Component Assemblies in the Context of Manycore

Ananda Basu, Saddek Bensalem, Marius Bozga, Paraskevas Bourgos, Mayur Maheshwari, Joseph Sifakis

▶ **To cite this version:**

Ananda Basu, Saddek Bensalem, Marius Bozga, Paraskevas Bourgos, Mayur Maheshwari, et al.. Component Assemblies in the Context of Manycore. Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011, Oct 2011, Torino, Italy. pp.314-333, 10.1007/978-3-642-35887-6_17 . hal-00878716

HAL Id: hal-00878716
https://hal.science/hal-00878716

Submitted on 30 Oct 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Component Assemblies in the Context of Manycore *

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

[1] UJF-Grenoble 1 / CNRS, VERIMAG UMR 5104, Grenoble, F-38041, France
name.surname@imag.fr
[2] RISD Laboratory, EPFL joseph.sifakis@epfl.ch

Abstract. We present a component-based software design flow for building parallel applications running on top of manycore platforms. The flow is based on the BIP - Behaviour, Interaction, Priority - component framework and its associated toolbox. It provides full support for modeling of application software, validation of its functional correctness, modeling and performance analysis on system-level models, code generation and deployment on target manycore platforms. The paper details some of the steps of the design flow. The design flow is illustrated through the modeling and deployment of two applications, the Cholesky factorization and the MJPEG decoding on MPARM, an ARM-based manycore platform. We emphasize the merits of the design flow, notably fast performance analysis as well as code generation and efficient deployment on manycore platforms.

1 Introduction

The emergence of manycore platforms is nowadays challenging the design practices for embedded software. Manycore platforms built on increasingly complex 2D or 3D hardware architectures which, besides a high number of computational cores, usually include complex memory/cache hierarchies, synchronization patterns and/or communication buses and networks. Commonly, all hardware resources are either partially or fully exposed to software developers. By doing so, one expects optimized exploitation of resources while meeting requirements for both software performance (e.g., real-time requirements) and efficient platform management (e.g., thermal and power efficiency).

Concurrency is paramount for boosting software performance on manycore platforms. Nonetheless, correct and fast development of highly parallel, fine-grain

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 248776 (PRO3D) and from ARTEMIS JU grant agreement ARTEMIS-2009-1-100230 (SMECY).

concurrent software is known to be notoriously hard even for expert developers. In general, the inherent complexity of concurrent (handwritten) software is hardly manageable by current verification and validation methods and tools. Moreover, software adaptation and deployment to selected manycore platform targets usually require significant manual transformation, with no strong guarantees about their correctness.

The PRO3D project [32] proposes a holistic approach for the development of embedded applications on top of manycore 3D platforms. PRO3D activities range from programming to architecture exploration and fabrication technologies. The major challenges are the thermal management of 3D platforms and the rigorous, tool-supported design flow of parallel application software.

We propose and implement a design flow for applications based on the BIP component framework [6]. This flow sticks to the general principles of *rigorous design* introduced in [8]. It has several key features, namely:

- it is *model-based*, that is, both application software and mixed hardware/ software system descriptions are modeled by using a single, semantic framework. As stated in [8], this allows maintaining the coherency along with the flow by proving that various transformations used to move from one description to another preserve essential properties.
- it is *component-based*, that is, it provides primitives for building composite components as the composition of simpler components. Using components reduces development time by favoring component reuse and provides support for incremental analysis and design, as introduced in [10, 11, 13]
- it is *tool-supported*, that is, all steps in the design flow are realized automatically by tools. This ensures significant productivity gains, in particular due to elimination of potential errors that can occur in manual transformations.

To the best of our knowledge, the BIP design flow is unique as it uses a single semantic framework to support application modeling, validation of functional correctness, performance analysis on system models and code generation for manycore platforms. Building faithful system models is mandatory for validation and performance analysis of concurrent software running on manycore platforms. Existing system modeling formalisms either seek generality at the detriment of rigorousness, such as SysML [30] and AADL [20] or have a limited scope as they are based on specific models of computation such as Ptolemy [18]. Simulation based methods use ad-hoc executable system models such as [22] or tools based on SystemC [28]. The latter provide cycle-accurate results, but in general, they have long simulation time as a major drawback. As such, these tools are not adequate for thorough exploration of hardware platform dynamics, neither for estimating effects on real-life software execution. Alternatives include trace-based co-simulation methods as used in Spade [26], Sesame [19] or Daedalus [29]. Additionally, there exist much faster techniques that work on abstract system models e.g., Real Time Calculus [36] and SymTA/S [4]. They use formal analytical models representing a system as a network of nodes exchanging

streams. They often oversimplify the dynamics of the execution characterized by execution times. Moreover, they allow only estimation of pessimistic worst-case quantities (delays, buffer sizes, etc) and require adequate abstract models of the application software. Building such models entails an additional significant modeling effort. Similar difficulties arise in performance analysis techniques based on Timed-Automata [3, 33]. These can be used for modeling and solving scheduling problems. An approach combining simulation and analytic models is presented in [23], where simulation results can be propagated to analytic models and vice versa through adequate interfaces.

The paper is organized as follows. Section 2 provides a brief overview of the BIP component framework and toolbox. Section 3 introduces the BIP design flow and details several of its steps. An illustration of the design flow is provided in section 4. We provide results about performance analysis and implementation of two non-trivial concurrent applications on manycore. Finally, section 5 concludes and provides future work directions.

2 The BIP Framework

The BIP – Behaviour / Interaction / Priority – framework [6] is aiming at design and analysis of complex, heterogeneous embedded applications. BIP is a highly expressive, component-based framework with rigorous semantical basis. It allows the construction of complex, hierarchically structured models from atomic components characterized by their behavior and their interfaces. Such components are transition systems enriched with data. Transitions are used to move from a source to a destination location. Each time a transition is taken, component data (variables) may be assigned new values, computed by user-defined functions (in C). Atomic components are composed by layered application of interactions and priorities. Interactions express synchronization constraints and define the transfer of data between the interacting components. 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.

Atomic Components. We define *atomic components* as transition systems extended with a set of ports and a set of variables. Formally, an *atomic component* B is a labelled transition system represented by a tuple (Q, X, P, T) where Q is a set of *control locations*, X is a set of *variables*, P is a set of *communication ports* and T is a set of *transitions*. Each transition τ is of the form (q, p, g, f, q') where $q, q' \in Q$ are control locations, $p \in P$ is a port, g is the *guard* and f is the *update function* of τ. g is a predicate defined over variables in X and f is a function (or a sequential procedure) that computes new values for X according to the current ones.

Interactions. In order to compose a set of n atomic components $\{B_i = (Q_i, X_i, P_i, T_i)\}_{i=1}^{n}$, we assume that their respective sets of ports and variables are pairwise disjoint; i.e., for all $i \neq j$ we require that $P_i \cap P_j = \emptyset$ and $X_i \cap X_j = \emptyset$.

We define the global set $P \stackrel{def}{=} \bigcup_{i=1}^{n} P_i$ of ports. An *interaction* a is a triple (P_a, G_a, F_a), where $P_a \subseteq P$ is a set of ports, G_a is a guard, and F_a is a data transfer function. By definition P_a contains at most one port from each component. We denote $P_a = \{p_i\}_{i \in I}$ with $I \subseteq \{1..n\}$ and $p_i \in P_i$. We assume that G_a and F_a are defined on the variables of participating components, (i.e. $\bigcup_{i \in I} X_i$). We denote by F_a^i the restriction of F_a on variables X_i.

Priorities. Given a set γ of interactions, we define a priority as a strict partial order $\pi \subseteq \gamma \times \gamma$. We write $a \pi b$ for $(a, b) \in \pi$, to express the fact that interaction a has lower priority than interaction b.

Composite Components. A *composite component* $\pi\gamma(B_1, \ldots, B_n)$ is defined by a set of atomic components $B_1, \cdots, B_n$, composed by a set of interactions γ and a priority $\pi \subseteq \gamma \times \gamma$. If π is the empty relation, then we may omit π and simply write $\gamma(B_1, \cdots, B_n)$.

A global state of $\pi\gamma(B_1, \cdots, B_n)$ where $B_i = (Q_i, X_i, P_i, T_i)$ is defined by a couple (q, v), where $q = (q_1, \cdots, q_n)$ is a tuple of control locations such that $q_i \in Q_i$ and $v = (v_1, \cdots, v_n)$ is a tuple of valuations of variables such that $v_i \in Val(X_i) = \{\sigma : X_i \to \mathcal{D}\}$, for all $i = 1, \cdots n$ and for $\mathcal{D}$ being some universal data domain. The behavior of a composite component without priority $\gamma(B_1, \cdots, B_n)$ is a labeled transition system $(S, \gamma, \to_\gamma)$, where $S = \bigotimes_{i=1}^{n} Q_i \times \bigotimes_{i=1}^{n} Val(X_i)$ and $\to_\gamma$ is the least set of transitions satisfying the rule:

$$\frac{\begin{array}{c} a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma \quad v_a = \{v_i\}_{i \in I} \quad G_a(v_a) \\ \forall i \in I.\ (q_i, g_i, p_i, f_i, q_i') \in T_i \quad g_i(v_i) \quad v_i' = f_i(F_a^i(v_a)) \\ \forall i \notin I.\ (q_i, v_i) = (q_i', v_i') \end{array}}{((q_1, \ldots, q_n), (v_1, \ldots, v_n)) \stackrel{a}{\to}_\gamma ((q_1', \ldots, q_n'), (v_1', \ldots, v_n'))} \text{ [INTERACTION]}$$

Intuitively, the inference rule INTERACTION specifies that a composite component $B = \gamma(B_1, \ldots, B_n)$ can execute an interaction $a \in \gamma$, iff (1) for each port $p_i \in P_a$, the corresponding atomic component B_i allows a transition from the current location labelled by p_i (i.e. the corresponding guard g_i evaluates to true), and (2) the guard G_a of the interaction evaluates to true. If the two above conditions hold for an interaction a at state (q, v), a is *enabled* at that state. Execution of a modifies participating components' variables by first applying the data transfer function F_a on variables of all interacting components and then the update function f_i for each interacting component. The (local) states of components that do not participate in the interaction stay unchanged.

We define the behavior of the composite component $B = \pi\gamma(B_1, \ldots, B_n)$ as the labeled transition system $(S, \gamma, \to_{\pi\gamma})$ where $\to_{\pi\gamma}$ is the least set of transitions satisfying the rule:

$$\frac{(q, v) \stackrel{a}{\to}_\gamma (q', v') \qquad \forall a' \in \gamma.\ a \pi a' \Longrightarrow (q, v) \stackrel{a'}{\not\to}_\gamma}{(q, v) \stackrel{a}{\to}_{\pi\gamma} (q', v')} \text{ [PRIORITY]}$$

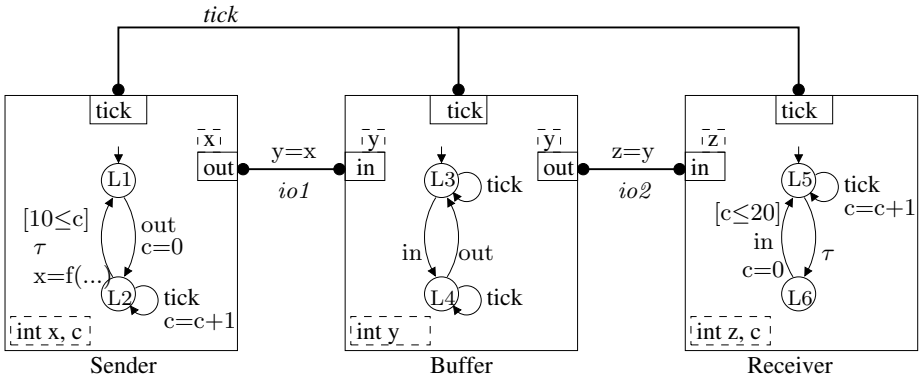

Fig. 1. BIP Example: Sender/Buffer/Receiver System

The inference rule PRIORITY filters out interactions which are not maximal with respect to the priority order. An interaction is executed only if no other one with higher priority is enabled.

Example 1. Figure 1 shows a graphical representation of an example model in BIP. It consists of atomic components *Sender*, *Buffer* and *Receiver*. The behavior of *Sender* is described as a transition system with control locations *L1* and *L2*. It communicates through ports *tick* and *out*. Port *out* exports the variable x. Components *Sender*, *Buffer* and *Receiver* are composed by two binary connectors *io1, io2* and a ternary connector *tick*. *tick* represents a rendezvous synchronization between the *tick* ports of the respective components. *io1* represents an interaction with data transfer from the port *out* of *Sender* to the port *in* of *Buffer*. As a result of the data transfer associated with *io1*, the value of variable x of *Sender* is assigned to the variables y of the *Buffer*.

BIP is supported by a rich toolset[1] which includes tools for checking correctness, for source-to-source transformations and for code generation. Correctness can be either formally proven using invariants and abstractions, or tested by using simulation. For the latter case, simulation is driven by a specific middleware, the BIP engine, which allows to explore and inspect traces corresponding to BIP models. Source-to-source transformations allow to realize static optimizations as well as specific transformations towards implementation i.e., distribution. Finally, code generation targets different platforms and operating systems support (e.g., distributed, multi-threaded, real-time, for single/multi-core platforms, etc.).

3 BIP Design Flow for Manycore

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. The design flow involves several distinct steps, as illustrated in figure 2 and explained below:

1. The *translation* of the application software into a BIP model. This allows its representation in a rigorous semantic framework. Translations for several programming models (including synchronous, data-flow and event-driven) and domain specific languages into BIP are defined and implemented.
2. *Correctness checking of functional properties* of the application software. Functional verification needs to be done only on high-level models since safety properties and deadlock-freedom are preserved by different transformations applied along the design flow. To avoid inherent complexity limitations, the verification method relies on compositionality and incremental techniques.
3. The *construction of an abstract system model*. This model is automatically generated from 1) the BIP model representing the application software; 2) a BIP model of the target execution platform; 3) a mapping of the atomic components of the application software model into processing elements of the platform. The abstract system model takes into account hardware constraints such as various latencies, mutual exclusion induced from sharing physical resources (like buses, memories and processors) as well as scheduling policies seeking optimal use of these resources.
4. The *construction of a distributed system model*. This model is automatically generated from the abstract system model by expressing high-level coordination mechanisms e.g., interactions and priorities, in terms of primitives of the execution platform. This transformation involves the replacement of atomic multiparty interactions and/or dynamic priorities by protocols using asynchronous message passing (send/receive primitives) and arbiters ensuring semantics preservation. These transformations are proved correct-by-construction [14].
5. The *generation of platform dependent code*, including both functional and glue code for deploying and running the application on the target manycore. In particular, components mapped on the same core can be statically composed thus avoiding extra overhead for (local) coordination at runtime.
6. The *calibration* step, which consists in estimating execution times of actions of the distributed system model. These are obtained through execution and profiling of code fragments compiled on the target platform. They are used to obtain an instrumented system model which takes into account dynamic behavior of the execution platform.
7. The *performance analysis* step involving simulation-based methods combined with statistical model checking on the instrumented system model.

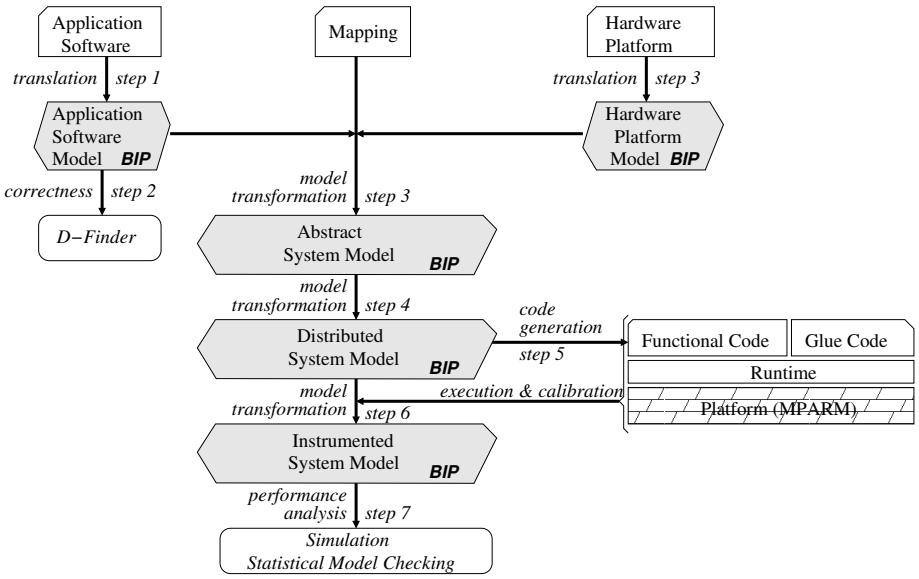

Fig. 2. BIP Design Flow for Manycore

Some of the steps of the design flow are detailed hereafter. We focus on the translation of the application software in BIP (step 1), functional correctness checking by using D-Finder (step 2), platform dependent code generation (step 5), calibration and performance analysis (steps 6 and 7). The construction of the abstract system model (step 3) is presented in [15]. A complete presentation of transformations for building distributed system models (step 4), ready for implementation on distributed platforms, can be found in [14].

3.1 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. We have implemented BIP model generators for several programming models and languages such as Lustre, Simulink and NesC/TinyOS. In this paper, we focus on applications described in the DOL (Distributed Operation Layer) framework [35].

An application software in DOL is a Kahn process network that consists of three basic entities: *Processes*, *FIFO* channels, and *Connections*. The network structure is described in XML. Processes are defined as sequential C programs with a particular structure. For a process P, its state is defined as an arbitrary C data structure named *P_state* and its behavior as the program *P_init(); while (true) P_fire();* where *P_init(), P_fire()* are arbitrary functions operating on the

process state. Communication is realized by two primitives, namely *write* and *read* for respectively sending and receiving data to FIFO channels. Moreover, the *P_fire()* method invokes a *detach* primitive in order to terminate the execution of the process.

The construction of the application software model in BIP is done through translation of the above entities in BIP. The construction is structure-preserving: every process and every FIFO are independently translated into atomic components in BIP and then connected according to the connections in the process network [15]. The translation of process behavior requires extraction of an explicit control flow graph from the C code and its representation as an atomic component in BIP. A FIFO channel is translated into a predefined BIP atomic component.

Example 2. The C description of a DOL process is presented in Figure 3. This process belongs to a process network used for the Cholesky factorization experiment, presented later in section 4. The BIP atomic component generated from the DOL process is shown in figure 4. It has ports *IN_SPLIT*, *IN_2_1*, *OUT_JOIN*, control locations *L1* ... *L6* and variables *index*, *len*, *A*, *L* and *X*. Transitions are labeled by ports *IN_SPLIT*, *IN_2_1*, *OUT_JOIN* and β (internal).

3.2 Checking Application Correctness

The BIP design flow includes a verification step for checking essential functional properties. Application software models in BIP are verified by using the D-Finder tool[10, 11]. D-Finder implements compositional methods for generation of invariants and verification of safety properties, including deadlock-freedom.

In general, compositional verification techniques [5, 2, 17, 16, 21, 27, 31, 34] are used to cope with state explosion in concurrent systems. The idea is to apply divide-and-conquer approaches to infer global properties of complex systems from properties of their components. Separate verification of components limits its state explosion. Nonetheless, designing compositional verification techniques is difficult since components mutually interact in a system and their behavior and properties are inter-related. As explained in [24], compositional rules are in general of the form:

$$\frac{B_1 < \Phi_1 >,\ B_2 < \Phi_2 >,\ C(\Phi_1, \Phi_2, \Phi)}{B_1 \| B_2 < \Phi >} \quad (1)$$

That is, if two components with behaviors B_1, B_2 meet individually properties Φ_1, Φ_2 respectively, and $C(\Phi_1, \Phi_2, \Phi)$ is some condition taking into account the semantics of parallel composition operation and relating the individual properties with the global property, then the system $B_1 \| B_2$ resulting from the composition of B_1 and B_2 will satisfy a global property Φ.

```
void p_2_2_init(DOLProcess *p) {
  p->local->index = 0;
  p->local->len = LENGTH; }
int p_2_2_fire(DOLProcess *p) {
  if (p->local->index < p->local->len) {
    // read input block A22 from splitter
    read((void*)IN_SPLT, p->local->A,
      (K)*(K)*sizeof(double), p);
    // read result block L21 from P21
    read((void*)IN_2_1, p->local->X,
      (K)*(K)*sizeof(double), p);
    // compute A22 = A22 − L21 × L21^t
    SubtractTProduct(p->local->A,
        p->local->X, p->local->X);
    // compute L22 = seq-cholesky(A22)
    Cholesky(p->local->L, p->local->A);
    // send the result L22 to the joiner
    write((void*)OUT_JOIN, p->local->L,
      (K)*(K)*sizeof(double), p);
    p->local->index++; }
  else {
    // termination
    detach(p);
    return -1; }
  return 0; }
```

Fig. 3. DOL Process Description in C

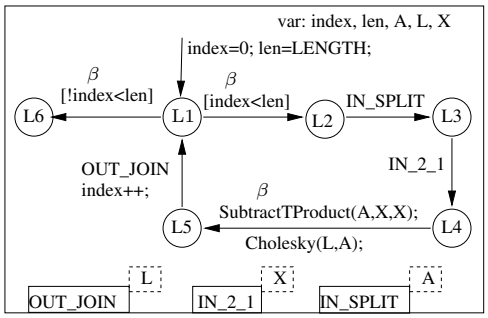

Fig. 4. Translation as BIP Atomic Component

D-Finder[10, 11] provides a novel approach for compositional verification of invariants in BIP based on the following rule:

$$\frac{\{B_i < \Phi_i >\}_{i=1}^{n},\ \Psi \in II(\|_\gamma, \{B_i\}_{i=1}^{n}, \{\Phi_i\}_{i=1}^{n}),\ (\bigwedge_i \Phi_i) \wedge \Psi \Rightarrow \Phi}{\gamma(B_1, \cdots B_n) < \Phi >} \qquad (2)$$

The rule (2) allows to prove a global invariant Φ for a composite component $\gamma(B_1, \ldots, B_n)$, obtained by composing a set of atomic components $B_1, ..., B_n$ by using a set of interactions γ. The premises ensure respectively that, Φ_i is a local invariant of component B_i for every $i = 1, \cdots n$ and Ψ is an interaction invariant of $\gamma(B_1, \ldots, B_n)$ computed automatically from interactions γ, components B_i and local invariants Φ_i. D-Finder provides methods for computing both component invariants and interaction invariants as follows:

– *Invariants for atomic components* are generated by static forward analysis of their behavior. D-Finder uses different strategies which allow to derive local assertions, that is, predicates attached to control locations and which are satisfied whenever the computation reaches the corresponding control

location. These assertions are obtained through syntactic analysis of the predicates occuring in guards and actions [12].
- *Interaction invariants* express global synchronization constraints between atomic components. Their computation consists of the following steps. First, for given component invariants Φ_i of the atomic components B_i, we compute a finite-state abstractions $B_i^{\alpha_i}$ of B_i where α_i is the abstraction induced by the elementary predicates occurring in Φ_i. This step is necessary only for components B_i which are infinite state. Second, the composition $\gamma(B_1^{\alpha_1}, \cdots, B_n^{\alpha_n})$ which is an abstraction of $\gamma(B_1, \cdots, B_n)$, can be considered as a 1-safe finite Petri net. The set of structural invariants (traps and locks) and linear invariants of this Petri net defines a global abstract interaction invariant, which is computed symbolically by D-Finder. Finally, the concretization of this invariant gives an interaction invariant of the original system.

D-Finder relies on a semi-algorithm to prove invariance of Φ by iterative application of the rule (2). The semi-algorithm takes a composite component $\gamma(B_1, \ldots, B_n)$ and a predicate Φ. It iteratively computes invariants of the form $\mathcal{X} = \Psi \wedge (\bigwedge_{i=1}^{n} \Phi_i)$ where Ψ is an interaction invariant and Φ_i an invariant of component B_i. If $\mathcal{X}$ is not strong enough for proving that Φ is an invariant ($\mathcal{X} \wedge \neg\Phi = false$) then either a new iteration with stronger Φ_i is started or the algorithm stops. In this case, we cannot conclude about invariance of Φ.

Checking global deadlock-freedom of a component $\gamma(B_1, \ldots, B_n)$ is a particular case of proving invariants - proving invariance of the predicate $\neg DIS$, where DIS is the set of the states of $\gamma(B_1, \ldots, B_n)$ from which all interactions are disabled.

3.3 Platform Dependent Code Generation

The design flow provides the facility for generating code for the MPARM platform [9] from distributed system models in BIP. The generated code is targeted for a runtime called *Native Programming Layer* (NPL) implemented for MPARM. The runtime provides APIs for thread management, memory allocation, communication and synchronization. The code generation consists of two parts, the generation of the functional code and the generation of the glue code.

The functional code is generated from the application components consisting of processes and FIFOs. Processes are implemented as threads, and FIFOs are implemented as shared *queue* objects provided by the NPL library. Each process component is translated into a thread. The implementation in C contains the thread local data, queue handles and the routine implementing the specific thread functionality. The latter is a sequential program consisting of plain C computation statements and communication calls (e.g., *queue* API) provided by the runtime. A *read* transition is substituted by a *pop* API call on the respective queue handle. Similarly a *write* transition is substituted by a *push* API call on its respective queue handle.

The glue code implements the deployment of the application to the platform, i.e., allocation of threads to cores and the allocation of data to memories. The glue code is essentially obtained from the mapping. Threads are created and allocated to cores according to the process mapping. Data allocation deals with allocation of the thread stacks and allocation of FIFO queues for communication. In particular, for MPARM deployment, every thread stack is allocated into the $L1$ memory of the core to which the thread is deployed. Queue handles and queue objects are allocated from the cluster shared $L2$ memory. All these operations are implemented by using the API provided by the runtime.

The code generator has been fully integrated into a tool-chain and connected to the BIP system model generation flow. The generated code is compiled by the `arm-gcc` compiler. The compiled code is linked with the runtime library to produce the binary image for execution on the MPARM virtual simulator.

3.4 System Level Modeling and Performance Analysis

In the BIP design flow, system models are used to integrate the (extra-functional) hardware constraints into the software model according to some chosen deployment mapping. The abstract system model is constructed through a series of transformations from the BIP models of respectively the application software and the hardware platform. These two models are *composed* according to the mapping. The construction has been introduced in [15]. The transformations preserve functional properties of the application software model.

The abstract system model is then transformed for distributed implementation and progressively refined by including timing constraints for execution on the chosen platform. These constraints define execution times for elementary functional blocks, that is, BIP transitions within the application software model. More precisely, execution times are measured by running the executable code on MPARM. We measure the CPU time spent by each process performing blocks of computations. This is done by instrumenting the generated code with profiling API provided by the runtime. The API provides cycle accurate estimates for executing a block of code in each processor.

The instrumented system model is therefore used to analyze non-functional properties such as contention for buses and memory accesses, transfer latencies, contention for processors, etc. In the BIP design flow, these properties are evaluated by simulation of the system model extended with observers. Observers are regular BIP components that sense the state of the system model and collect pertinent information with respect to relevant properties i.e., delay for particular data transfers, blocking time on buses, etc. Actually, we provide a collection of predefined observers monitoring and recording specific information for most common non-functional properties.

Simulation is performed by using the native BIP simulation tool[1]. The BIP system model extended with observers is used to produce simulation code that runs on top of the BIP engine, that is, the middleware for execution/simulation of BIP

models. The outcome of the simulation with the BIP engine is twofold. First, the information recorded by observers can be used as such to gain insight about the properties of interest. Second, the same information can be used to build much simpler, abstract stochastic models. These models can be further used to compute probabilistic guarantees on properties by using statistical-model checking. This two-phase approach combining simulation and statistical model-checking has been successfully experimented in a different context[7]. It is fully scalable and allows (at least partially) overcoming the drawbacks related to simulation-based approaches, that is, long simulation times and lack of confidence in the obtained results.

4 Experiments

In this section, we report results about implementation and performance evaluation of two applications using the BIP design flow. We consider Cholesky factorization, a useful inverse-like operation on particular matrices, and MJPEG decoding, a streaming application for decoding of video streams. For both applications, we target the MPARM platform, which is a highly customizable, experimental, many-core platform available in the PRO3D project.

4.1 MPARM Platform

The MPARM [9] platform is a virtual ARM-based multi-cluster manycore platform. It is configured by the number of clusters, the number of ARM cores per cluster, and the interconnect between the clusters. The MPARM simulator allows experimentation with at most four clusters, each with eight ARM7-TDMI processors. The clusters are connected through a 2×2 NoC interconnect. The architecture is shown in Figure 5. Inside a cluster, each ARM core is connected with its private ($L1$) memory through a local bus. There is also a shared cluster memory ($L2$) which is connected with the cores through a cross-bar interconnect. A NoC-based infrastructure is used for inter-cluster communication, which consists of a router, a link, and the network interface (NI) of the individual clusters. The simulator provides cycle-accurate measurements for the execution on the virtual platform. Henceforth, we will use the term MPARM execution to denote execution on the MPARM virtual simulator.

As input to our design flow, we have used the hardware model in BIP generated from a structural description in DOL. The DOL description of the hardware architecture specifies resources connected by communication paths. Resources are of type computation (processors, memories) or communication (buses, crossbar interconnect, routers and links, etc.). Communication paths define the connections between the resources. A part of the DOL description of MPARM is given in Figure 6.

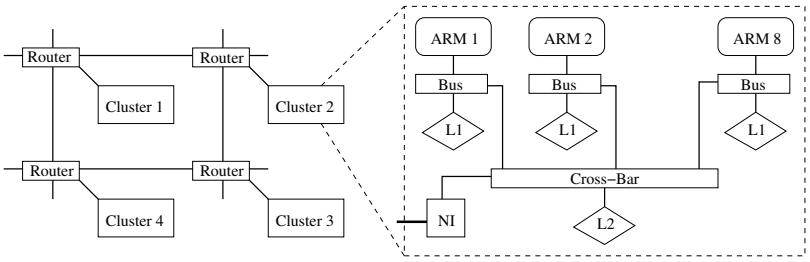

Fig. 5. An MPARM architecture with four clusters

```
<cluster name="C1" type="MPARM">
    <processor name="P1" type="ARMv7">
       <memory name="Private" type="L1">
            <configuration name="cycles" value="1"/>
       </memory>
       <hw_channel name="local" type="Bus">  </hw_channel>
    </processor>
    . . .
    <processor name="P8" type="ARMv7">
       <memory name="Private" type="L1">
            <configuration name="cycles" value="1"/>
       </memory>
       <hw_channel name="local" type="Bus">  </hw_channel>
    </processor>
    <hw_channel name="X-bar" type="CrossBar">
       <configuration name="cyclesperbyte" value="1"/>
    </hw_channel>
    <memory name="Shared" type="L2">
       <configuration name="cyclesperbyte" value="2"/>
    </memory>
</cluster>
```

Fig. 6. Fragment of the DOL description of an MPARM cluster

4.2 Cholesky Factorization

Cholesky Factorization decomposes a Hermitian positively-defined real-valued matrix A into the product $L \cdot L^T$ of a lower triangular real-valued matrix L and its conjugate transpose L^T. The Cholesky decomposition is used for solving numerically linear equations $Ax = b$. If A is symmetric and positive definite, then we can solve $Ax = b$ by first computing the Cholesky decomposition $A = L \cdot L^T$, then solving $Ly = b$ for y, and finally solving $L^T x = y$ for x.

The sequential Cholesky factorization algorithm has computational complexity $\mathcal{O}(N^3)$ for matrices of size $N \times N$. In this paper, our starting point is the sequential right-looking block-based version [25] provided as algorithm 1 which

Algorithm 1 Right-Looking Block-Based Cholesky Factorization

Require: *A Hermitian, positive definite matrix*
Ensure: $A = L \cdot L^T$, *L lower triangular*

```
for k = 1 to B do
    L_kk := seq-cholesky(A_kk)
    L_kk^-T := invert(transpose(L_kk))
    for i = k + 1 to B do
        L_ik := A_ik · L_kk^-T
    end for
    for j = k + 1 to B do
        L_jk^T := transpose(L_jk)
        for i = j to B do
            A_ij := A_ij − L_ik · L_jk^T
        end for
    end for
end for
```

provides immediate support for parallelization. In this algorithm, B denotes the number of blocks composing the original matrix A, that is $A = (A_{ij})_{1 \leq j \leq i \leq B}$ and every A_{ij} is a block matrix of size $K = N/B$. The algorithm computes the matrix L, block by block, such that $A = L \cdot L^T$. The algorithm 1 is easily parallelizable by separating computations related to different ij-blocks on different processes P_{ij}. Nevertheless, interactions between these processes are highly non-trivial. There are complex patterns for data dependencies, as illustrated in Figure 7 for the cases $B = 2, 3, 4$. Moreover, the amount of computation carried by each process is different. That is, as factorization proceeds, processes with higher indexes (i, j) become computationally more intensive. Furthermore, both data dependencies and the local amount of computation are tightly related to the decomposition size B as well as to the block size K. Altogether, finding optimal implementation on multi-processor platforms with fixed communication and computation resources is a non-trivial problem.

For every B, we denote by *Cholesky(B)* the Cholesky factorization using a $B \times B$ block decomposition. For our experiments, we implemented three versions in DOL, for respectively $B = 2, 3, 4$. In all cases, the process networks contain a *Splitter* process, a *Joiner* process and the computational processes for each block $(P_{ij})_{1 \leq j \leq i \leq B}$. Process *Splitter* splits the initial A matrix into blocks and dispatches them to computational processes. Every process P_{ij} implements the computation required on its corresponding matrix blocks A_{ij} and L_{ij}. As an example, the computational processes for *Cholesky(4)* are $P_{11}, P_{21}, P_{22}, P_{31} \ldots P_{44}$ as shown in Figure 7. The final L matrix is re-constructed by the *Joiner* process. Explicit communication between P_{ij} processes is used to enforce data dependencies. In these models, a dedicated FIFO is used for every pair of dependent processes to transfer the result block from the source to the target process. In the MPARM implementation, each computational process is deployed into an ARM processor and all the FIFO buffers are allocated to the $L2$ shared memory.

Table 1. DOL, BIP Models and MPARM Implementation Characteristics

		$B=2$	$B=3$	$B=4$
DOL Process Network	# processes	5	8	12
	# FIFOs	8	20	40
	# lines of code	864	1400	2171
BIP System Model	# components	40	120	181
	# interactions	182	445	882
	# lines of code	5207	7491	13648
MPARM implementation	# lines of code	1977	3163	4923

It is to be noted that for $B = 2, 3$ the implementation fits into a single cluster, and for $B = 4$, two clusters have been used. The magnitude of the different representations produced along the BIP design flow (number of processes, FIFOs, components, interactions, lines of code) is depicted in Table 1.

For every $B = 2, 3, 4$, we evaluate *Cholesky(B)* on 60×60 input matrices of double precision floating point numbers. Therefore, computational processes operate on matrix blocks of size 30×30, 20×20 and 15×15 for respectively $B = 2, 3, 4$. During the calibration phase, each computational routine on matrix blocks is characterized by the number of cycles required to execute it on an ARM processor. This is done by running the generated application code on MPARM and by accurate measurement of the number of cycles, for each routine. Table 2 reports the worst case execution times for different size of matrix blocks.

Table 3 presents an overview of the system-level performance analysis results obtained using two methods, respectively simulation of the system model *vs.* implementation and measurement of code execution on the MPARM platform.

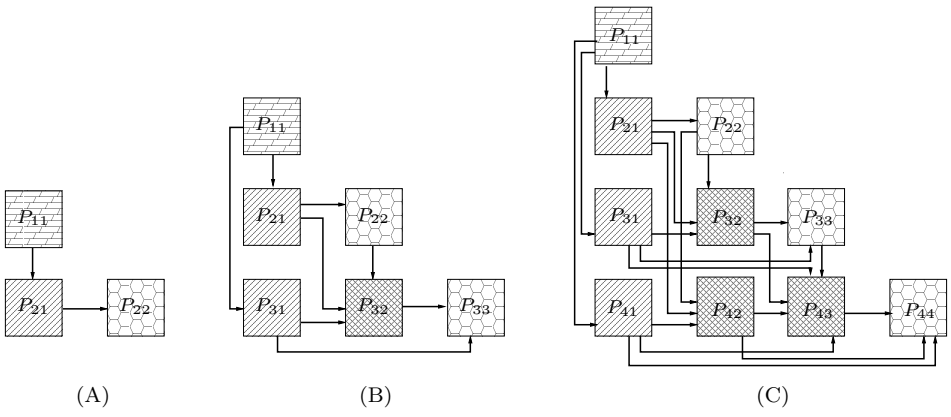

Fig. 7. Data dependencies for $2 \times 2(A), 3 \times 3(B)$ and $4 \times 4(C)$ process decomposition. Identical patterns indicate respectively a similar amount of local computation (processes) or potential for parallel communication (data dependencies).

Table 2. Execution times for computational routines on matrix blocks (in 10^6 cycles)

	$B = 2$	$B = 3$	$B = 4$
	$K = 30$	$K = 20$	$K = 15$
seq-cholesky	33.82	15.47	14.94
invert	34.85	16.06	15.47
transpose	0.13	0.08	0.08
multiply	115.64	53.23	47.16
tmultiply	104.80	45.01	34.89
subtract	1.66	1.05	1.05

Table 3. Performance Analysis: MPARM Execution *vs* BIP System Model Simulation

		$B = 2$	$B = 3$	$B = 4$
Total Execution Time (in 10^6 cycles)	MPARM Execution	317.70	229.58	-
	BIP System Model Simulation	325.23	277.69	356.00
	Accuracy	2.37%	20.95%	-
Analysis Time (in minutes)	MPARM Execution	$69'49''$	$34'25''$	-
	BIP System Model Simulation	$3'43''$	$7'54''$	$26'5''$
	Speed-up	18.78	4.35	-

For both methods, we report the *total execution time* taken by the application to run on the platform and the *analysis time*, that is, the time taken by the methods to produce the results. We point out that simulation of BIP system models produces fairly accurate results (max 20.95% relative error with respect to the cycle-accurate MPARM execution) while significantly reducing the analysis time (up to 19 times, in some situations). Note that for $B = 4$, the MPARM simulation did not terminate in 72 hours and the simulation data is unavailable. However, an estimate is obtained from the BIP system model simulation. A higher cycle count reflects the communication overhead due to the presence of two clusters with the NoC interconnect.

Finally, Figure 8 presents a detailed view of execution times and communication delays for computational processes for *Cholesky(4)*. For each process, the idle time denotes the waiting time spent before it gets access to read or write on FIFO channels. The communication time denotes the time effectively spent on reading or writing. The computation time denotes the total execution time without the idle and the communication time. The figure 8 (left) confirms that processes with higher indexes (i, j) are indeed computationally more intensive than the others. Additionally, the same processes are also idle for longer time than the others. This happens because of an increased number of data dependencies from processes with lower indexes (i, j). Communication time is impacted by memory conflicts. Memory conflicts occur when two different processes try to access simultaneously FIFO buffers located in the same shared memory. Figure 8 (right) depicts the delays due to memory conflicts for each process.

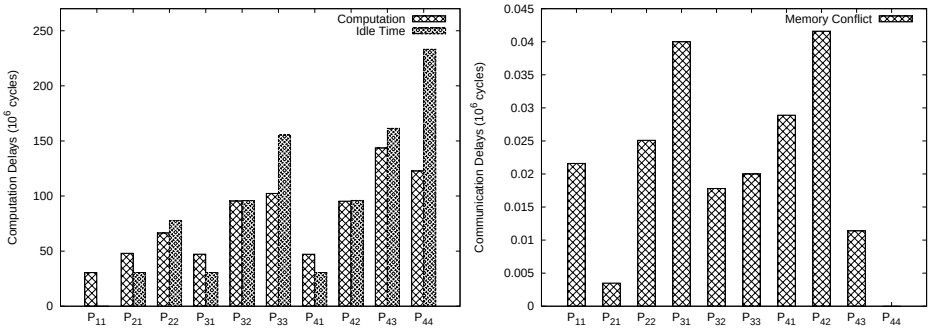

Fig. 8. Performance Results of Computational Processes in *Cholesky(4)*

4.3 MJPEG Decoding

The MJPEG decoder application software reads a sequence of JPEG frames and displays the decompressed video frames. The process network of the application software is shown in Figure 9. It contains five processes *SplitStream* (*SS*), *SplitFrame* (*SF*), *IqzigzagIDCT* (*IDCT*), *MergeFrame* (*MF*) and *MergeStream* (*MS*). The DOL description of the application processes contains approximately 1600 lines of C code.

The system model in BIP contains 42 atomic components with 198 interactions, and consists of approximately 7325 lines of BIP code. The implementation generated for MPARM is approximately 3174 lines of code.

For the experiments, we mapped the application on a single MPARM cluster. Each computational process is deployed into an ARM processor and all the FIFO buffers are allocated to the $L2$ shared memory. The performance results per process obtained by simulation of the system model are depicted in Figure 10. We remark that process *IqzigzagIDCT* is the heaviest in terms in computation, while process *MergeStream* stays idle most of the time. The low values of memory conflicts highlights the restricted parallelism within the application.

At system level, we measured the total execution time needed for the decompression of a single frame. Using BIP system model simulation, this time is estimated at 472.88 Mcycles. This result is very close to the cycle-accurate value obtained by measuring the MPARM execution, which is 468.83 Mcycles. The relative er-

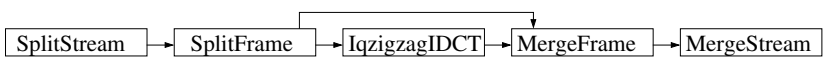

Fig. 9. Process Network of the MJPEG Decoder Application

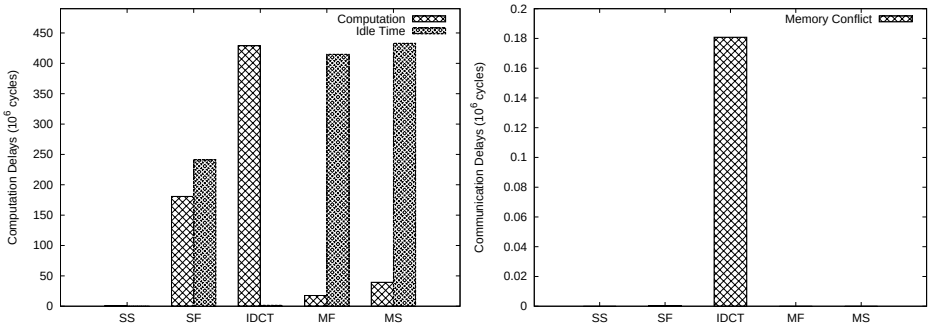

Fig. 10. Performance Results of Computational Processes in MJPEG Decoder

ror of our estimation is therefore less than 0.87%. Regarding analysis time, BIP system model simulation outperforms execution on (virtual) MPARM. The former completes in $9'46''$ and is approximately 5.2 times faster than the second, which completes in $50'48''$.

The above experiments show the capability of the BIP design flow for fine grain performance analysis on manycore platforms. It also shows the speedup compared to simulation based techniques, without adversely affecting the accuracy of the measurements.

5 Discussions

The presented method allows generation of a correct-by-construction system model for manycore platforms from an application software and a mapping. The method is based on source-to-source correct-by-construction transformation of BIP models. It is completely automated and supported by the BIP toolset. The system model is obtained by first refining the application software model and then composing it with the hardware architecture model. The composition is defined by the mapping. The construction of the system model is incremental and structure-preserving. This ensures scalability as the complexity of system models increases polynomially with the size of the application software and of the target hardware architecture. Mastering system model complexity is achieved thanks to the expressiveness of the BIP modeling framework.

The method clearly separates software and hardware design issues. It is also parameterized by design choices related to resource management such as scheduling policies, memory size and execution times. This allows estimation of the impact of each parameter on system behavior. Using BIP as a unifying modeling formalism for both hardware and software confers multiple advantages, in particular

rigorousness. The obtained system models are correct-by-construction. This is a main difference from other ad hoc model construction techniques.

When the generated system model is adequately instrumented with execution times, it can be used for performance analysis and design space exploration. Experimental results show the feasibility of the approach for fine grain analysis of architecture and mapping constraints on system behavior. The method is tractable and allows design space exploration to determine optimal solutions.

References

1. http://www-verimag.imag.fr/bip-tools,93.html
2. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems 17(3), 507–534 (1995)
3. Abdeddaim, Y., Asarin, E., Maler, O.: Scheduling with Timed Automata. Theoretical Computer Science 354, 272–300 (2006)
4. Henia et al., R.: System-level performance analysis - the SymTA/S approach. In: IEEE Proceedings Computers and Digital Techniques. vol. 152, pp. 148–166 (2005)
5. Alur, R., Henzinger, T.: Reactive modules. In: Proceedings of LICS'96. pp. 207–218. IEEE Computer Society Press (1996)
6. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Systems in BIP. In: Proceedings of SEFM'06. pp. 3–12. IEEE Computer Society Press (2006)
7. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. In: Proceedings of FMOODS/FORTE'10. LNCS, vol. 6117, pp. 32–46. Springer (2010)
8. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based design using the BIP framework. IEEE Software, Special Edition – Software Components beyond Programming – from Routines to Services 28(3), 41–48 (2011)
9. Benini, L., Bertozzi, D., Bogliolo, A., Menichelli, F., Olivieri, M.: MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. Journal of VLSI Signal Processing Systems 41, 169–182 (2005)
10. Bensalem, S., Bozga, M., Nguyen., T., Sifakis, J.: Compositional Verification for Component-based Systems and Application. In: Proceedings of ATVA'08. LNCS, vol. 5311, pp. 64–79. Springer (2008)
11. Bensalem, S., Bozga, M., Nguyen, T.H., Sifakis, J.: D-Finder: A Tool for Compositional Deadlock Detection and Verification. In: Proceedings of CAV'09. LNCS, vol. 5643, pp. 614–619. Springer (2009)
12. Bensalem, S., Lakhnech, Y.: Automatic generation of invariants. FMSD 15(1), 75–92 (1999)
13. Bensalem, S., Bozga, M., Legay, A., Nguyen, T.H., Sifakis, J., Yan, R.: Incremental Component-based Construction and Verification using Invariants. In: Proceedings of FMCAD'10. pp. 257–256. IEEE (2010)
14. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A Framework for Automated Distributed Implementation of Component-based Models. Distributed Computing (2012), to appear
15. Bourgos, P., Basu, A., Bozga, M., Bensalem, S., a nd K. Huang, J.S.: Rigorous system level modeling and analysis of mixed HW/SW systems. In: Proceedings of MEMOCODE'11. pp. 11–20. IEEE/ACM (2011)

16. Chandy, K., J.Misra: Parallel program design: a foundation. Addison-Wesley Publishing Company (1988)
17. Clarke, E., Long, D., McMillan, K.: Compositional model checking. In: Proceedings of LICS'89. pp. 353–362 (1989)
18. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity: The Ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (2003)
19. Erbas, C., Pimentel, A.D., Thompson, M., Polstra, S.: A framework for system-level modeling and simulation of embedded systems architectures. EURASIP Journal on Embedded Systems 2007 (2007)
20. Feiler, P.H., Lewis, B., Vestal, S.: The SAE Architecture Analysis and Design Language (AADL) Standard: A basis for model-based architecture-driven embedded systems engineering. In: Proceedings of RTAS Workshop on Model-driven Embedded Systems. pp. 1–10 (2003)
21. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16(3), 843–871 (1994)
22. Kienhuis, B., Deprettere, E., Vissers, K., van der Wolf, P.: An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings of ASAP'97. pp. 338–349. IEEE Computer Society (1997)
23. Künzli, S., Poletti, F., Benini, L., Thiele, L.: Combining Simulation and Formal Methods for System-level Performance Analysis. In: Proceedings of DATE'06. pp. 236–241 (2006)
24. Kupferman, O., Vardi, M.Y.: Modular Model Checking. In: Revised Lectures of COMPOS'97. LNCS, vol. 1536, pp. 381–401 (1998)
25. Leary, D.P., Stewart, G.: Data-flow algorithms for parallel matrix computations. Communications of the ACM 28(8), 840–853 (1985)
26. Lieverse, P., Stefanov, T., van der Wolf, P., Deprettere, E.: System level design with SPADE: an M-JPEG case study. ICCAD pp. 31–38 (2001)
27. McMillan, K.L.: A compositional rule for hardware design refinement. In: Proceedings of CAV'97. LNCS, vol. 1254, pp. 24–35. Springer (1997)
28. Moussa, I., Grellier, T., Nguyen, G.: Exploring SW Performance Using SoC Transaction-Level Modeling. In: Proceedings of DATE'03. pp. 20120–20125 (2003)
29. Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., Deprettere, E.: Daedalus: toward composable multimedia mp-soc design. In: Proceedings of DAC'08. pp. 574–579. ACM (2008)
30. OMG: OMG Systems Modeling Language SysML (OMG SysML). Object Management Group (2008)
31. Pnueli, A.: In transition from global to modular temporal reasoning about programs pp. 123–144 (1985)
32. PRO3D: Programming for Future 3D Architecture with Many Cores, FP7 project funded by the EU under grant agreement 248 776, http://pro3d.eu/
33. Salah, R.B., Bozga, M., Maler, O.: Compositional Timing Analysis. In: Proceedings of EMSOFT'09. pp. 39–48 (2009)
34. Stark, E.W.: A proof technique for rely/guarantee properties. In: Proceedings of FSTTCS'85. vol. 206, pp. 369–391. Springer (1985)
35. Thiele, L., Bacivarov, I., Haid, W., Huang, K.a.: Mapping Applications to Tiled Multiprocessor Embedded Systems. In: Proceedings of ACSD'07. pp. 29–40. IEEE Computer Society (2007)
36. Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: Proceedings of ISCAS'02. vol. 4, pp. 101–104. IEEE (2002)