Rigorous System Design: The BIP Approach

Ananda Basu[1], Saddek Bensalem[1,2], Marius Bozga[1], Paraskevas Bourgos[1], and Joseph Sifakis[1]

[1] Verimag Laboratory, Université Joseph Fourier Grenoble, CNRS
[2] CEA-Leti, MINATEC Campus, Grenoble France

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. This is essential for ensuring the overall coherency and correctness. The paper introduces a rigorous design flow based on the BIP (Behavior, Interaction, Priority) component framework [1]. This design flow relies on several, tool-supported, source-to-source transformations allowing to progressively and correctly transform high level application software towards efficient implementations for specific platforms.

1 System Design

Traditional engineering disciplines such as civil or mechanical engineering are based on solid theory for building artifacts with predictable behavior over their life-time. In contrast, we lack similar constructivity results for computing engineering: computer science provides only partial answers to particular system design problems. With few exceptions in this domain, predictability is impossible to guarantee at design time and therefore, a posteriori validation remains the only means for ensuring their correct operation.

System design is facing several difficulties, mainly due to our inability to predict the behavior of an application software running on a given platform. Usually, systems are built by reusing and assembling components that are, simpler subsystems. This is the only way to master complexity and to ensure correctness of the overall design, while maintaining or increasing productivity. However, system level integration becomes extremely hard because components are usually highly heterogeneous: they have different characteristics, are often developed using different technologies, and highlight different features from different viewpoints. Other difficulties stem from current design approaches, often empirical and based on expertise and experience of design teams. Naturally, designers attempt to solve new problems by reusing, extending and improving existing solutions proven to be efficient and robust. This favors component reuse and avoids re-inventing and re-discovering designs. Nevertheless, on a longer term perspective, this may also be counter-productive: designers are not always able to adapt in a satisfactory manner to new requirements. Moreover, they a priori exclude better solutions simply because they do not fit their know-how.

Z. Kotásek et al. (Eds.): MEMICS 2011, LNCS 7119, pp. 1–19, 2012.

System design is the process leading to a mixed software/hardware 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 should 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.

A system design flow consists of steps starting from specifications and leading to an implementation on a given execution platform. It involves the use of methods and tools for progressively deriving the implementation by making adequate design choices.

We consider that a system design flow must meet the following essential requirements:

– *Correctness*: This means that the designed system meets its specifications. Ensuring correctness requires that the design flow relies on models with well-defined semantics. The models should consistently encompass system description at different levels of abstraction from application software to its implementation. Correctness can be achieved by application of verification techniques. It is desirable that if some specifications are met at some step of the design flow, they are preserved in all the subsequent steps.

– *Productivity*: This can be achieved by system design flows

  – providing high level domain-specific languages for ease of expression
  – allowing reuse of components and the development of component-based solutions
  – integrating tools for programming, validation and code generation

– *Performance*: The design flow must allow the satisfaction of extra-functional properties regarding optimal resource management. This means that resources such as memory, time and energy are first class concepts encompassed by formal models. Moreover, it should be possible to analyze and evaluate efficiency in using resources as early as possible along the design flow. Unfortunately, most of the widely used modeling formalisms offer only syntactic sugar for expressing timing constraints and scheduling policies. Lack of adequate semantic models does not allow consistency checking for timing requirements, or meaningful composition of features.

– *Parcimony*: The design flow should not enforce any particular programming or execution model. Very often system designers privilege specific programming models or implementation principles that a priori exclude efficient solutions. They program in low level languages that do not help discover parallelism or non determinism and enforce strictly sequential execution. For instance, programming multimedia applications in plain C may lead to designs obscuring the inherent functional parallelism and involving built-in scheduling mechanisms that are not optimal. It is essential that designers use adequate programming models. Furthermore, design choices should be driven only by system specifications to obtain the best possible implementation.

We call *rigorous* a design flow which allows guaranteeing essential properties of the specifications. Most of the 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 [2]. Alternatively, real-time programming based on scheduling theory for periodic tasks, targets dedicated real-time multitasking platforms [3].

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 $n-1$. This means in particular that the semantic model is expressive enough to directly encompasses various types of component heterogeneity arising along the design flow [4]:

  – Heterogeneity of computation: The semantic model should encompass both synchronous and asynchronous computation by using adequate coordination mechanisms. This should allow in particular, modeling mixed hardware/software systems.
  – Heterogeneity of interaction: The semantic model should enable natural and direct description of various mechanisms used to coordinate execution of components including semaphores, rendezvous, broadcast, method call, etc.
  – Heterogeneity of abstraction: The semantic model should support the description of a system at different abstraction levels from its application software to its implementation. This makes possible the definition of a clear correspondence between the description of an untimed platform-independent behavior and the corresponding timed and platform-dependent implementation.

– 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.

For instance, if the composition is by strong synchronization (rendezvous) modeling broadcast requires an extra component to choose amongst the possible strong synchronizations a maximal one. We need frameworks providing families of composition operators for natural and direct description of coordination mechanisms such as protocols, schedulers and buses.
– It should rely on tractable theory for guaranteeing *correctness by construction* to avoid as much as possible monolithic a posteriori verification. Such a theory is based on two types of rules:

- Compositionality rules for inferring global properties of composite components from the properties of composed components e.g. if a set of components are deadlock-free then for a certain type of composition the obtained composite components is deadlock-free too. A special and very useful case of compositionality is when a behavioral equivalence relation between components is a congruence [5]. In that case, substituting a component in a system model by a behaviorally equivalent component leads to an equivalent model.
- Composability rules ensuring that essential properties of a component are preserved when it is used to build composite components.

The paper presents a rigorous design flow based on the BIP (Behavior, Interaction, Priority) component framework [1]. It is organized as follows. Section 2 introduces the underlying modeling framework and the main steps of the design flow. Subsection 2.1 presents the BIP language. Subsection 2.2 explains the principle of translating different programming models into BIP. Subsection 2.3 introduces a method for compositional verification of BIP programs, especially used for checking deadlock-freedom. Subsection 2.4 presents a method for integrating architectural constraints into the BIP model of application software and subsection 2.5 presents a method for generating distributed implementations. The design flow is illustrated through non trivial examples in section 3. In section 4, we conclude and discuss future work directions.

2 The BIP Design Flow

BIP [1] (Behavior, Interaction, Priority) is a general framework encompassing rigorous design. It uses the BIP language and an associated toolset supporting the design flow. The BIP language is a notation which allows building complex systems by coordinating the behavior of a set of atomic components. Behavior is described as a finite-state automaton extended with data and functions described in C. The transitions of the Petri are labelled with guards (conditions on the state of a component and its environment) as well as functions that describe computations on local data. The description of coordination between components is layered. The first layer describes the interactions between components. The second layer describes dynamic priorities between the interactions and is

used to express scheduling policies. The combination of interactions and priorities characterizes the overall architecture of a component. It confers BIP strong expressiveness that cannot be matched by other languages [6]. BIP has clean operational semantics that describe the behavior of a composite component as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model (transition systems) and its implementation.

The BIP design flow uses a single language to ensure consistency between the different design steps. This is mainly achieved by applying source-to-source transformations between refined system models. These transformations are proven correct-by-construction, that means, they preserve observational equivalence and consequently essential safety properties. Functional verification is applied only to high level models for checking safety properties such as invariants and deadlock-freedom. To avoid inherent complexity limitations, the verification method applies compositionality techniques implemented in the D-Finder tool.

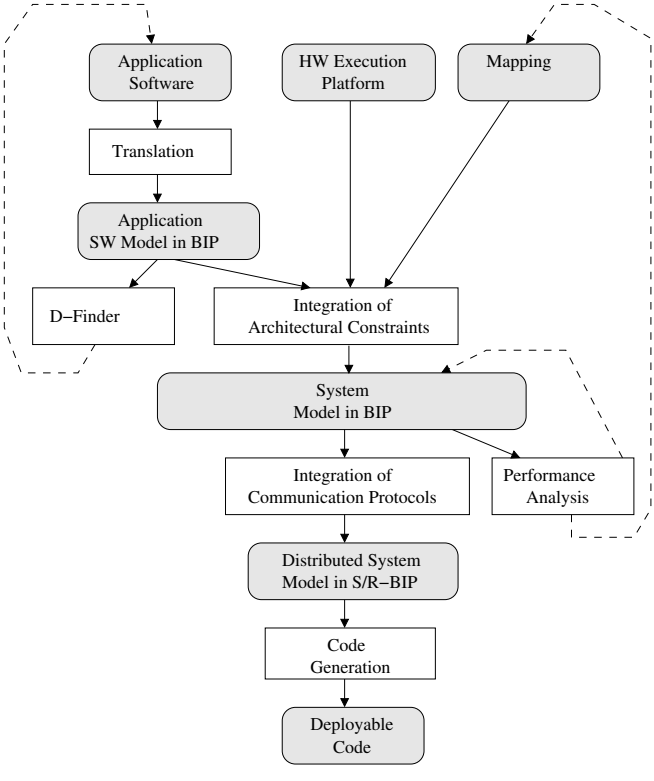

Fig. 1. BIP Design Flow

The design flow involves 4 distinct steps:

1. The translation of the application software into a BIP model. This allows its representation in a rigorous semantic framework. There exist translations of several programming models into BIP including synchronous, data-flow and event driven models.
2. The generation of an abstract system model from the BIP model representing the application software, a model of the target execution platform as well as a mapping of the atomic components of the application software model into processing elements of the platform. The obtained model takes into account hardware architecture constraints and execution times of atomic actions. Architecture constraints include mutual exclusion induced from sharing physical resources such as buses, memories and processors as well as scheduling policies seeking optimal use of these resources.
3. The generation of a concrete system model obtained from the abstract model by expressing high level coordination mechanisms e.g., interactions and priorities by using primitives of the execution platform. This transformation usually involves the replacement of atomic multiparty interactions by protocols using asynchronous message passing (send/receive primitives) and arbiters ensuring overall coherency e.g. non interference of protocols implementing different interactions.
4. The generation of executable, monolithic C/C++ or MPI code from sets of interacting components executed by the same processor. This allows efficient implementation by avoiding overhead due to coordination between components.

The BIP design flow is entirely supported by the BIP language and its associated toolset, which includes translators from various programming models, verification tools, source-to-source transformers and C/C++-code generators for BIP models.

2.1 The BIP Language

The BIP language, introduced in [1], supports a design flow for building systems from *atomic components*. It ses *connectors*, to specify possible interaction patterns etween components, and *priorities*, to select amongst possible interactions.

Atomic components are finite-state automata that are extended with variables and ports. Variables are used to store local data. Ports are action names, and may be associated with variables. They are used for interaction with other components. States denote control locations at which the components await for interaction. A transition is a 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 computation defined on local variables. In BIP, data and their transformations are written in C.

For a given valuation of variables, a transition can be executed if the guard evaluates to true and some *interaction* involving the 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 internal computation associated with the transition.

Composite components are defined by assembling sub-components (atomic or composite) using *connectors*. Connectors relate ports from different subcomponents. They represent sets of interactions, that are, non-empty sets of ports that have to be jointly executed. For every such interaction, the connector provides the guard and the data transfer, that are, respectively, an enabling condition and an exchange of data across the ports involved in the interaction.

Finally, *priorities* provide a mean to coordinate the execution of interactions within a BIP system. They are used to specify scheduling or similar arbitration policies between simultaneously enabled interactions. More concretely, priorities 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.

Figure 2 shows a graphical representation of an example model in BIP. It consists of atomic components *Sender*, *Receiver1* and *Receiver2*. The behavior of *Sender* is described as an automaton with control locations *Idle* and *Active*. It communicates through port s which exports the variable x. Components *Receiver1* and *Receiver2* are composed by the connector *C1*, which represents a rendezvous interaction between ports *r1* and *r2*, leading to the composite component *Receivers*. The composite exports *C1* as port r. As a result of the data transfer in *C1*, the sum of the local variables *y1* and *y2* is exported as v through the port r, and *y1*, *y2* eventually receive the value of v. The system is the composition of *Sender* and *Receivers* using the connector *C2* which represents a broadcast interaction from the *Sender* to the *Receivers*. When the broadcast occurs, as a result of the composed data transfer, the *Sender* gets the sum of *y1* and *y2*, and each *Receiver* gets the value x from the *Sender*.

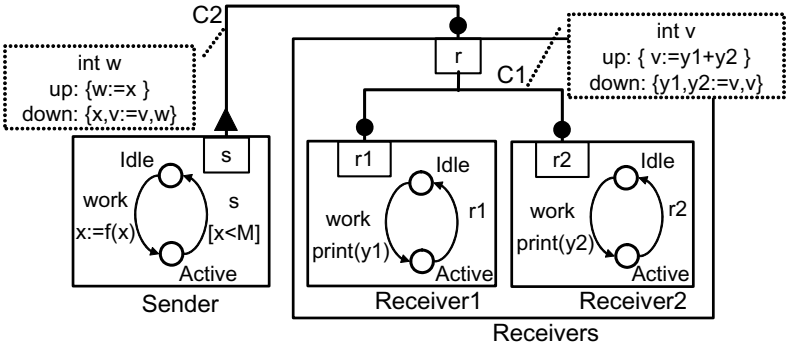

Fig. 2. An example of a BIP system

2.2 Translating Application Software into BIP

The first step in our 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. The principle of the method is depicted in Figure 3. 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 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.

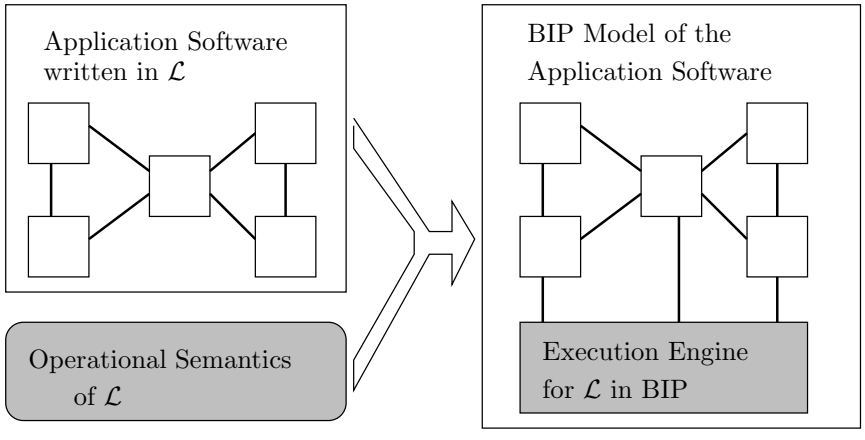

Fig. 3. Principle of translating application software

We have developed BIP model generators for several programming models used by embedded system developers including Lustre [2], MATLAB/Simulink^{TM}, the *Architecture Analysis and Design Language* AADL, NesC/TinyOS, the *Distributed Operation Layer* DOL [7], the programming model *GeNoM* [8], etc. 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. These facts confirm the adequacy and expressive power of BIP.

2.3 Compositional Verification by Using D-Finder

Monolithic verification of component-based systems often requires computing the product of their atomic components by using interleaving and synchronization. In general, the size of this product is prohibitive and cannot be handled without manual intervention. In a series of recent works, it has been advocated that *compositional techniques* could be used to cope with state explosion in verification of concurrent systems. A key issue is the existence of composition frameworks ensuring *compositionality*, which is, establishing global properties of composite components from properties of their constituent components.

A compositional verification method for BIP based on invariant computation is presented in [9]. This method computes 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 (and individual) analysis of atomic components. Interaction invariants are generated from abstractions of the composite to be verified.

The method is based on the following rule:

$$\frac{\{B_i < \Phi_i >\}_i,\ \Psi \in II(\|_\gamma\{B_i\}_i, \{\Phi_i\}_i),\ (\bigwedge_i \Phi_i) \wedge \Psi \Rightarrow \Phi}{\|_\gamma\{B_i\}_i < \Phi >}$$

The rule allows to prove invariance of property Φ for systems obtained by using an n-ary composition operation $||$ parameterized by a set of interactions γ. Φ is implied by the conjunction of invariants Φ_i of components B_i and an *interaction invariant* Ψ. The latter expresses constraints on the global state space induced by interactions. In [9], we have shown that Ψ can be computed automatically from abstractions of the system to be verified. These are the composition of finite state abstractions of the components B_i with respect to their invariants Φ_i.

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 method [10] 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.

The above methods have been implemented in the D-Finder tool [11] for checking deadlock-freedom of systems described in BIP. Experimental results on classical benchmarks (as illustrated in Figure 4) show that D-Finder can be exponentially faster than well-established verification tools. Nonetheless, D-Finder has been also successful for the verification of complex software applications, as illustrated later in section 3.

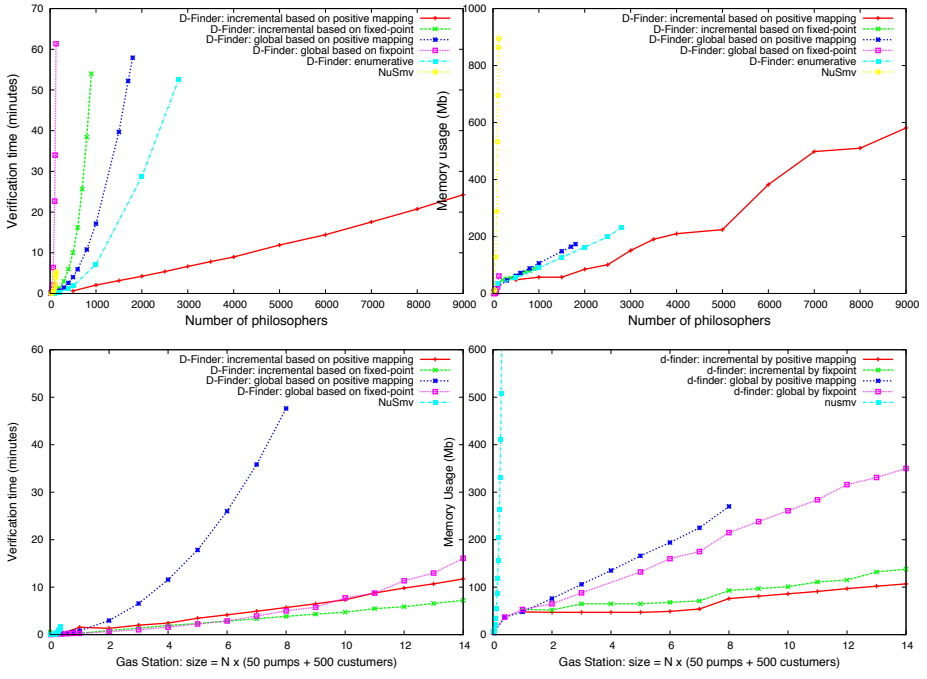

Fig. 4. D-Finder results: time (left) and memory usage (right) as a function of complexity for i) monolithic verification with NuSMV, ii) compositional verification, iii) incremental verification on two benchmarks, dining philosophers (up) and gas station (down)

2.4 Integrating Architectural Constraints in BIP

We developed in [12] a rigorous method for generating a model which faithfully represents the behavior of a mixed hardware/software system from a model of its application software and a model of its underlying hardware architecture. The method takes as input a model of the application software in BIP, a model of the hardware architecture (in XML) and a mapping associating communication operations of the application software with execution/communication paths in the architecture. It builds a model of the corresponding mixed hardware/software system in BIP. This system model can be simulated and analyzed for the verification of functional and extra-functional properties.

The method consists in progressively enriching the application software model by doing:

1. Integration of hardware components used in the system model and,
2. Application of a sequence of source-to-source transformations to synthesize *hardware dependent software routines* implementing communication by using the hardware components.

   The transformations are proved correct-by-construction, that is, they preserve functional properties of the application software.

The system model is parameterized and allows flexible integration of specific target architecture features, such as arbitration policy, throughput, latency for buses and scheduling policy, execution speed, etc. We have defined a library of BIP atomic components that characterize multi-processor tiled architectures, including models for hardware components (e.g., processor, memory) and for hardware-dependent software components (e.g., FIFO channel read/write, bus controllers, schedulers).

The method has been implemented and integrated in the BIP toolset. We used the DOL framework [7] as a frontend to describe the application software, hardware architectures and mapping specifications. The backend of the tool produces the system model in BIP, which can be analyzed by the BIP tool chain for:

- Code generation for simulation/validation on a Linux PC
- Functional correctness using the D-Finder tool, checking for deadlocks
- Performance analysis (e.g. delay computation), based on simulation and statistical model checking

We generated different system models of an MJPEG decoder running on a simplified MPARM platform. The decoder is described in DOL [7], and consists of five processes communicating asynchronously through FIFO channels. The process description consists of about 2500 lines of C. The description is automatically translated into the application software model, which is about 10000 lines of BIP. This model is purely functional and can be analyzed with D-Finder to assess its correctness. It has been mapped on a MPARM platform consisting of three processors, their local memories, and a global shared memory, with all being connected via a global bus. Different mappings were considered, leading to different system models. These models have been used for performance analysis. Using simulation, we measured computation and communication times for relevant parts of the application software. As future work, we intend to use these results in order to build (simpler) statistical abstractions of the system models on which properties can be validated using statistical model-checking.

2.5 Generating Distributed Implementations

To generate distributed implementations from BIP models it is necessary to transform these models into S/R-BIP models. These are a subclass of models where multi-party interaction is replaced by protocols using S/R (Send/Receive) primitives. Then, 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.

We developed in [13] a general method for generating distributed implementations from BIP models. The method uses the following sequence of correct-by-construction transformations, that preserve *observational equivalence*:

1. We transform a BIP system model 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 a transformation of atomic components in the original model.
   (b) The *interaction protocol* layer consists of a set of components, each hosting a user-defined subset of interactions from the original BIP model. This layer 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* [14] using, so far, one distributed algorithm amongst (i) fully centralized, (ii) token-ring, and (iii) dining philosophers [15,14].
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.
   The composition operation has been implemented in the BIP2BIP tool, by using three elementary source-to-source transformations:
   (a) *Component flattening*, which replaces the hierarchy on components by a set of hierarchically structured connectors applied on atomic components;
   (b) *Connector flattening*, which computes for each hierarchically structured connector an equivalent flat connector;
   (c) *Component composition*, which composes atomic components to get an atomic component.

We conducted a set of experiments [16,13] to analyze the behavior and performance of the generated code using different scenarios (i.e., different partitioning of interactions, choice of committee coordination algorithm, mapping). Our experiments clearly show that particular configurations are suitable for different topology, size of the distributed system, communication load, and of course, the structure of the initial BIP model.

Table 1 taken from [16] summarizes experimental results obtained for different distributed implementations of a bitonic sorting algorithm [17]. We run experiments for three platform configurations denoted $m \times c$, for m interconnected machines with c cores each. The table provide the total sorting time for arrays of size $k \times 10^4$ elements, and $k = 20, 40, 80, 160$. As can be seen, execution times for handwritten MPI are slightly better than for plain C++ with TCP/IP communication. For example, the execution time for sorting an array of size 80×10^4, for the configuration 2×2 is: 240 seconds for MPI, and 390 seconds for plain C++.

In the case of S/R-BIP models auto-generated as described earlier, it is frequent that some of the atomic components and engines cannot run in parallel.

Table 1. Total sorting time for different implementations of a bitonic sorting algorithm (handwritten or generated, with or without optimisation) deployed on different execution platforms ($m \times c$ denotes m interconnected machines with c cores each) on unsorted arrays of size $k \times 10^4$ elements

	*MPI (*handwritten*)*			*Plain C++ with TCP/IP*				*MPI (*generated*)*	
optimised	-			no	no	yes	no	no	yes
$m \times c$	1×1	2×2	4×1	1×1	2×2	2×2	4×1	2×2	2×2
$k = 20$	80	14	14	96	23	24	24	63	24
$k = 40$	327	59	60	375	96	96	100	271	96
$k = 80$	1368	240	240	1504	390	391	397	964	394
$k = 160$	5605	1007	958	6024	1539	1548	1583	4158	1554

Therefore, they can be composed without losing any parallelism. For the bitonic sorting example, the original S/R-BIP model has 7 atomic components (4 atomic components and 3 engines), and can be transformed into a *merged* S/R-BIP model containing only 4 components, while preserving all the parallelism.

The performance gain obtained by using static composition on 2 dual-core machines (2×2 setting) is shown in Table 1. Observe that the performance of the C++ implementation is approximately identical in both cases, with or without optimisation. This is because TCP/IP communication is interrupt-driven. Thus, if a component is waiting for a message, it does not consume CPU time. On the other hand, MPI uses active waiting, which results in CPU time wasting when components are waiting. Since we have four cores for more processes (seven), the MPI code generated from the original S/R-BIP model is much slower than the plain C++ code. Nevertheless, reducing the number of components to one per core by composition allows the MPI code to reach the same speed as the C++ implementation.

3 Case Studies

BIP has been applied to several non trivial case studies. These include the componentization of a MPEG encoder [18] and of the control software of the DALA robot of LAAS [19]. Another case study is modeling TinyOS-based wireless sensor networks [20]. Moreover, BIP has been also used for modeling, verification and performance evaluation of a self-stabilizing distributed reset algorithm [21].

3.1 MJPEG Decoder

The MJPEG decoder application software reads a sequence of MJPEG frames and displays the decompressed video frames. The process network of the application software is illustrated in Figure 5. It contains five processes *SplitStream* (*SS*), *SplitFrame* (*SF*), *IqzigzagIDCT* (*IDCT*), *MergeFrame* (*MF*) and *MergeStream* (*MS*), and nine communication FIFO channels *C1*, ..., *C9*. The total lines of

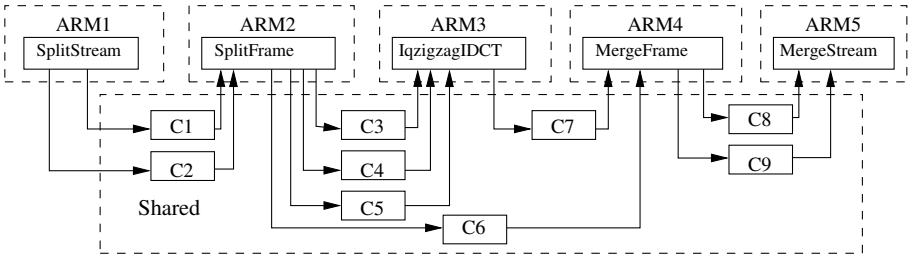

Fig. 5. MJPEG Decoder application software and a mapping

Table 2. Mapping Description of the processes and the FIFOs

	$ARM1$	$ARM2$	$ARM3$	$ARM4$	$ARM5$
1	all				
2	SS, SF, IQ	MF, MS			
3	SS, SF	IQ, MF, MS			
4	SS, SF	IQ	MF, MS		
5	SS, MS	SF	IQ	MF	
6	SS	SF	IQ	MF	MS
7	SS, SF	IQ	MF, MS		
8	SS	SF	IQ	MF	MS

	$Shared$	$LM1$	$LM2$	$LM3$	$LM4$
1		all			
2	C6, C7	C1, C2, C3, C4, C5	C8, C9		
3	C3, C4, C5, C6	C1, C2	C7, C8, C9		
4	C3, C4, C5, C6, C7	C1, C2		C8, C9	
5	all				
6	all				
7	C6, C7	C1, C2, C3, C4, C5		C8, C9	
8		C1, C2	C3, C4, C5, C6	C7	C8, C9

C code describing the behavior of the application software processes is approximately 1600.

We analyzed the effect of eight different mappings on the total computation and communication delay for decoding a frame. The process and the FIFO mappings are illustrated on Table 2.

For these mappings a system model contains around 50 BIP atomic components and 220 BIP interactions, and consists of approximately 6K lines of BIP code, generating around 19.5K lines of C code for simulation.

The total computation and communication delays for decoding a frame for different mappings are shown in Figure 6. Mapping (1) produces the worst computation delay as all processes are mapped to a single processor. Mapping (2) uses two processors, but still the performance does not improve much. Mapping (3) drastically improves performance as the computation load is balanced. The other mappings cannot further enhance performance as the load cannot be further balanced, even if more processors are used. The communication overhead is

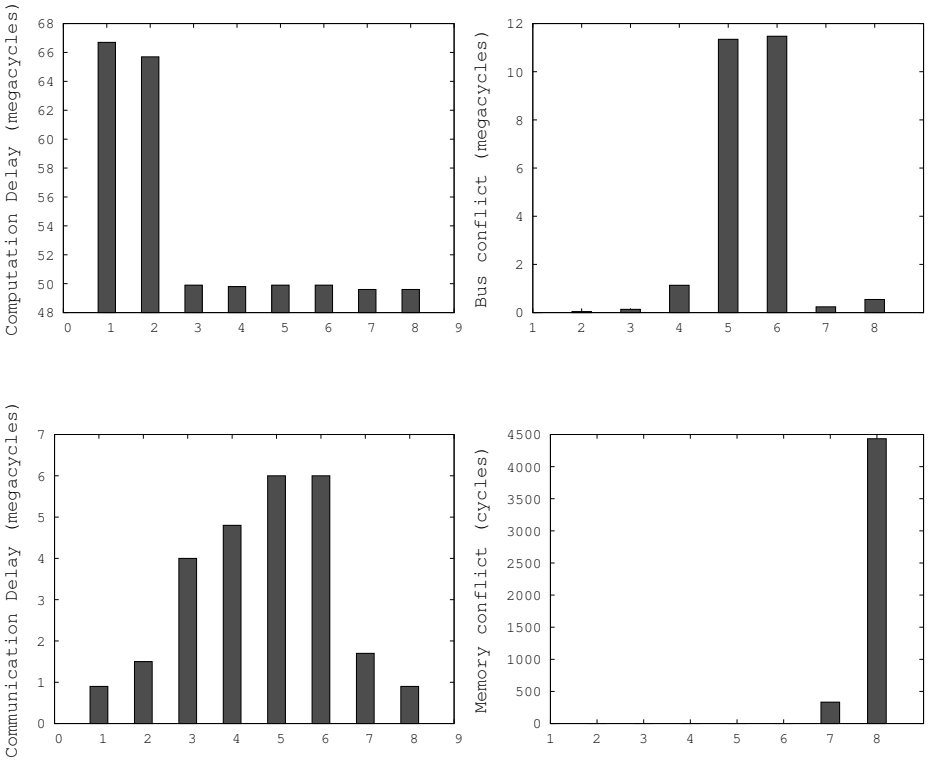

Fig. 6. Mjpeg Performance Analysis Results

reduced if we map more FIFOs to the local memories of the processors. The bus and memory access conflicts are shown in Figure 6. As more FIFOs are mapped to the local memory, the shared bus contention is reduced. However, this might increase the local memory contention, as shown for (8).

3.2 Heterogeneous Communication System

This case study deals with a distributed heterogeneous communication system (HCS) providing an all electronic communication infrastructure, typically for cabin communication in airplanes or for building automation. HCS contains various devices such as sensors (video camera, smoke detectors, temperature, pressure, etc.) and actuators (loudspeakers, light switches, temperature control, signs, etc.) connected through a wired Ethernet network to a central server. The server runs a set of services to monitor the sensors and to control the actuators. The devices are connected to the server using network access controllers.

The architecture and functionality delivered by HCS are highly heterogeneous. The system includes different hardware components, which run different

protocols and software services ensuring functions with different characteristics and degree of criticality e.g, audio streaming, clock synchronization, sensor monitoring, video surveillance, etc. Moreover, HCS has to guarantee stringent requirements, such as reliable data transmission, fault tolerance, timing and synchronization constraints. For example, the latency for delivering alarm signals from sensors, or for playing audio announcements should be smaller than certain predefined thresholds. Or, the accuracy of clock synchronization between different devices, should be guaranteed under the given physical implementation of the system.

Complete details of this case study can be found in [22]. We have developed a structural model of HCS using BIP. At top level, the structure of the model follows the natural decomposition into physical elements e.g., server, network access controllers and devices are the top-level components. Moreover, these components are connected and interact according to the wired network connections defined in the original system. Then, one level down, every (physical) component has a functional decomposition. Inner subcomponents provide features for network operation (e.g., packet delivery, filtering, routing, scheduling, ...), protocols (e.g., clock synchronization) or services (e.g., audio/video streaming, event handling, etc.)

The overall complexity of this case study is extremely high. A model for a relevant functional subsystem required approximately 300 atomic components and 1900 connectors in BIP. Almost all atomic components have timed behavior. They totalize approximately 250 clocks variables to express all timing constraints. Moreover, the use of large domain data (e.g., packet numbers) and complex data structures (e.g., FIFO queues of packets) made the state space of the model extremely huge. One single state needs approximately 400 bytes to be represented. Furthermore, the state space has a heterogeneous structure which prevents its compact representation using symbolic techniques based on BDDs.

We have been interested to verify the clock synchronization protocol i.e., the application used to synchronize the clocks of all devices within the system. The challenge is to guarantee that the protocol maintains the difference between a master clock (running on the server) and all the slave clocks (running on devices) under some bound. A first major difficulty is network communication which makes all applications interfering and therefore requires exploration of the whole model. A second difficulty comes from the time granularity i.e., one microsecond, needed to perform faithful observations. These two factors significantly restrict brute-force simulation approaches: 1 second system lifetime needs approximately 10 minutes simulation time with microsecond precision on the BIP model.

To overcome these difficulties, we proposed in [22] a new verification technique which combines random simulation and statistical model checking. We have been able to derive exact bounds on clock synchronization for all devices in the system. We also computed probabilities of clock synchronization for smaller values of the bound. Being able to provide such information is of clear importance, especially when the exact bounds are too high with respect to user's requirements. In particular, we have shown that the bounds strongly depend on the

position of the device in the network. We also estimated the average and worst proportion of failures per simulation for smaller bounds i.e., how often the clock synchronization exceeds the given bound on some arbitrary run.

4 Discussion and Future Work

We have shown that the BIP component framework, and the associated design flow and supporting tools allow rigorous and effective system design. 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 coordination mechanisms used for the collaboration 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 will hold for all the models obtained by transformation. The complexity of the transformations is linear with the size of the transformed models. So correctness is ensured at minimal cost and by construction thus overcoming obstacles of design flows involving different and not semantically related languages and models.

The use of a single modeling framework allows to maintain the overall coherency of the design flow by comparing different architectural solutions and their properties. This is a significant advantage of our approach. Semantically related models are used for verification, simulation and performance evaluation. Designers use many different languages e.g. programming languages, UML, SystemC, SES/Workbench. Code generation and deployment is often independent from validation and evaluation.

Clearly, using a single modeling framework does not suffice. An advantage of BIP over other existing frameworks is its expressiveness. It uses a few powerful primitives to express coordination between components. Architecture is a first class concept and can be characterized as the combination of interactions and priorities. It can model in a natural and direct manner both timed and untimed behavior, synchronous and asynchronous. Using less expressive frameworks e.g. based on a single composition operator, would lead to intractable models. For instance, BIP directly encompasses multiparty interaction between components. This type of coordination would require the development of complex coordination mechanisms for frameworks supporting only point-to point interaction. This would lead to models with complicated coordination structure and would make the whole design flow intractable. In particular for such models establishing a clean refinement relation between the different models would be compromised.

Empirical design flows are limited to simple execution models and execution platforms involving a few processing elements. We believe that rigorous and automated design flows are crucial for system development especially when the target architecture is distributed and/or heterogeneous.

References

1. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Systems in BIP. In: Proceedings of Software Engineering and Formal Methods SEFM 2006, pp. 3 –12. IEEE Computer Society Press (2006)
2. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic Publishers (1993)
3. Burns, A., Welling, A.: Real-Time Systems and Programming Languages, 3rd edn. Addison-Wesley (2001)
4. Henzinger, T., Sifakis, J.: The Embedded Systems Design Challenge. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006)
5. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
6. Bliudze, S., Sifakis, J.: A Notion of Glue Expressiveness for Component-Based Systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008)
7. Thiele, L., Bacivarov, I., Haid, W., Huang, K.: Mapping Applications to Tiled Multiprocessor Embedded Systems. In: Proceedings of Application of Concurrency to System Design ACSD 2007, pp. 29–40. IEEE Computer Society (2007)
8. Fleury, S., Herrb, M., Chatila, R.: GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. In: IROS 1997, pp. 842–848 (1997)
9. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.-H.: Compositional Verification for Component-based Systems and Application. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 64–79. Springer, Heidelberg (2008)
10. Bensalem, S., Bozga, M., Legay, A., Nguyen, T.-H., Sifakis, J., Yan, R.: Incremental Component-based Construction and Verification using Invariants. In: FMCAD 2010, pp. 257–266. IEEE (2010)
11. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: D-Finder: A Tool for Compositional Deadlock Detection and Verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009)
12. Bourgos, P., Basu, A., Bozga, M., Bensalem, S., Sifakis, J., Huang, K.: Rigorous system level modeling and analysis of mixed hw/sw systems. In: Proceedings of MEMOCODE, pp. 11–20. IEEE/ACM (2011)
13. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From high-level component-based models to distributed implementations. In: Proceedings of Embedded Software EMSOFT 2010. ACM (2010)
14. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley Longman Publishing Co. Inc., Boston (1988)
15. Bagrodia, R.: Process synchronization: Design and performance evaluation of distributed algorithms. IEEE Transactions on Software Engineering 15(9), 1053–1065 (1989)
16. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: Automated Conflict-free Distributed Implementation of Component-Based Models. In: Proceedings of Industrial Embedded Systems SIES 2010. IEEE (2010)
17. Batcher, K.E.: Sorting Networks and Their Applications. In: Proceedings of AFIPS 1968 (Spring), pp. 307–314 (1968)

18. Combaz, J., Fernandez, J.C., Sifakis, J., Strus, L.: Symbolic quality control for multimedia applications. Real-Time Systems 40(1), 1–43 (2008)
19. Basu, A., Gallien, M., Lesire, C., Nguyen, T.-H., Bensalem, S., Ingrand, F., Sifakis, J.: Incremental Component-Based Construction and Verification of a Robotic System. In: ECAI 2008. FAIA, vol. 178, pp. 631–635. IOS Press (2008)
20. Basu, A., Mounier, L., Poulhiès, M., Pulou, J., Sifakis, J.: Using BIP for Modeling and Verification of Networked Systems – A Case Study on TinyOS-based Networks. In: Proceedings of Network Computing and Applications NCA 2007, pp. 257–260. IEEE (2007)
21. Basu, A., Bonakdarpour, B., Bozga, M., Sifakis, J.: Brief Announcement: Incremental Component-Based Modeling, Verification, and Performance Evaluation of Distributed Reset. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 174–175. Springer, Heidelberg (2009)
22. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical Abstraction and Model-Checking of Large Heterogeneous Systems. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 32–46. Springer, Heidelberg (2010)