IEEE Software
www.computer.org/software

Rigorous Component-Based System Design Using the BIP Framework

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

Vol. 28, No. 3
May/June 2011

Rigorous Component-Based System Design Using the BIP Framework

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

// An autonomous robot case study illustrates the use of the behavior, interaction, priority (BIP) component framework as a unifying semantic model to ensure correctness of essential system design properties. //

SYSTEM DESIGN DIFFERS radically from pure software design in that it must account not only for functional requirements but also for extrafunctional requirements regarding the use of execution platform resources, such as time, memory, and energy. Meeting extrafunctional requirements is essential in embedded system design and requires evaluation of how design choices affect overall system behavior. It also implies a deep understanding of how the application software interacts with the underlying execution platform. Yet system designers currently lack rigorous techniques for deriving global models of a given system from models of its application software and execution platform.

We define a rigorous design flow as one that guarantees essential system properties. Most existing design flows that aspire to this goal privilege a unique programming model and associate it with a compilation chain that's 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]

At the Verimag Laboratory, we've been developing the behavior, interaction, priority (BIP) component framework to support a rigorous system design flow. The BIP framework is

- *model-based*, describing all software and systems according to a single semantic model. This maintains the flow's overall coherency by guaranteeing that a description at step *n*+*1* meets essential properties of a description at step *n*.
- *component-based*, providing a family of operators for building composite components from simpler components. This overcomes the poor expressiveness of theoretical frameworks based on a single operator, such as the product of automata or a function call.
- *tractable*, guaranteeing correctness

RELATED WORK IN COMPONENT FRAMEWORKS

BIP differs significantly from existing component frameworks for software engineering. These often use multithreaded programming and point-to-point interaction mechanisms, such as function calls, for coordination between components. In contrast, BIP executes atomic components concurrently and coordinates them in terms of high-level mechanisms such as protocols and scheduling policies.

Because BIP focuses on the organization of computation between components, it can be viewed as an architecture description language (ADL). Like other existing ADLs, such as Acme (www.cs.cmu.edu/~acme)[1] and Darwin,[2] BIP uses the connector concept to express coordination between components. Nonetheless, connectors in BIP are stateless. The architecture, consisting of connectors and priorities, is clearly distinguished from behavior.

Another significant difference from other frameworks is that BIP is intended for system modeling. It directly encompasses timing and resource management. Other system modeling formalisms either seek generality to the detriment of rigorousness, such as (Systems Modeling Language (SysML)[3] and (Architecture Analysis and Design Language (AADL; http://standards.sae.org/as5506a),[4] or limit their scope to specific computation models, such as Ptolemy.[5]

References

1. D. Garlan, R. Monroe, and D. Wile, "Acme: An Architecture Description Interchange Language, *Proc. 1997 Conf. Centre for Advanced Studies on Collaborative Research* (CASCON 97), IBM Press, 1997, pp. 169–183.
2. J. Magee and J. Kramer, "Dynamic Structure in Software Architectures," *Proc. 4th ACM SIGSOFT Symp. Foundations of Software Eng.* (SIGSOFT 96), ACM Press, 1996, pp. 3–14.
3. *OMG Systems Modeling Language SysML (OMG SysML)*, v. 1.2, Object Management Group, 2010; www.omg.org/spec/SysML/1.2.
4. P.H. Feiler, B. Lewis, and S. Vestal, "The SAE Architecture Analysis and Design Language (AADL): A Standard for Engineering Performance Critical Systems," *IEEE Conf. Computer Aided Control Systems Design (CACSD 06), IEEE Press, 2006, pp. 1206–1211.*
5. J. Eker et al., "Taming Heterogeneity: The Ptolemy Approach," *Proc. IEEE*, vol. 91, no. 1, 2003, pp. 127–144.

by construction and thereby avoiding monolithic a posteriori verification as much as possible.

BIP supports the construction of composite, hierarchically structured components from atomic components characterized by their behavior and interfaces. It lets developers compose components by layered application of interactions and priorities. This enables an expressiveness unmatched by any other existing formalism (see the related work sidebar).[3] Architecture is a first-class concept in BIP, with well-defined semantics that system designers can analyze and transform.

In this article, we present the BIP component framework, highlighting its design flow and the main steps for deriving correct implementations from a given application's software and a target platform. Case study results from a BIP implementation of the Dala autonomous robot demonstrate its effectiveness.

The BIP Component Framework

The BIP framework uses connectors to specify possible *interactions* between components and *priorities* to select among possible interactions. Interactions express synchronization constraints between the composed components' activities, and priorities filter possible interactions to steer system evolution toward meeting performance requirements. The combination of interactions and priorities is the source of BIP's expressive power. It defines a clean, abstract concept of architecture separate from behavior.

Atomic components are finite-state automata or Petri nets extended with data and ports. *Ports* are action names that can be associated with data and used for interactions with other components. *States* denote control locations where components wait for interactions. A *transition* is an execution step, labeled by a port, from one control location to another. Each transition has an associated guard and action—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: first, execution of the interaction involving the port, which is a synchronization between several components with possible data exchange, followed by execution of the action associated with the transition.

Example 1: Atomic Components

The right side of Figure 1 shows two atomic components, Service-Controller and Activity, for the Dala robot controller. Activity wraps the long-time computation of some specific application function. Service-Controller provides execution control (triggering, canceling, error control, and so on) over the associated Activity component. For simplicity's sake, the figure presents only the skeleton control behavior (ports and

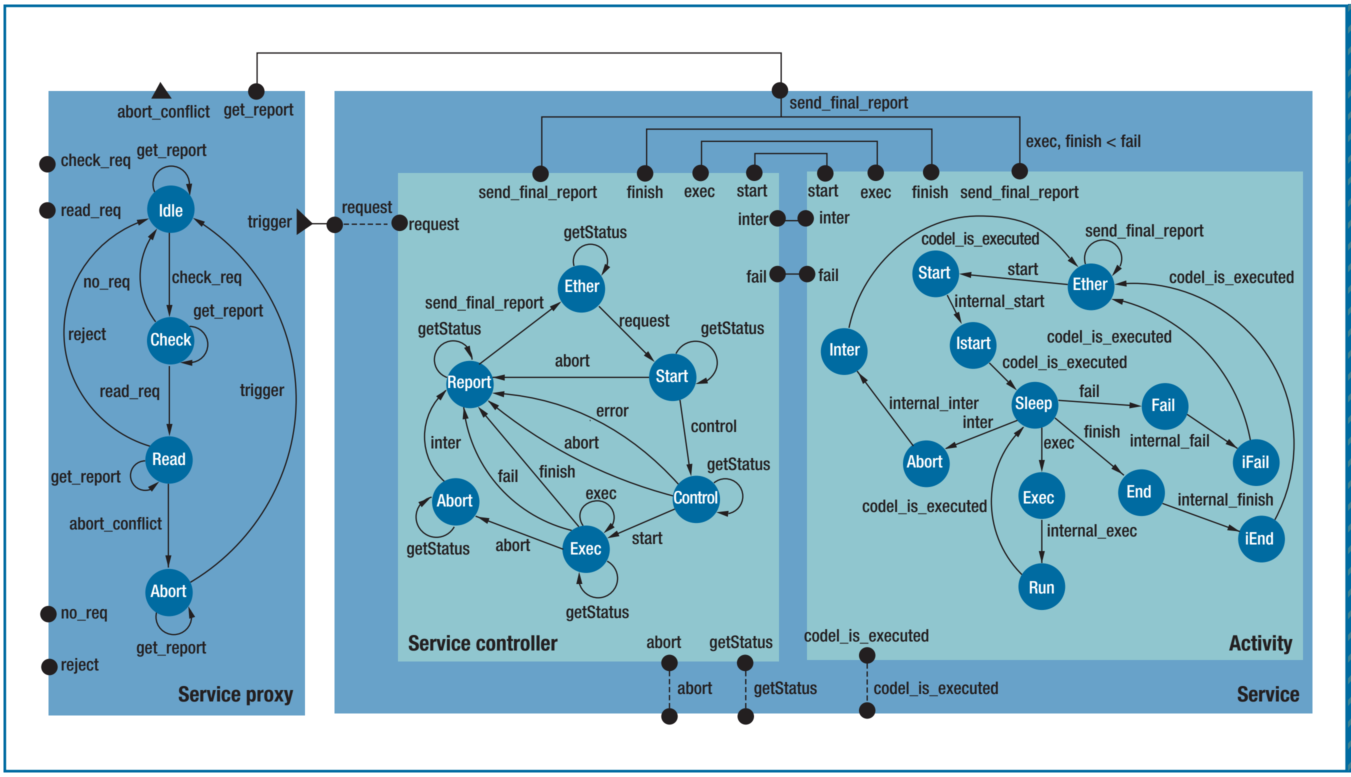

FIGURE 1. BIP component schematic. The Service composite component on the left, for a Dala robot service, consists of two atomic components: Service Controller and Activity.

transitions) and omits the data and associated code. For example, Activity is initialized (start transition) and then executes its associated functions (exec, internal_exec transitions). The execution might finish normally (finish transition), fail (fail transition) or be interrupted (inter transition).

Composite components are defined by assembling constituent components (atomic or composite) using connectors. *Connectors* define relationships between ports of interacting components. They represent sets of interactions—that is, nonempty sets of ports that must be jointly executed. Within a connector, an interaction can occur in two situations: when all involved ports are ready to participate (strong synchronization) or when a port triggers the interaction without waiting for other ports (broadcast).

The valid interactions within connectors are formally defined by algebraic expressions on ports using a binary *fusion* operator and a unary *typing* operator.[4] Typing associates connector ends (ports or connectors) to synchronization types: *trigger* (active port, initiates broadcast) or *synchron* (passive port). Moreover, every connector interaction is associated with a guard and a data transfer function. An interaction can be executed only when its guard is true. Its execution consists of transferring the data and then, notifying the components involved in the interaction.

Finally, the priorities for choosing between simultaneously enabled interactions within a BIP component are defined as rules, each consisting of a 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: Composite Components

The Service component on the left side of Figure 1 is a composite of Activity and Service-Controller through connectors that enforce strong synchronizations of several actions (for example, start, exec, finish, fail). The connectors allow the Service-Controller to initiate and follow the computation performed within Activity. The composite component is equipped with priorities to privilege the execution of a fail interaction—that is, error handling—over finish and exec interactions, which correspond to normal behavior.

The example also illustrates the encapsulation principle used in BIP. Service is further composed with the Service-Proxy component by using the ports available on the Service interface, which are explicitly redirected either to subcomponent ports or to inner connectors. The trigger-request connector between Service-Proxy and Service illustrates a broadcast initiated by the trigger port. A trigger action is either executed alone or synchronized with request actions when they're enabled.

A concrete modeling language

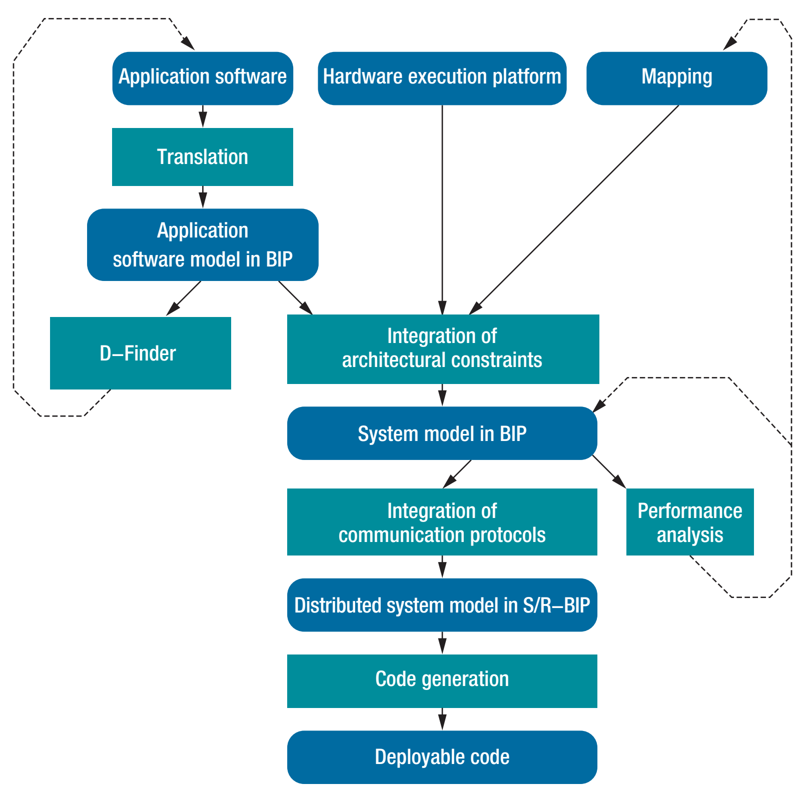
Application software
Hardware execution platform
Mapping
Translation
Application software model in BIP
D–Finder
Integration of architectural constraints
System model in BIP
Integration of communication protocols
Performance analysis
Distributed system model in S/R–BIP
Code generation
Deployable code

FIGURE 2. BIP design flow. An implementation—that is, deployable code—is generated from the application software, a model of the hardware platform, and a mapping.

supports the BIP framework. The BIP language leverages C++-style variables and data-type declarations, expressions, and statements. It also provides additional structural syntactic constructs for defining component behavior and describing connectors and priorities. Moreover, it provides constructs for handling parametric and hierarchical descriptions and for expressing timing constraints associated with behavior.

The BIP Design Flow

Figure 2 illustrates a rigorous system design flow that uses BIP as a unifying semantic model to ensure consistency between the different design steps.

The design flow involves four distinct steps that translate the application software into a BIP model and progressively derive an implementation by applying source-to-source transformations:

1. *Translating the application software into a BIP model.* The translation focuses on the definition of adequate interfaces. It encapsulates and reuses the application software's data structures and functions.
2. *Integrating architectural constraints in the application software model.* The integration derives a system model in BIP from a model of the hardware target platform and a mapping.
3. *Translating interactions and priorities* of the system model in terms of protocols using send/receive primitives.
4. *Generating deployable C code* from which an implementation can be obtained.

The transformations are "correct by construction" because the obtained BIP models are observationally equivalent to the original model. In particular, they preserve the application software's safety properties. Furthermore, we developed D-Finder, a verification tool that checks essential safety properties of the application software.

Figure 3 shows an extensible toolset that supports the entire BIP design flow, including D-Finder.

Translating Application Software into BIP

The first step in the design flow consists of generating 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 steps for a given application software written in a language *L*:

1. *Translating the source language* L*'s atomic components* into BIP components. The translation focuses on the definition of adequate interfaces. It encapsulates and reuses the application software's data structures and functions.
2. *Translating the coordination mechanisms between application software components* into the target BIP model's connectors and priorities.
3. *Generating a BIP component that models* L*'s operational semantics.* This component plays the role of an engine coordinating the execution of the application software components.

We developed BIP model generators for several programming models used by embedded system developers (the source-to-source transformers in Figure 3). The generated models preserve the structure of the initial programs, their size is linear with respect to the initial program size, and they're easy for the system developers to understand.

Using D-Finder for Compositional Verification

D-Finder bases its compositional verification method on computing invariants.[5] It computes increasingly stronger invariants for composite components as conjunctions of atomic components' local invariants and interaction invariants that characterize the composition glue. Static analysis of atomic components generates the local component invariants. Interaction invariants are generated from abstractions of the composite component to be verified.

We recently improved this method to take advantage of the incremental system design process, which proceeds by adding new interactions to a component under construction. Each time a new interaction is added, it's possible to verify whether the resulting component violates a given property and so discover design errors as they appear. The incremental verification technique[6] uses sufficient conditions to ensure the preservation of invariants when new interactions are added during the component construction process. If these conditions aren't satisfied, D-Finder generates new invariants by reusing invariants of the constituent components. Reusing invariants considerably reduces the verification effort.

The D-Finder tool implements the compositional verification techniques for checking the deadlock-freedom of systems described in BIP.[7] Experimental results on classical benchmarks show that D-Finder can run exponentially faster than existing monolithic verification tools, such as NuSMV.

Generating Implementations

The BIP toolset offers several compilation chains, targeting different execution platforms. To implement BIP on single-core platforms requires using *engines*—that is, dedicated middleware for execution of the C++ code autogenerated from BIP descriptions. BIP currently provides two engines: one for real-time single-thread and one for multithread execution. For multithread execution, each atomic component is assigned to a thread, with the engine itself being a thread. Communication occurs only between atomic components and the engine—never directly between different atomic components.

To generate distributed implementations from BIP models, we transform them into send/receive (S/R)-BIP models,[8] a subclass of models in which protocols using S/R primitives replace multiparty interactions. From S/R-BIP models and a mapping of atomic components into a platform's processing elements, we generate efficient C/C++ or message passing interface (MPI) code.

We use the following sequence of correct-by-construction transformations, which preserve observational equivalence.[8] First, given a user-defined partition of a BIP system model's interactions, we break the atomicity of its transitions by separating the interaction from the computation. We then replace multiparty interactions with protocols that use S/R primitives. Moreover, we structure the target S/R-BIP model in three layers:

- The *component layer* consists of the original model's atomic components in which each port involved in strong interactions is replaced by a pair of corresponding S/R ports.
- The *interaction protocol layer* consists of a set of components, each of which manages a class of the par-

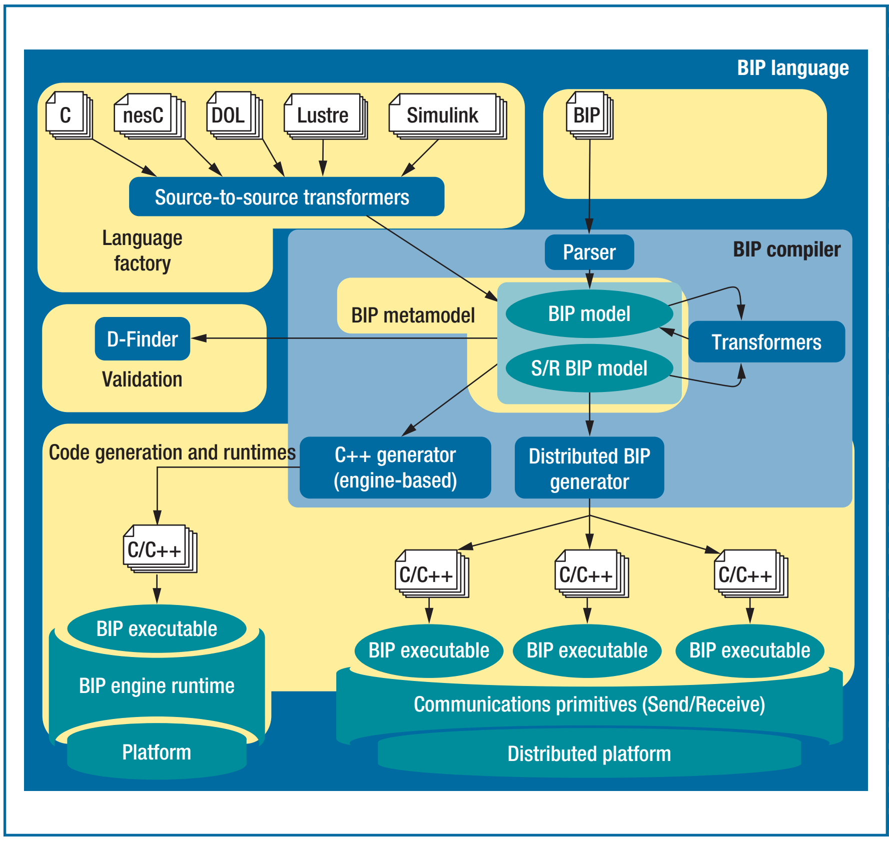

FIGURE 3. BIP toolset. The tools include translators from various programming models, verification tools, source-to-source transformers, and C/C++ code generators for BIP models.

Functional Layer	::=	(*Module*)+
Module	::=	(*Service*)+ . (*Execution-Task*) . (Poster)+
Service	::=	(Service-Controller) . (Activity)
Execution-Task	::=	(Timer) . (Scheduler-Activity) . (Execution-Controller)

FIGURE 4. Hierarchical decomposition of Functional Layer into components.

tition's interactions. The protocol detects whether interactions are enabled and executes them after resolving conflicts either locally or with assistance from the third layer.
- The *conflict resolution protocol layer* consists of distributed algorithms for resolving conflicts as requested by the interaction protocol layer. The conflict resolution protocol, which basically solves a committee coordination problem, uses either a fully centralized, token-ring, or dining philosophers algorithm.[9,10]

In the second step, we use the three-layer S/R-BIP model and a mapping of its atomic components on processors to generate either an MPI program or a set of plain C/C++ programs that use TCP/IP communication. This process statically composes atomic components running on the same processor to obtain a single observationally equivalent component, and reduce coordination overhead at runtime.

Case Study: Dala Robot Controller

We used BIP to develop a new version of the functional layer of the Dala robot controller from an existing version developed using the GeNoM framework.[11] We presented preliminary results of this work elsewhere,[12] including the complete modeling of the functional layer, its functional verification, and the synthesis of a correct-by-construction software controller. Here, we briefly introduce the model and summarize our latest results on the verification of deadlock-freedom.

Functional layer design in BIP involves three steps:

1. *Hierarchical decomposition into components.* A tree structure represents the overall architecture with its root corresponding to the functional layer and its leaves to atomic components. The grammar in Figure 4 shows how to obtain the designed system as the incremental composition of components.
2. *Description of each atomic component's behavior.* In addition to component abstractions, such as those we described for the Service-Controller and Activity components (see Figure 1), the functional layer includes Poster components to store and communicate data associated with different modules; Timer components that trigger periodic, time-dependent computations; Scheduler-Activity and Execution-Controller components to control execution control at the module level.
3. *Description of composite components.* Atomic components are composed using only interactions and priorities because BIP is expressive enough to describe any kind of coordination solely through architectural constraints.

The entire functional layer contains eight distinct modules. Their functionalities are

- collecting data from the laser sensors (LaserRF),
- generating an obstacle map (Aspect),
- navigating using the near diagram approach (NDD),
- managing the low-level robot wheel controller (Rflex),
- emulating the communication with an orbiter (Antenna),
- providing power and energy for the robot (Battery),
- heating the robot in a low-temperature environment (Heating), and
- controlling the movement of two cameras (Platine).

Table 1 presents characteristics of the software componentized in BIP. For example, the NDD module uses 117 connectors to interconnect 27 atomic components comprising 152 control locations. This module consists of 5,343 lines of BIP code and calls external functions totalizing 51,653 lines of C/C++ code. In total, the functional layer modules use 268 atomic components and 1,141 connectors. The whole model has 37,294 lines of BIP code and calls more than 279,818 lines of external C/C++ code.

We used D-Finder to formally verify the functional layer's BIP model for deadlock-freedom and other safety properties, such as data freshness. We have the capability to check safety and deadlock freedom properties for all the modules. We successively detected (and corrected) two deadlocks, one in Antenna and the other in NDD. We also successfully verified deadlock freedom for composition of three modules (LaserRF, Aspect, and NDD), and data freshness between two modules (Aspect and NDD). Table 1 includes verification times for checking deadlock freedom of individual modules as well as other characteristics such as the number of atomic components, control locations, lines of BIP code, and lines of C/C++ code.

We also used the BIP model to synthesize the execution controller that encodes and enforces safety properties, thereby facilitating the development of safe, dependable robotic architectures. The initial version of this software used a centralized, hand-written, request-and-report checker (R2C) to ensure the proper execution of services and to en-

TABLE 1

Deadlock-freedom-checking results for Dala robot controller modules.

Module	Atomic components	Control locations	Connectors	BIP LoC	C/C++ LoC	Estimated state space size	Verification time (minutes)
LaserRF	43	213	202	5,343	51,653	$2^{20} \times 3^{29} \times 34$	1:22
Aspect	29	160	117	3,029	30,204	$2^{17} \times 3^{23}$	0:39
NDD	27	152	117	4,013	32,600	$2^{22} \times 3^{14} \times 5$	8:16
Rflex	56	308	227	8,244	57,442	$2^{34} \times 3^{35} \times 1045$	9:39
Antenna	20	97	73	1,645	16,501	$2^{12} \times 3^{9} \times 13$	0:14
Battery	30	176	138	3,898	21,527	$2^{22} \times 3^{17} \times 5$	0:26
Heating	26	149	116	2,453	18,380	$2^{17} \times 3^{14} \times 145$	0:17
Platine	37	174	151	8,669	51,511	$2^{19} \times 3^{22} \times 35$	0:59

force the safety constraints on module interactions. The BIP model inherently enforced these constraints by connectors and priorities.

As an example, consider a requirement for the robot to navigate using the NDD module's GoTo service only if services Init, SetParams, and SetSpeed have already executed successfully. BIP enforces this requirement by adding a connector between the GoTo service's request port and the other ports' getStatus ports. The status values guard and may prevent the triggering of the GoTo service.

Finally, we ran experiments on the code generated automatically from the Dala rover's BIP model, using fault injections to demonstrate that the BIP engine successfully stops the robot from reaching undesired or unsafe states.

BIP's rigorous semantics and expressive power are unique among component frameworks and associated system design flows. In contrast to other formalisms, BIP's mathematical foundation on a minimal concept set and structuring principles doesn't hamper its effective use for modeling complex real-life systems. In contrast to less expressive frameworks, it models various synchronization types in a natural and direct manner. BIP directly encompasses multiparty interaction between components, avoiding the complexities necessary in frameworks supporting only point-to-point interactions. In contrast to object-oriented software, BIP models are easy to understand and analyze as compositions of integrated features. Furthermore, their explicit use of automata in behavior ensures module robustness by enforcing the right execution order of functions independently of their use context.

Progressively refining the application software model by applying correctness-preserving source-to-source transformations takes hardware architecture constraints into account as well as coordination mechanisms between processors in a distributed implementation. Essential properties are verified as early as possible in the design flow in an incremental, compositional verification process that avoids complexity limitations. When the validity of a property is established for a model, the property holds for all the models obtained by transformation. Transformation complexity is linear with the size of the transformed models.

As a unifying modeling framework, BIP can maintain a design flow's overall coherency by comparing different architectural solutions and their properties. This differs significantly from approaches that decouple code generation and deployment from validation and use many different, semantically unrelated formalisms for programming, hardware description, and simulation.

Acknowledgments

The research leading to these results received funding from the European Community's Seventh Framework Programme under grant agreement 248776 and from the Artemis Joint Undertaking grant agreement 2009-1-100230.

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*, 3rd ed., Addison-Wesley, 2001.
3. S. Bliudze and J. Sifakis, "A Notion of Glue Expressiveness for Component-Based Systems," *Proc. 19th Int'l Conf. Concurrency Theory* (CONCUR 08), LNCS 5201, Springer, pp. 508–522.
4. S. Bliudze and J. Sifakis, "Causal Semantics for the Algebra of Connectors," *Formal Methods in System Design*, vol. 36, no. 2, 2010, pp. 167–194.
5. S. Bensalem et al., "Compositional Verification for Component-Based Systems and Application," *Proc. 6th Int'l Symp. Automated Technology for Verification and Analysis* (ATVA 08), LNCS 5311, Springer, 2008, pp. 64–79.
6. S. Bensalem et al., "Incremental Component-

Based Construction and Verification Using Invariants," *Proc. Formal Methods on Computer-Aided Design* (FMCAD 10), Formal Methods in Computer-Aided Design, 2010, pp. 257–266; http://fmcad10.iaik.tugraz.at/Papers/FMCAD10.pdf.
7. S. Bensalem et al., "D-Finder: A Tool for Compositional Deadlock Detection and Verification," *Proc. 21st Int'l Conf. Computer Aided Verification* (CAV 09), LNCS 5643, Springer, 2009, pp. 614–619.
8. B. Bonakdarpour et al., "From High-Level Component-Based Models to Distributed Implementations," *Proc. 10th Int'l Conf. Embedded Software* (EmSoft 10), ACM Press, 2010, pp. 209–281.
9. K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley Longman, 1988.
10. R. Bagrodia, "Process Synchronization: Design and Performance Evaluation of Distributed Algorithms," *IEEE Trans. Software Eng.*, vol. 15, no. 9, 1989, pp. 1053–1065.
11. S. Fleury, M. Herrb, and R. Chatila, "GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture," *Proc. 1997 IEEE/RSJ Int'l Conf. Intelligent Robots and Systems* (IROS 97), IEEE Press, 1997, pp. 842–848.
12. A. Basu et al., "Incremental Component-Based Construction and Verification of a Robotic System," *Proc. 18th European Conf. Artificial Intelligence* (ECAI 08), IOS Press, 2008, pp. 631–635.

ABOUT THE AUTHORS

ANANDA BASU is a postdoctoral researcher at the Verimag Laboratory. His research focuses on system-level modeling and performance analysis of mixed software-hardware systems and deriving their implementations on target hardware platforms. His interests include component-based modeling of embedded systems, in particular modeling and simulation frameworks for complex and heterogeneous systems. Basu has a PhD in computer science from the University Joseph Fourier, Grenoble. Contact him at ananda.basu@imag.fr.

SADDEK BENSALEM is a professor at the University of Joseph Fourier. His research focus is modeling and validation of real-time systems, including component-based modeling, verification, and synthesis of distributed systems. Bensalem has a PhD in computer science from INP Grenoble (Institut National Polytechnique de Grenoble). Contact him at saddek.bensalem@imag.fr.

MARIUS BOZGA is a research engineer at CNRS (Centre National de la Recherche Scientifique) and a member of the Verimag Laboratory. His research interests focus on component-based design for distributed real-time systems and include formal models for components, model-based design and implementation, and automatic validation methods and tools. Bozga has a PhD in computer science from the University of Grenoble. Contact him at marius.bozga@imag.fr.

JACQUES COMBAZ is a research engineer at CNRS (Centre National de la Recherche Scientifique) and a member of the Verimag Laboratory. His research interests include the design of adaptive applications and real-time systems. He developed the real-time engine for BIP programs. Combaz has a PhD in mathematics and computer science from the University of Grenoble. Contact him at jacques.combaz@imag.fr.

MOHAMAD JABER is a postdoctoral researcher at the Verimag Laboratory. His research focuses on component-based design and implementation. Jaber has a PhD in computer science from the University of Grenoble. Contact him at mohamad.jaber@imag.fr.

THANH HUNG NGUYEN is a postdoctoral researcher at the Verimag Laboratory. His research interests are in the modeling and verification of component-based systems. Nguyen has a PhD in computer science from the University of Grenoble. Contact him at thanh-hung.nguyen@imag.fr.

JOSEPH SIFAKIS is a CNRS researcher and founder of the Verimag Laboratory. His research includes pioneering work on theoretical and practical aspects of concurrent systems specification and verification. His current interests include component-based design, modeling, and analysis of real-time systems with a focus on correct-by-construction techniques. Sifakis has a PhD in computer science from the University of Grenoble. In 2007, he shared the Turing Award with Ed Clarke and Allen Emerson for their contribution to model checking. Contact him at joseph.sifakis@imag.fr.