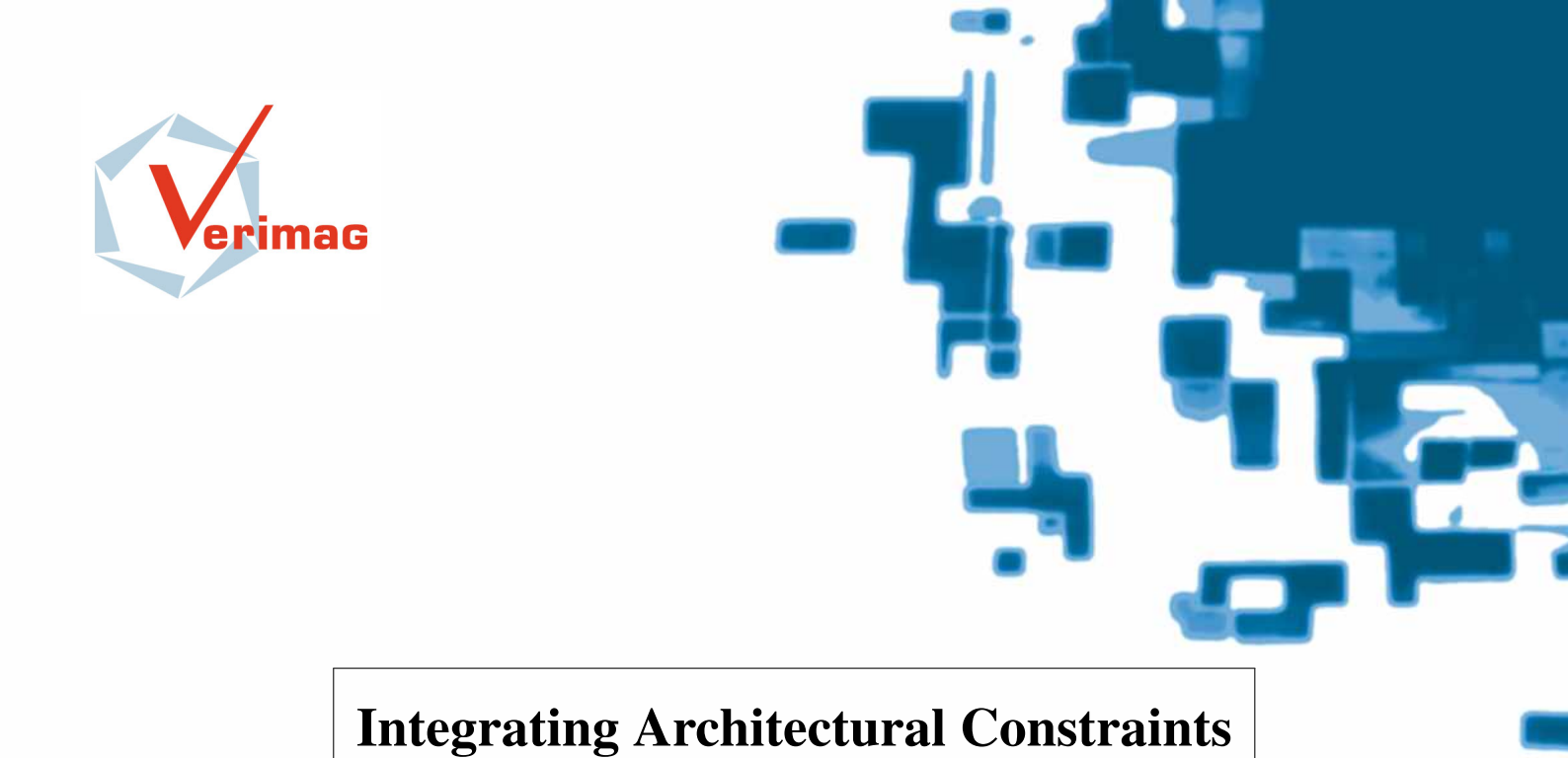

Integrating Architectural Constraints in Application Software by Source-to-Source Transformation in BIP

P. Bourgos, A. Basu, S. Bensalem, K. Huang, J. Sifakis

Verimag Research Report n^o TR-2011-1

01-01-2011

Reports are downloadable at the following address
http://www-verimag.imag.fr

Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Integrating Architectural Constraints in Application Software by Source-to-Source Transformation in BIP

P. Bourgos, A. Basu, S. Bensalem, K. Huang, J. Sifakis

01-01-2011

Abstract

Performance of embedded applications strongly depends on features of the hardware platform on which they are deployed. A grand challenge in complex embedded systems design is developing methods and tools for modeling and analyzing the behavior of an application software running on a given hardware architecture. We propose a rigorous method that allows to obtain 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 a model of the application software in BIP, a model of the hardware architecture in XML and a mapping associating read and write operations of the application software with execution paths in the architecture. It builds a model of the corresponding mixed hardware/software system in BIP. The latter can be simulated and analyzed for verification of both functional and extra-functional properties. The method consists in progressively enriching the application software model. It involves three steps: 1) The generation of a BIP model of the application software; 2) The generation of a BIP model of the hardware architecture; 3) The composition of the two models. The steps are implemented by application of source-to-source transformations that are correct-by-construction. In particular they preserve functional properties of the application software. The obtained system model is highly parametrized and allows flexible integration of specific target architecture features, such as bus policy and scheduling policy of the processors. The method has been implemented for application software and hardware architectures described in the DOL tool for performance evaluation. It is illustrated through the construction of a system model of an MJPEG application running on an MPARM architecture.

Keywords: System Level Design, Hardware Architecture, Mapping, Hardware Constraints, Performance Estimation

Reviewers:

Notes: This work is supported by the PRO3D EU project

How to cite this report:

```
@techreport {TR-2011-1,
   title = {Integrating Architectural Constraints in Application Software by Source-to-Source
Transformation in BIP},
   author = {P. Bourgos, A. Basu, S. Bensalem, K. Huang, J. Sifakis},
   institution = {{Verimag} Research Report},
   number = {TR-2011-1},
   year = {2010}
}
```

1 Introduction

Performance of embedded applications strongly depends on features of the underlying hardware platform. For application software running on multicore or distributed platforms rigorous performance analysis techniques are essential for determining optimal implementations with respect to resource management criteria. These techniques require the use of faithful models of mixed hardware/software systems. The models should be founded on rigorous semantics and be suitable for analysis and design space exploration.

There exist performance evaluation techniques applied on very abstract system models. Some use formal analytical models representing a system as a network of nodes exchanging streams. The dynamics of the execution platform is characterized by execution times. These techniques allow only estimation of pessimistic worst-case execution delays. DOL [14] provides system level performance analysis based on formal analysis techniques using Real Time Calculus [15]. It also offers multi-objective mapping and optimizations. A similar performance evaluation framework is SymTA/S [8]. There also exist performance analysis techniques based on Timed-Automata [13, 10, 2, 9]. These can be used for modeling and solving scheduling problems.

Other approaches for performance evaluation use ad hoc executable system models e.g., models in SystemC [7]. They combine a model of the application software derived from a C/C++ based design flow, and a hardware architecture described in TLM [12]. The obtained system models may be useful for debugging, but are not adequate for thorough exploration of the hardware architecture dynamics and its effects on the software execution. Furthermore, long simulation time is a major drawback.

Finally, an approach combining simulation and analytic models is presented in [11], where simulation results can be propagated to analytic models and vice versa through well defined interfaces.

We propose a performance evaluation method that is both rigorous and allows a fine analysis of system dynamics. It is rigorous because it is applied to system models that are faithful, have precise semantics and thus can be analyzed by using formal techniques. A system model is derived by progressively integrating constraints induced on an application software by the underlying hardware architecture. Both models are described in BIP [3]. In contrast to ad hoc modeling approaches, the system model is obtained from a BIP model of the application software and a description of the hardware architecture, by application of source-to-source transformations that are correct-by-construction [6].

The paper is structured as follows. Section 2 presents the method and the main steps in the design flow, with a brief introduction to the BIP component framework in section 2.1, translation of application software into BIP in section 2.2 and generation of the system model in section 2.2. Tool impementation and experimental results are shown in section 3. In section 4 we conclude and discuss future work directions.

2 The Proposed Method

The method is illustrated in Figure 1. The application software is translated into a BIP model. We assume that it consists of a set of processes communicating through fifo channels by executing atomic write/read operations. Each process computes a function transforming local data described in a programming language e.g. C. The BIP model of the application software is transformed by taking into account a model of a target hardware architecture and a mapping. Hardware architecture is described by an abstract grammar specifying its structure as the interconnection of physical devices such as processors, buses and memories. The mapping associates with each block of a partition of processes a processor of the hardware architecture and a scheduler for managing shared resources. It induces a correspondence between atomic write/read operations of the application software and execution paths. These are sequences of operations of the hardware architecture which are refinements of atomic operations. The method is completely automated and has been implemented in a tool. The tool uses as a frontend the DOL tool for performance evaluation, that is the application software, the architecture and the mapping are described in DOL. By using the BIP toolset the produced system model can be: 1) simulated/validated on a linux PC; 2) checked for functional correctness using the D-Finder tool; 3) transformed to generate code for execution on distributed architectures; and 4) analyzed to estimate delay bounds by simulation or statistical model checking techniques.

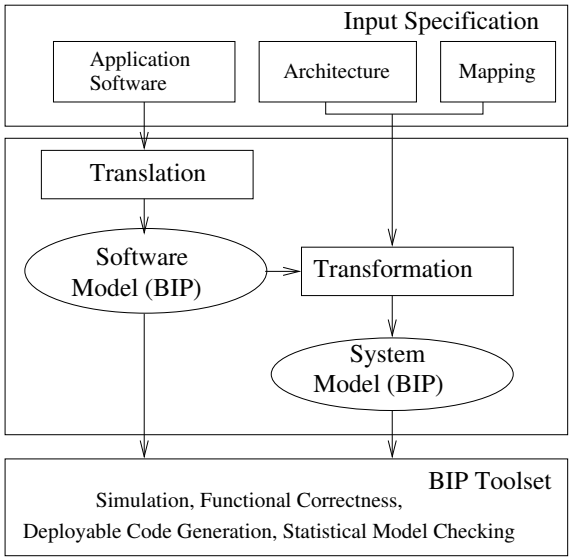

Figure 1: The Method

2.1 The BIP Component Framework

The BIP design flow is entirely supported by the BIP language and its associated toolset. 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 automata or Petri nets extended with data and functions described in C/C++. Transitions are labelled with ports (action names), guards (enabling conditions on the state of a component) as well as functions (computations on local data). The description of coordination between components is layered. The first layer describes the interactions between components by using connectors. An interaction is a set of strongly synchronized ports. It is labelled with guards (enabling conditions) and data transfer functions (data exchange) between interacting components. The second layer describes dynamic priorities between 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 [5]. 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. Figure

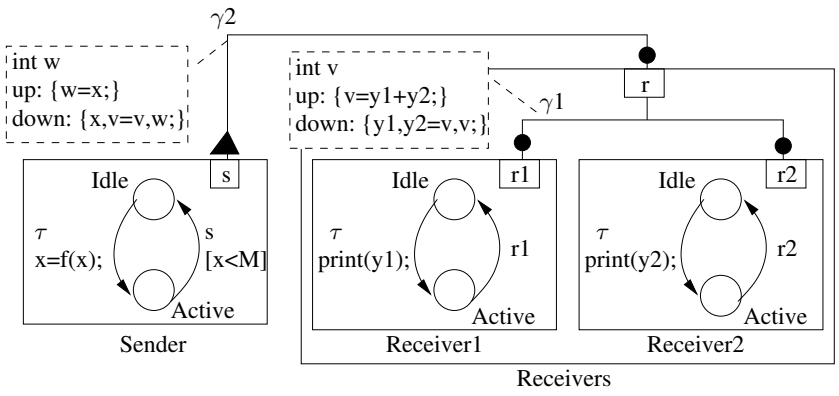

Figure 2: An example of a BIP model

2 shows a graphical representation of a BIP model. 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 $\gamma 1$, which represents a rendezvous interaction between ports *r1* and *r2*, leading to the composite component *Receivers*. The composite component exports $\gamma 1$ by using port *r*. As a result of the data transfer in $\gamma 1$, the sum of the local variables *y1* and *y2* is exported through the port *r* by using variable *v*. The interaction is completed by assigning *v* to variables *y1* and *y2*. The model is the composition of *Sender* and *Receivers* using the connector $\gamma 2$ which represents a broadcast 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*.

2.2 Translating Application Software into BIP

2.2.1 Description of the Application Software

In DOL [14] an application software is a process network that consists of three basic entities: $Process$, $Channel$, and $Connection$, described by an abstract grammar as follows:

Application_Software ::= Process$^+$. Channel$^+$. Connection
Process ::= (InPort | OutPort)$^+$. Behavior
Channel ::= RecvPort . SendPort
Connection ::= Read_Connection | Write_Connection
Write_Connection ::= OutPort . RecvPort
Read_Connection ::= SendPort . InPort
Behavior ::= function

Each process $Pross$ has input ports $Pross.InPort_i$, output ports $Pross.OutPort_j$ and behavior $Pross.Behavior$. Each channel Cha has a single input port $Cha.RecvPort$ and a single output port $Cha.SendPort$. A $Write\_Connection$ between a process $Pross$ and a channel Cha is a pair ($Pross.OutPort$, $Cha.RecvPort$). A $Read\_Connection$ between a process $Pross$ and a channel Cha is a pair ($Cha.SendPort$, $Pross.InPort$).

Application software can be represented as a bipartite graph with two kinds of nodes: $Processes$ and $Channels$. There exist edges corresponding to $Write\_Connections$ relating output ports of $Processes$ to receive ports of $Channels$ as well as edges corresponding to $Read\_Connections$ relating send ports of $Channels$ to input ports of $Processes$. We assume that each $Channel$ has one $Write\_Connection$ and one $Read\_Connection$. Also, each output and each input port of a $Process$ is uniquely associated to a $Write\_Connection$ and a $Read\_Connection$ respectively.

2.2.2 Generation of the Application Software Model in BIP

Each node in the application software graph defines an atomic component in BIP. For a process $Pross$, its behavior $Pross.Behavior$ is defined as a function in C which contains specific communication primitives ($write$ and $read$) for inter process communication, in addition to C statements. A $read$ operation reads data from a port $Pross.InPort_i$, and a $write$ operation writes data to a port $Pross.OutPort_j$. The function defines a sequence of computation statements and calls to the $read$/$write$ primitives, encapsulated in control statements. The function also contain calls to a special primitive ($detach$) used to terminate the process. The behavior of each process is invoked in a loop and exits when the termination ($detach$) primitive is called.

The translation converts $Pross.Behavior$ of a process $Pross$ to an extended automaton, describing the behavior of the atomic component. Each port of $Pross$ is defined as a port in the atomic component. Data structure defined in the C code are directly translated as data in the atomic component. Control locations correspond to $read$/$write$ primitives for which synchronization is required. Transitions are labeled by the port name associated with the primitives. Computation statements are added as actions of the transitions.

The translation requires analysis of arbitrary C code and hence is non-trivial. It starts by parsing the C source code of a process into an itermediate object model. The translation to BIP is done in two steps.

In the first step, the interaction points in the code are identified. Each call to a $read/write$ primitive is registered as an interaction point.

The second step involves the generation of the automaton from the C statements. For every call to a $read/write$, a control location is created. An outgoing transition is added from this location, labeled by the port used in the primitive. This transition models the primitive call, which requires synchronization with a software FIFO component. The port of the transition is associated with data that is read/written by the primitive invocation. Additional assignment statements are added to load/store the data into the local variables in the function.

A block statement that contains interaction points is transformed into sequence of control locations and transitions in the automaton. For such statements, e.g., conditional (if-else, switch) or loop (for, while) or control satatement (break, continue, return), additional control locations are created and internal transitions guarded by the control condition are added to model the control automaton.

For a conditional statement, a new control location is created with an incoming transition where the branch condition evaluation action is added. Outgoing transitions, one for the positive branch and another for the negative branch are created. The branches are finally merged to a new control location.

For a loop statement, a new control location is created with an incoming transition where the loop initialization action and exit condition are added. Outgoing transitions, one for the positive exit condition and the other for the negative exit condition are created. For the negative exit branch, a transition back to the starting location of the loop is added, with the exit condition updation action.

Statements that do not contain interaction points are added as actions to the existing transition. Subroutine calls that contain $read/write$ primitive calls (either directly or through nested subroutines) are inlined in the automaton.

For the termination primitive ($detach$), a control location with an incoming transition and without any outgoing transition is created.

From the last control location generated in the automaton, a transition to the starting contol location is added. This models the invocation of the process behavior in a loop at runtime. The termination of the process behavior is modeled as a move to a deadlocked location, that corresponds to the $detach$ primitive call.

The generation of the automata is restricted to non-recursive subroutine calls and without usage of global data structures. However, global read only data structures are translated as local data in the individual components.

An example of the translation of a function $process\_fire()$ into an atomic component in BIP is shown in figure 3. The function defines local integer variables i, j, k and $size$. Its invocation reads data from port $inPort$ of size $size$ into the variable i. It then performs an iteration of local computation and writes to port $outPort$ the value k.

The BIP component generated from $process\_fire()$ has ports $inPort$, $outPort$ and control locations $L1$, $L2$ and $L3$. i, j, k and $size$ are defined as variables in the component. i is associated with $inPort$ and k is associated with $outPort$. At $L1$, the component awaits synchronization through $inPort$ corresponding to the $read$ primitive call. At $L3$ it awaits synchronization through $outPort$ corresponding to the $write$ primitive call. At $L2$, the component can perform internal transitions (labeled by τ) guarded by the loop control condition. Computations between successive $read/write$ calls in $process\_fire()$ are added as action statements with the respective transitions in the generated automaton.

For a channel Cha, the behavior is modeled as a predefined BIP atomic component modeling a FIFO represented in figure 4. It has ports $recvPort$ and $sendPort$, and a single control location. It contains an array variable $buff$ parametrized by size N. The variable x associated with $recvPort$ gets the received value which is inserted into $buff$. The variable y associated with $sendPort$ contains the value to be read next. The FIFO policy is implemented by using the two indices i and j, for insertion and deletion respectively from $buff$.

Each edge of the application software graph representing $Write\_Connections$ and $Read\_Connections$ defines a BIP connector which strongly synchronizes the corresponding ports, as shown in the example of figure 5. This consists of a process Pross_1 sending data to Pross_2 and Pross_3 by using channels Cha_1 and Cha_2 respectively. The connectors are associated with transfer of data, implementing the $read$ and $write$ operations. A connector implementing $write$ transfers data from a process to a channel, whereas the one implementing $read$ transfers data from a channel to a process.

```
process_fire() {
  int i, j, k;
  int size = sizeof(i);
  read(inPort, i, size);
  i++;
  for (j=0; j<10; j++) {
    i=i+j;
    k = compute(i);
    write(outPort, k, size);
    i–;
  }
}
```

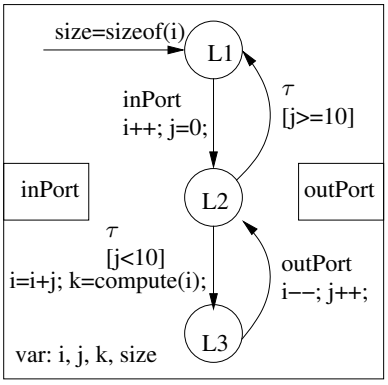

Figure 3: Translating a process into a BIP component

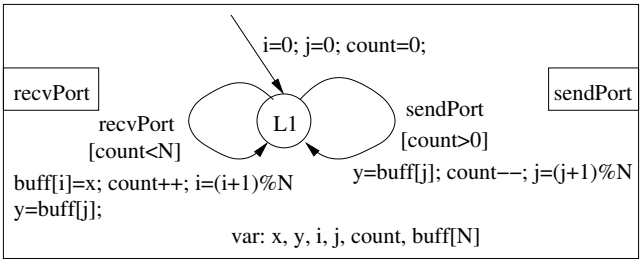

Figure 4: FIFO channel in BIP

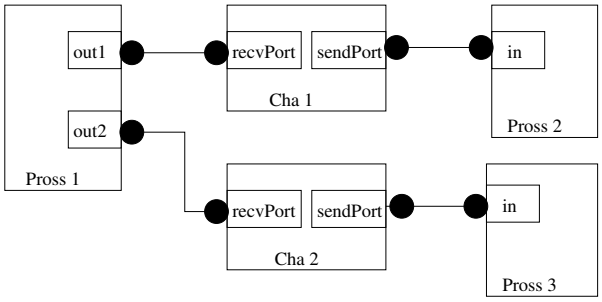

Figure 5: Example application software model in BIP

2.3 Integrating Architecture Constraints in the Application Software

2.3.1 Description of Architecture and Mapping

A harware architecture consists of resources and communication paths. Resources are used for computation, e.g., processor and memory, or for communication, e.g., bus. Communication paths define paths between computational resources using communication resources. This is an abstract syntax of harware architecture.

Hardware_Architecture ::= Resource$^+$. Comm_Path$^+$
Resource ::= Processor$^+$ | Memory$^+$ | Bus$^+$
Comm_Path ::= Processor . Bus$^+$. Memory . Bus$^+$. Processor

BIP model of Harware Architecture: A hardware architecture is modeled as a template composite component in BIP which is the composition of generic components of type $Processor$, $Memory$ and

Bus. Each $Processor$ is a composite component C with ports $C.wr\_begin$, $C.wr\_end$, $C.rd\_begin$, $C.rd\_end$, corresponding to the initiation and termination of write and read operations. A generic model of a processor $Pror$ is shown in figure 6. It may have several read and write ports.

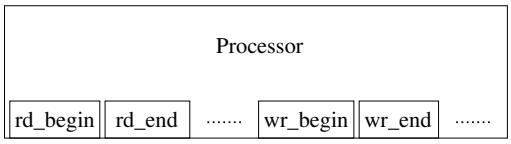

Figure 6: Generic processor interface

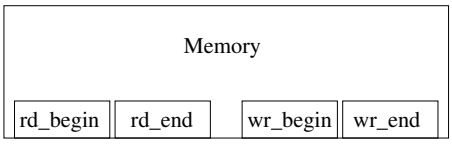

Figure 7: Generic memory interface

A $Memory$ has four ports each one corresponding to begin and end of writing and reading, as shown in figure 7.

A Bus is a composite component configured by the number of its master, slave, and the bus scheduling policy. For each master, a bus C has the following master ports: $C.wr\_req$, $C.wr\_ack$ corresponding to begin and end of writing, and $C.rd\_req$, $C.rd\_ack$ corresponding to begin and end of reading; for each slave, it has the following slave ports: $C.wr\_begin$, $C.wr\_end$ corresponding to begin and end of writing, and $C.rd\_begin$, $C.rd\_end$ corresponding to begin and end of reading. Each master port is connected to each slave port internally through virtual links, explained in details in section 2.3.2. A master port is connected to a master component (i.e., processor) which initiate data transfer in the bus. For a $write$ operation, the processor sends the data, its size and the address of the memory where the data has to be written. For a $read$, the processor sends the address of the memory to be read and the data size. A slave port is connected to a slave component (i.e., memory) which respond to data transfer initiated by some master component.

The generic architecture of a bus component configured for two master and one slave, with the connections to the master and the slave is shown in figure 8.

A hardware architecture can be represented as a graph with three kinds of nodes: processor, memory and bus. A communication path describes a path between two processors, via a common memory, using one or more buses. It is a sequence of nodes of the form $Pror_i.(Bus_1^i \dots Bus_n^i).Mem.(Bus_1^j \dots Bus_m^j).Pror_j$ with edges between the nodes. The edges represent the flow of information from $Pror_i$ to $Pror_j$ through two sets of buses and the target memory Mem. Edges relate ports $C.wr\_begin$, $C.wr\_end$ and $C.rd\_begin$, $C.rd\_end$ of a master component (processor) C to the master ports $C'.wr\_req$, $C'.wr\_ack$ and $C'.rd\_req$, $C'.rd\_ack$ respectively of a successor bus component C'. Similarly, edges relate the slave ports $C'.wr\_begin$, $C'.wr\_end$ and $C'.rd\_begin$, $C'.rd\_end$ of the bus C' to ports $C''.wr\_begin$, $C''.wr\_end$ and $C''.rd\_begin$, $C''.rd\_end$ of a slave component (memory) C''.

The master and slave ports of the bus induces some rule defining the edges between the nodes: 1) A processor can only be connected to the master port of a bus, 2) A memory can only be connected to the slave port of a bus, 3) A processor cannot be directly connected to a memory, and, finally 4) A bus can be connected to another bus only through a bridge component (not considered in our current model), which acts as a slave for one and the master for the other bus.

Each edge of the graph defines a BIP connector which strongly synchronizes the corresponding ports. The behavior of the connector implements the transfer of data, its address and size between the successessive components, corresponding to the $write$ and $read$ operation.

A communication path can be decomposed into a $Write\_Path$: $Pror_i.(Bus_1^i \dots Bus_n^i).Mem$ and a $Read\_Path$: $Mem.(Bus_1^j \dots Bus_m^j).Pror_j$. That is

Write_Path ::= Processor . Bus$^+$. Memory
Read_Path ::= Memory . Bus$^+$. Processor

Mapping Given an $Application\_Software$ and a $Hardware\_Architecture$, a mapping Map associates: 1) a set of processes $Pross_1 \dots Pross_n$ to a processor $Pror$ and a $Scheduler$, 2) a set of channels $Cha_1 \dots Cha_k$ to a memory Mem.

A mapping Map should be consistent: If there is a $write\_connection$ from $Pross$ to Cha in the application software, there should be a $write\_path$: $Map(Pross).(Bus_1 \dots Bus_n).Map(Cha)$ in the

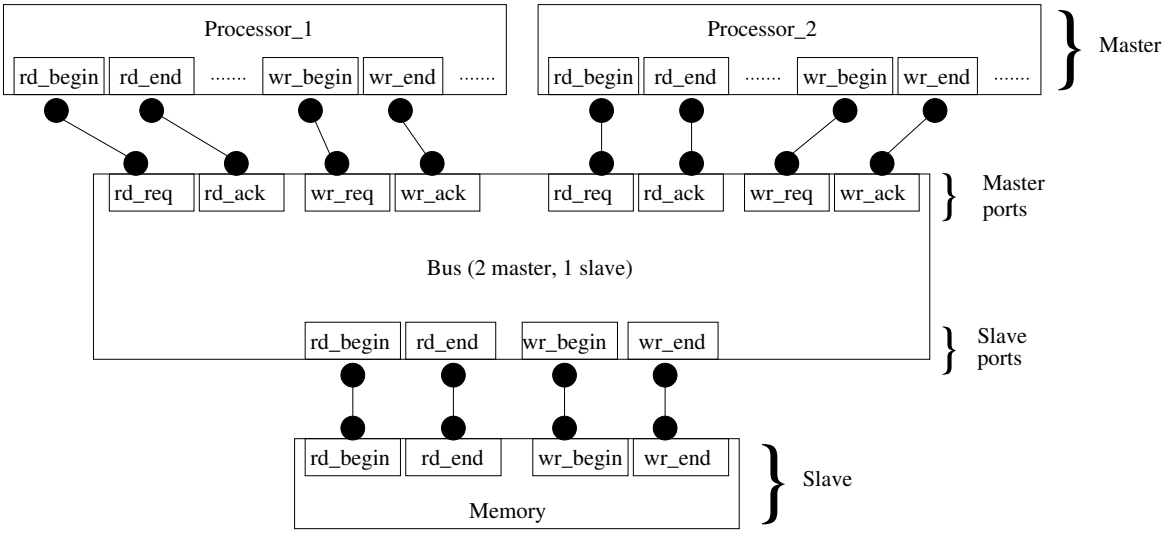

Figure 8: Generic bus interface

graph of the hardware architecture. Similarly, if there is a $read\_connection$ from Cha to $Pross$, there should be a $read\_path$: $Map(Cha).(Bus_1 \dots Bus_k).\ Map(Pross)$.

A $Scheduler$ is a component with ports acq and rel. A simple model of a scheduler that models mutual exclusion among the processes mapped on a processor is shown in figure 9.

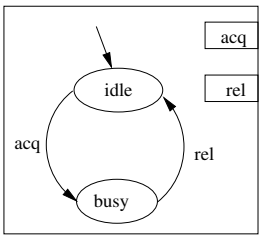

Figure 9: Scheduler component

2.3.2 Generation of the System Model in BIP

We describe the method for generating a BIP System model from an application software model and a harware architecture for a given mapping.

If $Pror$ is a processor to which are mapped processes $Pross_1, \dots, Pross_n$ then the component associated with $Pror$ will contain components $Pross'_1 \dots Pross'_n$ such that $Pross'_i$ is obatained from $Pross_i$ by:

1. Breaking atomicity of $write$ and $read$ operations: each $OutPort$ is replaced by $OutPort\_begin$ and $OutPort\_end$. Similarly, each $InPort$ is replaced by $InPort\_begin$ and $InPort\_end$. This is obtained by adding new control locations for each read/write operations in the behavior of the process, as shown in figure 10.

2. Adding interactions with the scheduler: ports acq and rel are added for interaction with the scheduler $Pror.Scheduler$ of $Pror$. The port acq is for acquiring the processor and rel is for releasing the processor. A process acquires the processor at the start of its behavior. It releases the processor on its termination. This is shown in the transformed behavior of a process in figure 11.

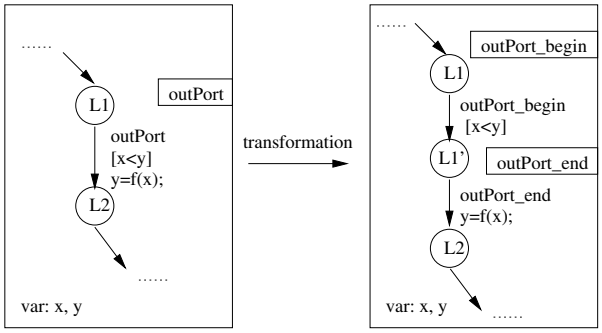

Figure 10: Breaking atomicity of a $write$

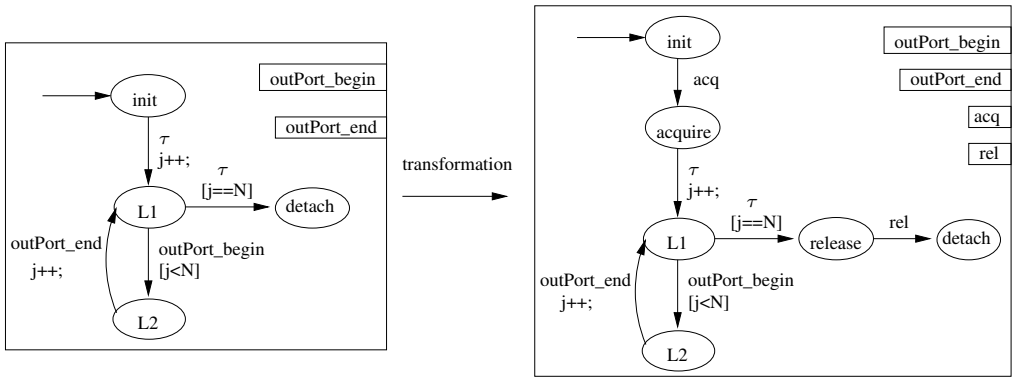

Figure 11: Adding ports acq and rel to a process

Additionally, $Pror$ contain components $FIFO\_write$ and $FIFO\_read$ for implementing the $write$ and $read$ operation respectively. Each channel Cha in the application software is decomposed into $FIFO\_write$, $FIFO\_read$ and a $buffer$.

Cha ::= FIFO_write . FIFO_read . Buffer

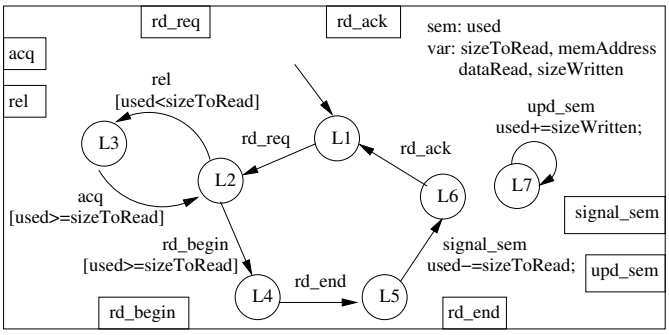

Figure 12: FIFO_read component

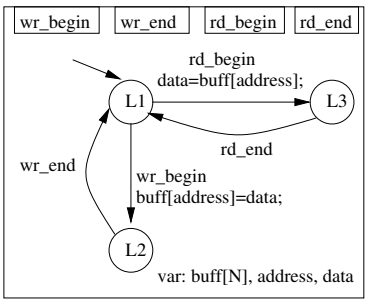

Figure 13: Buffer component

A $FIFO\_read$ component (figure 12) implements read action of Cha. It has the ports rd_req, rd_ack for its interface with a process $read$ operation, and ports rd_begin, rd_end for its interface with the buffer. A $FIFO\_write$ implements the write action of Cha in the same manner.

The decomposition of a channel into a $FIFO\_write$, $FIFO\_read$ and a buffer is shown in figure 14. The $FIFO\_write$ and the $FIFO\_read$ require synchronization with each other in order to preserve the size invariant of the buffer. This is implemented by strong synchronization between the $signal\_sem$ and

upd_sem ports. They also have ack and rel ports for interaction with the processor scheduler. This is required to implement blocking $read/write$ operation without blocking the processor.

The channel buffer $Cha.buff$ becomes a buffer component $Buffer$ (figure 13). It has ports wr_begin, wr_end and rd_begin, rd_end for writing and reading respectively. The ports for writing synchronizes with a $FIFO\_write$ and the ports for reading synchronizes with a $FIFO\_read$. The architecture of a

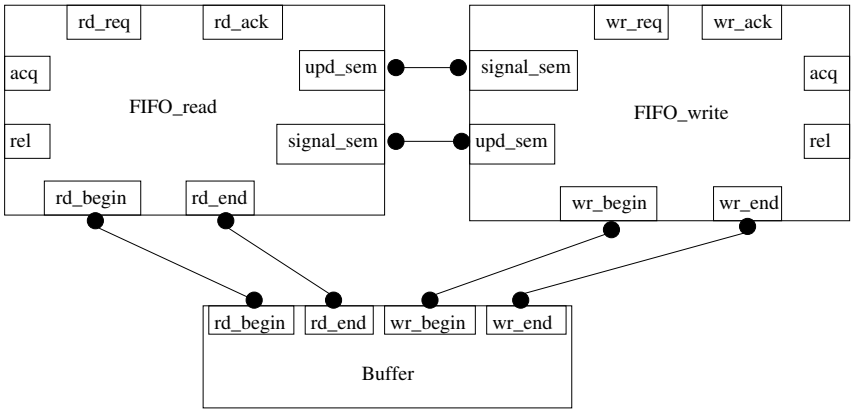

Figure 14: Splitting a channel

composite processor component with its sub components is shown in figure 15

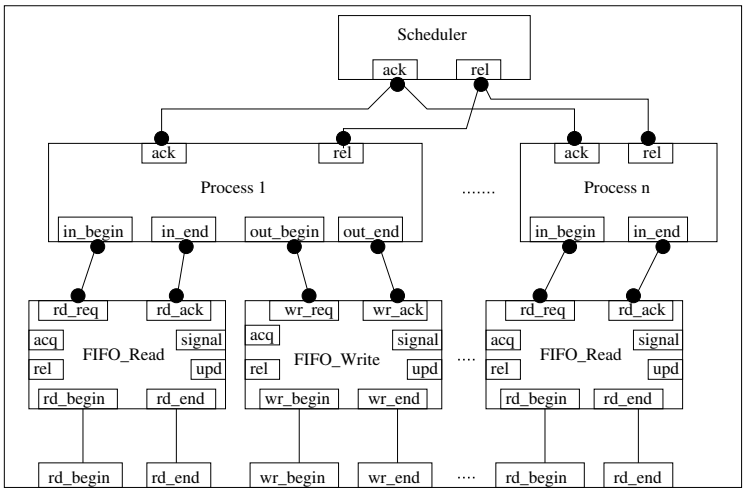

Figure 15: Processor architecture

If Mem is a memory to which are mapped channels $Cha_1, \ldots, Cha_n$ then the component associated with Mem will contain components $Buff'_1 \ldots Buff'_n$ (shown in figure 16) such that $Buff'_i$ is obtained from the decomposition of Cha_i. We assume high cache rate for the local variables of the processes mapped on a processor, and hence we do not model explicitly the allocation of process data in the memory. The memory is used only to model inter process data communications.

A bus component Bus is the composition of $Master\_Interface$, $Slave\_Interface$ and $Virtual_Link$s, and contain a bus scheduler. For the interfaces and virtual links, we have separate models for read and write operations. A $Master\_Interface\_Read$ component is essentially a multiplexer, which decodes the address of a read request (rd_req) and activates the corresponding virtual link in order to route the request to the destination slave. The model is shown in figure 17. A $Virtual\_Link$ models the physical connection between the master and slave interfaces. Figure 18 shows the BIP model of a virtual link for a read operation.

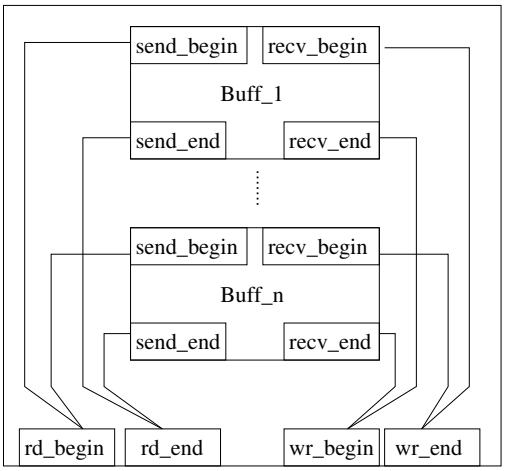

Figure 16: Memory architecture

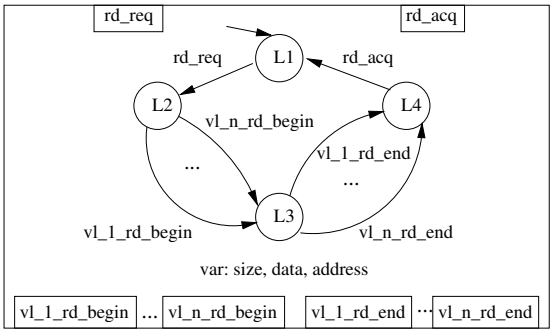

Figure 17: Bus: Master read interface component

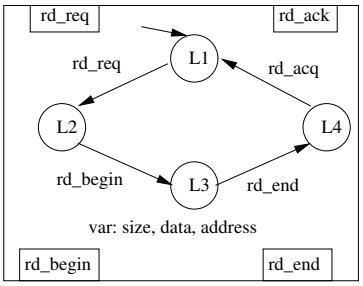

Figure 18: Bus: Virtual link component

The model of a bus component configured using two master and two slave ports is partially shown in figure 19. It shows the components realizing the bus read. The bus write is modeled using a similar set of components, and sharing the common scheduler. We modeled a set of scheduler components for realizing different bus scheduling policies like fixed-priority, round-robin and TDMA.

Addressing the memory for a $read$ and a $write$ operation is based on generation of a global address which consists of a base address and a relative address. The base address identifies the channel buffer, and the relative address identifies an element in the buffer. From the mapping of a channel to a memory, its base address is determined. This base address is taken into account by the FIFO components corresponding to the channel in order to generate the addresses for the $read$ and $write$ operations. In the bus, the master interface decodes the base address to identify the virtual link for communication, which determines the slave port and the memory connected to it. In the memory $read/write$ operation, the relative address is used to read or write the particular memory cell.

The construction of the system model uses a set of generic components provided as a library of system components listed in figure 20. The library is classified into software, hardware dependent software, and hardware components.

3 Implementation and Experimental Results

3.1 Tool Implementation

The method described in section 2 has been implemented in a tool. DOL is used as a frontend to describe the application software as a process network, the hardware architecture as well as the mapping. The

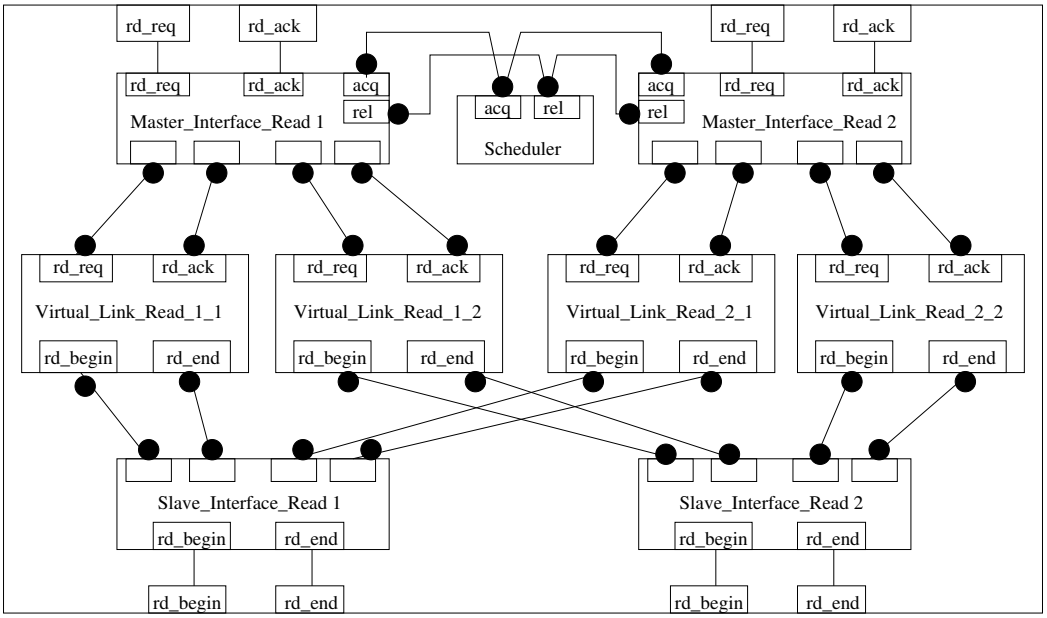

Figure 19: Bus architecture

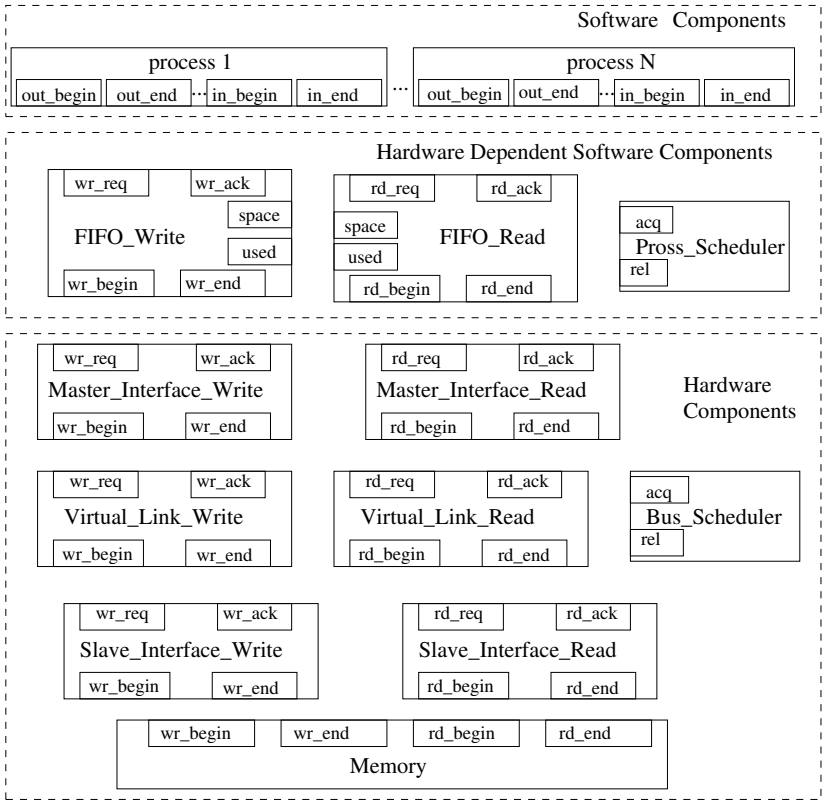

Figure 20: BIP atomic components Library

overall organization of the tool is shown in figure 21. It consists of two parts, the frontend translator and the backend transformation tool.

The frontend uses an open source C parser called codegen [1] to parse C files that describe the behavior of the DOL processes into an intermediate model. This, along with the structure of the process network extracted from the XML description, is transformed into the BIP application software model.

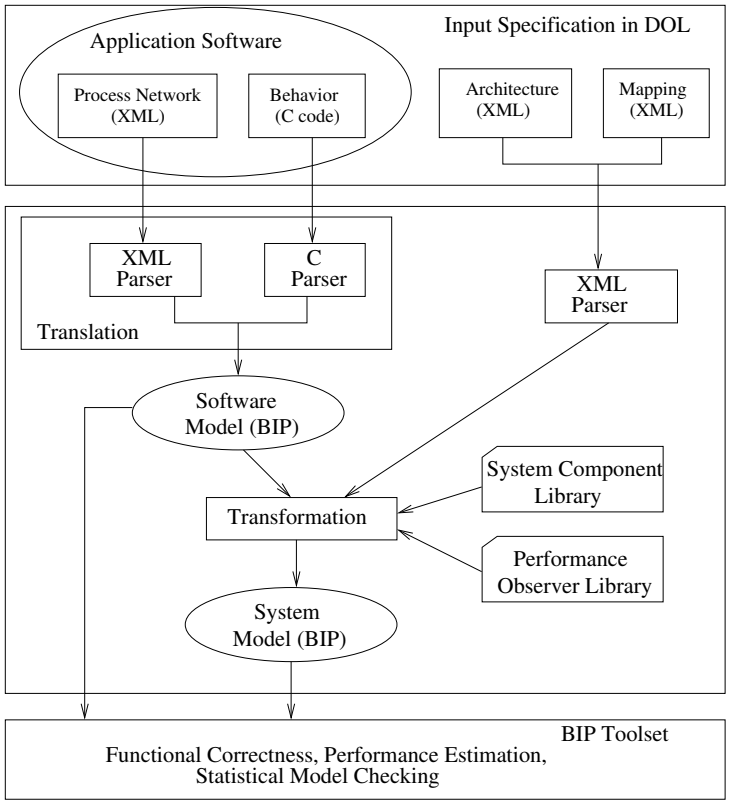

Figure 21: Tool Architecture

The frontend tool supports behavior described in C with a few restrictions, namely: 1) no use of global variable; 2) no *goto* statement; and 3) no call to the read/write primitives in recursive routines.

3.2 Performance Estimation on the System Model

Performance estimation of execution time is based on native simulation. The results are obtained dynamically by fine-granular code analysis. The basic idea is to take advantage of coverage tools to get the profiling result of C code during the simulation and then analyze the profiling time of each C statement. A target platform is characterized by a weight-table of instruction execution time. Computation time on a target architecture is obtained in two steps: 1) profiling of generate C code with profiling API and obtaining results by simulation; 2) analysis of C profiling results based on target architecture weight-table to obtain execution time estimation.

Bus and memory latency is measured using observer components.

3.3 MJPEG Decoder Case Study

We apply our method for generating an implementation of a MJPEG decoder on MPARM [4] architecture and provide performance results. The MJPEG decoder application software reads a sequence of MJPEG frames and displays the decompressed video frames. The process network of the application is illustrated in figure 22. It contains five processes $SplitStream$ (SS), $SplitFrame$ (SF), $IqzigzagIDCT$ ($IDCT$), $MergeFrame$ (MF) and $MergeStream$ (MS), and nine communication channels $C1$, ..., $C9$. The

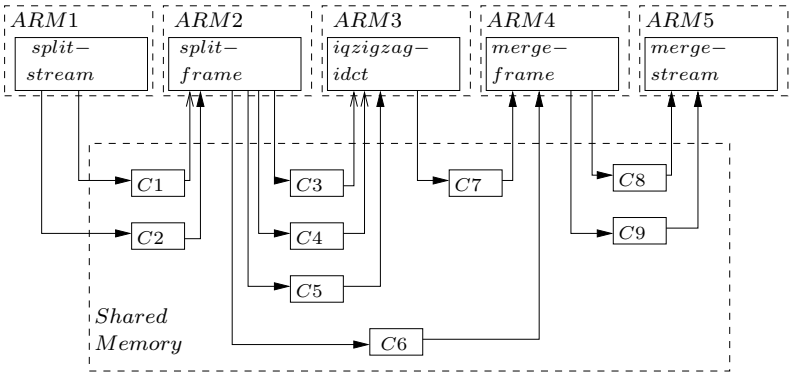

Figure 22: MJPEG Decoder application and a mapping

target architecture is a simplified MPARM, illustrated in figure 23. It is configured using five identical tiles and a shared memory, connected via a shared AMBA-AHB bus. Each tile contains a processor (ARM) connected to a local memory (LM) via a local bus. For the hardware model in BIP, we assumed all the

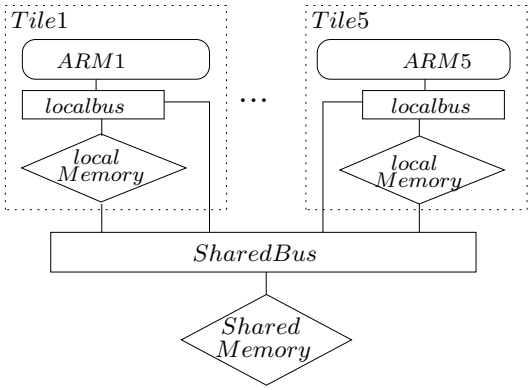

Figure 23: 5-Tile MPARM Architecture

local memory as SRAM with an access time of 2 cycles. The shared memory is a DRAM with an access time of 6 cycles. Clock frequency of all processors, memory and buses are assumed to be the same.

We experimented with eight different mapping to analyze their effect on the total computation and communication time for decoding a frame. The process mappings are described in table 1, and the sw_channel mappings are described in table 2.

For the mappings described above, a system model contains about 50 atomic components and 220 connectors, and consists of approximately 6K lines of BIP code, generating around 19.5K lines of C code. The total computation and communication delays for decoding a frame for different mappings are shown in figure 24. Mapping (1) produces the worst computation time as all processes are mapped to a single processor. Mapping (2) uses two processors, sill the performance does not improve much due to bad mapping. But (3) gives much better performance as the computation load is balanced. The other mappings can not produce better performance as the load can not be further distributed, even if more processors are used. The communication overhead is reduced if we map more channels to the local memories of the processors. The bus and memory access conflicts are shown in figure 25). As more channels are mapped to the local memory, the shared bus contention is reduced. However, this might increase the local memory contention, as is evident for (8).

The results show the feasibility of the system model for fine granular analysis of the effects of architecture and mapping constraints on the system behavior.

	$ARM1$	$ARM2$	$ARM3$	$ARM4$	$ARM5$
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

Table 1: Mapping Description of the processes

	$Shared$	$LM1$	$LM2$	$LM3$	$LM4$
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

Table 2: Mapping Description of the sw_channels

4 Conclusion

The presented method allows generation of a correct-by-construction model of a mixed hardware/software system from its application software, a description of the hardware architecture and a mapping. The method is completely automated and supported by tools. The system model is obtained by refining the application software model and composing it with the hardware architecture model. The composition is defined by the mapping.

Using BIP is instrumental for incremental construction of the models. Its expressiveness allows the integration of architecture constraints into the application model without suffering complexity explosion. DOL is used mainly as a front-end. Any other performance evaluation tool providing similar functionality could have been used.

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 mastering the complexity and appreciation of the impact of each parameter on system behavior.

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 that the method is tractable and allows design space exploration to determine optimal solutions.

Future work directions include extension to other programming models for the application software and richer hardware architecture models.

References

[1] http://think.ow2.org. 3.1

[2] Yasmina Abdeddaim, Eugene Asarin, and Oded Maler. Abstract scheduling with timed automata. 1

[3] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12, 2006. 1

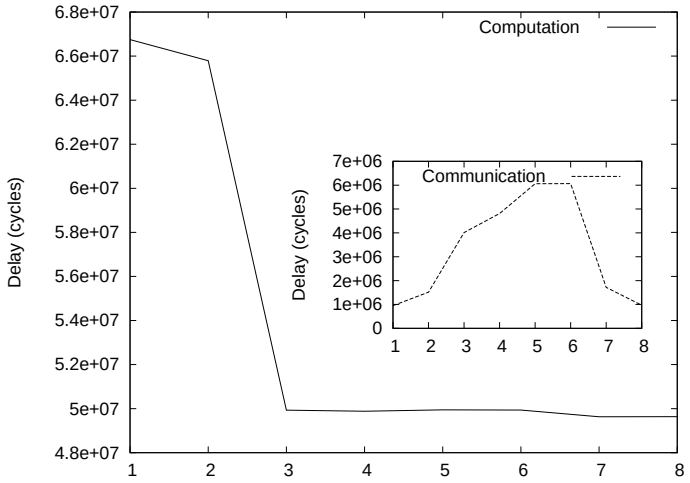

Figure 24: Performance Analysis

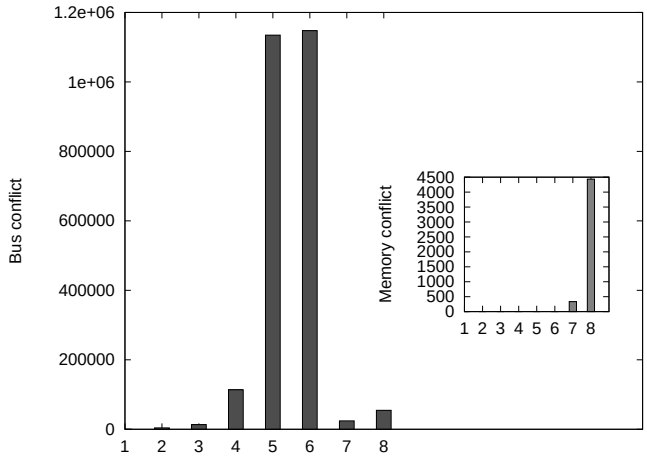

Figure 25: Communication Analysis

[4] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41:169–182, September 2005. 3.3

[5] S. Bliudze and J. Sifakis. A Notion of Glue Expressiveness for Component-Based Systems. In *Concurrency Theory CONCUR'08 Proceedings*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008. 2.1

[6] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. 1

[7] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. 1

[8] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis - the symta/s approach. 1

[9] Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using uppaal. 1

[10] Hermann Kopetz and Günther Bauer. The time-triggered architecture. 1

[11] Simon Künzli, Francesco Poletti, Luca Benini, and Lothar Thiele. Combining simulation and formal methods for system-level performance analysis. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 236–241, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. 1

[12] Imed Moussa, Thierry Grellier, and Giang Nguyen. Exploring sw performance using soc transaction-level modeling. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 20120, Washington, DC, USA, 2003. IEEE Computer Society. 1

[13] Ramzi Ben Salah, Marius Bozga, and Oded Maler. Compositional timing analysis. 1

[14] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *ACSD '07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, pages 29–40, Washington, DC, USA, 2007. IEEE Computer Society. 1, 2.2.1

[15] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. 1