions and their properties. This is a significant difference with approaches using many different semantically unrelated formalisms e.g. for programming, hardware description, performance evaluation and where code generation and deployment is decoupled from verification and performance evaluation.

References

[1] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[2] A. Burns and A. Welling. *Real-Time Systems and Programming Languages*. Addison-Wesley, 2001. 3rd edition.

[3] S. Bliudze and J. Sifakis. A Notion of Glue Expressiveness for Component-Based Systems. In *CONCUR'08*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.

[4] David Garlan, Robert Monroe, and Dave Wile. ACME : An architecture description interchange language. In *CASCON'97*, pages 169–183, 1997. see also http://www.cs.cmu.edu/ acme/.

[5] J. Magee and J. Kramer. Dynamic structure in software ar- chitectures. In *SIGSOFT'96*, pages 3–14, 1996.

[6] OMG. *OMG Systems Modeling Language SysML (OMG SysML)*. Object Management Group, 2008.

[7] P. H. Feiler, B. Lewis, and S. Vestal. The SAE Architecture Analysis and Design Language (AADL) Standard: A basis for model-based architecture-driven embedded systems engineering. In *RTAS Workshop on Model-driven Embedded Systems*, pages 1–10, 2003. see also http://www.sae.org.

[8] 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.

[9] S. Bliudze and J. Sifakis. Causal semantics for the algebra of connectors. *Formal Methods in System Design*, 36(2):167–194, 2010.

[10] A. Basu, M. Gallien, C. Lesire, T-H. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis. Incremental Component-Based Construction and Verification of a Robotic System. In *ECAI'08*, volume 178 of *FAIA*, pages 631–635. IOS Press, 2008.

[11] S. Bensalem, M. Bozga, T-H. Nguyen., and J. Sifakis. Compositional Verification for Component-based Systems and Application. In *ATVA'08*, volume 5311 of *LNCS*, pages 64–79. Springer, 2008.

[12] S. Bensalem, M. Bozga, A. Legay, T-H. Nguyen, J. Sifakis, and R. Yan. Incremental Component-based Construction and Verification using Invariants. In *FMCAD'10*, pages 257–266. IEEE, 2010.

[13] S. Bensalem, M. Bozga, T-H. Nguyen, and J. Sifakis. D-Finder: A Tool for Compositional Deadlock Detection and Verification. In *CAV'09*, volume 5643 of *LNCS*. Springer, 2009.

[14] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. From high-level component-based models to distributed implementations. In *Embedded Software EMSOFT'10 Proceedings*. ACM, 2010.

[15] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[16] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, 1989.

[17] S. Fleury, M. Herrb, and R. Chatila. GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. In *IROS'97*, pages 842–848, 1997.