From High-Level Modeling Towards Efficient and Trustworthy Circuits

SCHOLARONE™
Manuscripts

From High-Level Modeling Towards Efficient and Trustworthy Circuits

Mohamad Jaber, Mohamad Noureddine, and Fadi A. Zaraket

American University of Beirut, Beirut, Lebanon

{mj54,man17,fz11}@aub.edu.lb

Abstract. Behavior-Interaction-Priority (BIP) is a layered embedded system design and verification framework that provides separation of functionality, synchronization, and priority concerns to simplify system design and to establish correctness by construction. The framework comes with a runtime engine and a suite of verification tools that uses D-Finder and NuSMV as model checkers. In this paper we provide a method and a supporting tool that takes a BIP system and a set of invariants and computes a reduced sequential circuit with a system-specific scheduler and with a designated output that is true when the invariants hold. Our method uses ABC, a sequential circuit synthesis and verification framework to (1) generate an efficient FPGA implementation of the system, and to (2) verify the system and debug it in case a counterexample was found. Moreover we generate a concurrent C implementation of the circuit that can be directly used as a simulator. We evaluated our method with two large systems and our results outperform those possible with existing techniques.

1 Introduction

In recent years, *embedded systems* have witnessed a large expansion, especially with the emergence of automotive electronics, mobile and control devices. An embedded system is a composition of *heterogeneous* intellectual property (IP) components. Figure 1 shows a typical flow of the composition process where the components are specified as imperative programs, finite state machines (FSM), labeled transition systems (LTS), data flow networks, and discrete event based circuits. Computations in embedded systems are subject to several physical and architectural constraints that render the separation between software and hardware design impractical [23]. The partitioning task, often done manually, decides whether a component is to be implemented as a programmed process or as a realtime logic circuit. A plethora of software, behavioral, and logic compilation and synthesis techniques are used in the process [19].

The Behavior-Interaction-Priority (BIP) framework is a *Component-Based System* (CBS) design framework that uses a dedicated language and tool-set to support a rigorous and layered design flow for embedded systems. BIP allows to build complex systems by coordinating the behavior of a set of atomic components [5]. BIP makes use of (1) the DFinder [8] compositional and incremental verification tool-set, and
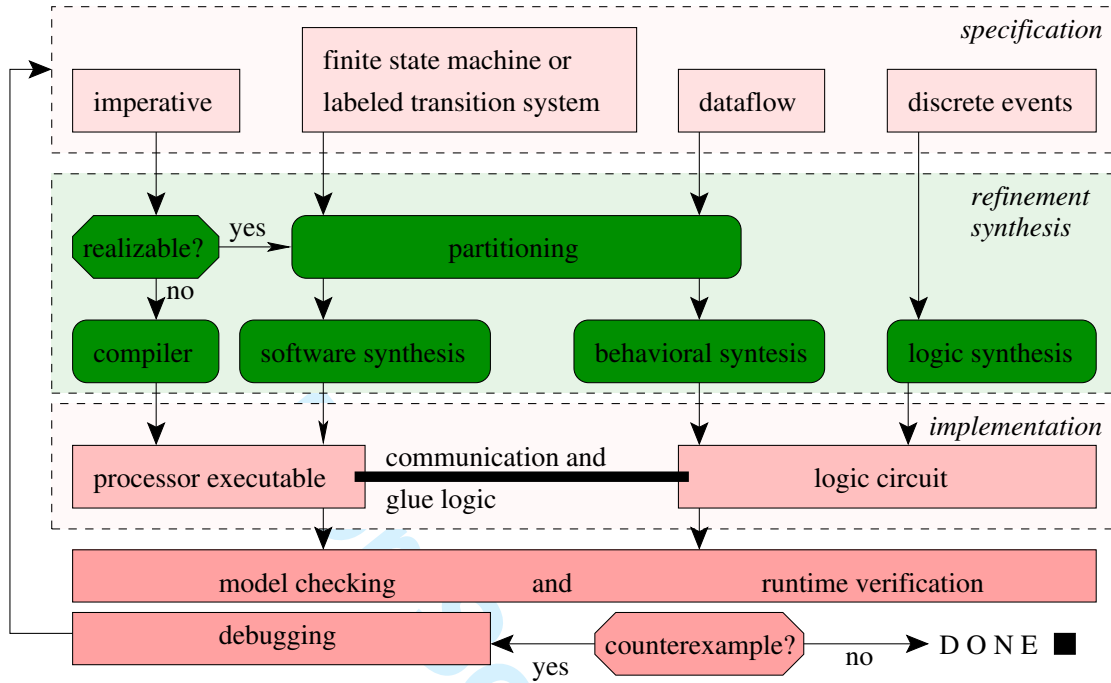
Fig. 1. Embedded system specification, refinement, and implementation stages

(2) the NuSMV [17] model checker, to check the correctness of BIP systems. However, DFinder [7] does not handle data transfer between components, and it does not support checking for invariants other than deadlock freedom. Additionally, for complex systems, NuSMV often suffers from the state space explosion problem [37], and fails to perform its verification tasks.

ABC [13] is a transformation-based verification (TBV) [26] framework that operates on And-Inverter Graphs (AIG); semi-canonical Boolean netlists with memory elements, and iteratively and synergistically employs powerful reduction, abstraction and decision algorithms such as retiming [26], redundancy removal [30, 27, 11, 1], logic rewriting [10], interpolation [29], and localization [39], symbolic model checking, bounded model checking, induction, interpolation, circuit SAT solving, and target enlargement [31, 32, 24, 6, 28].

In this paper, we present a method and a supporting tool ($\mathcal{B}ip\mathcal{SV}$) for embedded system synthesis, runtime verification, and model checking with a cycle based execution model. The method leverages transformation-based synthesis and verification techniques as follows.

1. The method takes a BIP system and a set of invariants and generates an AIG circuit with an output therein that is true iff the system is deadlock free, and satisfies the system invariants. The method

passes the generated AIG circuit to ABC for verification. ABC either proves correctness or produces a counter example where the system violates an invariant. This enabled us to find defects and prove systems that were not possible using DFinder and NuSMV.

2. The supporting tool $\mathcal{BipSV}$ provides a debugging mechanism where the counter example is mapped back to the original BIP system. The debugging tool is integrated with a wave form visualization tool [15].

3. The method generates an FPGA implementation of the BIP system with a system-specific execution framework. The FPGA implementation is passed to ABC synthesis reduction algorithms which reduce the area and the critical time of the FPGA implementation by removing latches and logic gates. To the best of our knowledge, we are the first to synthesize a BIP system directly into an FPGA.

4. The method generates a concurrent C implementation that simulates the BIP system with a system-specific execution framework.

BIP uses a runtime engine to simulate its execution semantic. The main loop of the engine consists of the following steps:

1. Each atomic component sends to the engine its current location.

2. The engine enumerates the list of interactions in the system, selects the enabled ones based on the current location of the atomic components and eliminates the ones with low priority.

3. The engine non-deterministically selects an interaction out of the enabled interactions.

4. Finally, the engine notifies the corresponding components and schedule their transitions for execution.

We differ in that, the system specific scheduler is a bit vector of interactions directly embedded in the implementation. The interaction bit vector evaluates in real-time and directly depends on the locations and the values of the variables of the input system. The system specific execution framework empirically reduces the space and time requirements for the C simulation and the FPGA execution.

Several frameworks for the design and verification of embedded systems exist. We briefly introduce them here and discuss and compare to them later in Section 7. Metropolis [2, 19] is a design framework that takes a Metropolis Meta Model (MMM) description of an embedded system and generates a SystemC [34] based simulator of the system. It has also a path to SIS [36] which is a synthesis predecessor tool of ABC, and a path to SPIN for model checking [25]. SystemC [34] in turn is a design framework based on C++ that allows system components to communicate through ports, interfaces, and channels. Extensions to SystemC such as ForSyDe [35] restrict the expressiveness to enable formal verification tools to handle the system. In

3

brief, our method supports the synthesis, model checking, and runtime verification concerns of embedded systems using tool independent semantics across the three concerns by embedding the execution model of the embedded system in the generated systems for each concern. This allows for simple debugging and design flow cycle iterations. Furthermore, the use of AIG circuits for synthesis and model checking allows our method to leverage the mature and rich literature of logic synthesis techniques.

The rest of this paper is organized as follows. In Section 2, we recall the necessary concepts of the BIP framework. Section 3 defines one loop program ($\mathcal{OLP}$). Section 4 formalizes sequential circuit and shows how to translate a sequential circuit into $\mathcal{OLP}$. Section 5 shows how to translate a BIP system into $\mathcal{OLP}$. Section 6 describes $\mathcal{BipSV}$, a full implementation of our framework and some benchmarks. Section 7 discusses related work. Section 8 draws some conclusions and perspectives.

2 BIP - Behavior Interaction Priority

We recall the necessary concepts of the BIP framework [5]. BIP allows to construct systems by superposing three layers of desin: Behavior, Interaction, and Priority. The *behavior* layer consists of a set of atomic components represented by transition systems. The *interaction* layer provides the collaboration between components. Interactions are described using sets of ports. The *priority* layer is used to specify scheduling policies applied to the interaction layer, given by a strict partial order on interactions.

2.1 Component-based Construction

BIP offers primitives and constructs for designing and composing complex behaviors from atomic components. Atomic components are Labeled Transition Systems (LTS) extended with C functions and data. Transitions are labeled with sets of communication ports. Composite components are obtained from atomic components by specifying interactions and priorities.

Atomic Components. An atomic component is endowed with a finite set of local variables X taking values in a domain Data. Atomic components synchronize and exchange data with each others through *ports*.

Definition 1 (Port). *A port $p[x_p]$, where $x_p \subseteq X$, is defined by a port identifier p and some data variables in a set x_p (referred to as the support set). We denote by $p.X$ the set of variables assigned to the port p, that is, x_p.*

4

Definition 2 (Atomic component). *An atomic component B is defined as a tuple $(P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$, where:*

- *(P, L, T) is an LTS over a set of ports P. L is a set of control locations and $T \subseteq L \times P \times L$ is a set of transitions.*
- *X is a set of variables.*
- *For each transition $\tau \in T$:*
    - *g_τ is a Boolean condition over X: the guard of τ,*
    - *$f_\tau = \{(x, f^x(X)) \mid x \in X\}$ where $(x, f^x(X)) \in f_\tau$ expresses the assignment statement $x := f^x(X)$ updating x with the value of the expression $f^x(X)$.*

For $\tau = (l, p, l') \in T$ a transition of the internal LTS, l (resp. l') is referred to as the source (resp. destination) location and p is a port through which an interaction with another component can take place. Moreover, a transition $\tau = (l, p, l') \in T$ in the internal LTS involves a transition in the atomic component of the form $(l, p, g_\tau, f_\tau, l')$ which can be executed only if the guard g_τ evaluates to $\texttt{true}$, and f_τ is a computation step: a set of assignments to local variables in X.

In the sequel we use the dot notation. Given a transition $\tau = (l, p, g_\tau, f_\tau, l')$, $\tau.src$, $\tau.port$, $\tau.guard$, $\tau.func$, and $\tau.dest$ denote l, p, g_τ, f_τ, and l', respectively. Also, the set of variables used in a transition is defined as $\varphi(f_\tau) = \{x \in X \mid x := f^x(X) \in f_\tau\}$. Given an atomic component B, $B.P$ denotes the set of ports of the atomic component B, $B.L$ denotes its set of locations, etc.

Given a set X of variables, we denote by $\mathbf{X}$ the set of valuations defined on X. Formally, $\mathbf{X} = \{\sigma : X \rightarrow \text{Data}\}$, where Data is the set of all values possibly taken by variables in X.

Semantics of Atomic Components. The semantics of an atomic component is an LTS over configurations and ports, formally defined as follows:

Definition 3 (Semantics of Atomic Components). *The semantics of the atomic component $B = (P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ is defined as the labeled transition system $S_B = (Q_B, P_B, T_B)$, where:*

- *$Q_B = L \times \mathbf{X}$, where $\mathbf{X}$ denotes the set of valuations on X,*
- *$P_B = P \times \mathbf{X}$ denotes the set of labels, that is, ports augmented with valuations of variables,*
- *T_B is the set of transitions defined as follows. $T_B = \{((l', v'), p(v_p), (l, v)) \in Q_B \times P_B \times Q_B \mid \exists \tau = (l', p[x_p], l) \in T : g_\tau(v') \wedge v = f_\tau(v'/v_p)\}$, where v_p is a valuation of the variables of p.*

5

A configuration is a pair $(l, v) \in Q_B$ where $l \in L$ is a control location, $v \in \mathbf{X}$ is a valuation of the variables in X. The evolution of configurations $(l', v') \xrightarrow{p(v_p)} (l, v)$, where v_p is a valuation of the variables attached to the port p, is possible if there exists a transition $(l', p[x_p], g_\tau, f_\tau, l)$, such that $g_\tau(v') = \mathtt{true}$. As a result, the valuation v' of variables is modified to $v = f_\tau(v'/v_p)$.

Creating composite components. Assuming some available atomic components $B_1, \ldots, B_n$, we show how to connect the components in the set $\{B_i\}_{i \in I}$ with $I \subseteq [1, n]$ using an *interaction*. An interaction a is used to specify the sets of ports that have to be jointly executed.

Definition 4 (Interaction). *An interaction a is a tuple (P_a, G_a, F_a), where:*

- $P_a \subseteq \cup_{i=1}^n B_i.P$ *is a nonempty set of ports that contains at most one port of every component, that is,*
  $\forall i : 1 \le i \le n : |B_i.P \cap P_a| \le 1$. *We denote by $X_a = \cup_{p \in P_a} p.X$ the set of variables available to a,*
- $G_a : \mathbf{X_a} \to \{\mathtt{true}, \mathtt{false}\}$ *is a guard,*
- $F_a : \mathbf{X_a} \to \mathbf{X_a}$ *is an update function.*

P_a is the set of connected ports called the support set of a. For each $i \in I$, x_i is a set of variables associated with the port p_i.

Definition 5 (Composite Component). *A composite component is defined from a set of available atomic components $\{B_i\}_{i \in I}$ and a set of interactions $\gamma = \{a_j\}_{j \in J}$. The connection of the components in $\{B_i\}_{i \in I}$ using the set γ of connectors is denoted by $\gamma(\{B_i\}_{i \in I})$.*

Definition 6 (Semantics of Composite Components). *A state q of a composite component $\gamma(\{B_1, \ldots, B_n\})$, where γ connects the B_i's for $i \in [1, n]$, is an n-tuple $q = (q_1, \ldots, q_n)$ where $q_i = (l_i, v_i)$ is a state of B_i. Thus, the semantics of $\gamma(\{B_1, \ldots, B_n\})$ is precisely defined as the labeled transition system $S = (Q, \gamma, \longrightarrow)$, where:*

- $Q = B_1.Q \times \ldots \times B_n.Q$,
- $\longrightarrow$ *is the least set of transitions satisfying the following rule:*

$$\frac{a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma \qquad G_a(\{v_{p_i}\}_{i \in I})}{\forall i \in I : q_i \xrightarrow{p_i(v_i)}_i q_i' \wedge v_i = F_a^i(\{v_{p_i}\}_{i \in I}) \qquad \forall i \notin I : q_i = q_i'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

6

where v_{p_i} denotes the valuation of the variables attached to the port p_i and F_a^i is the partial function derived from F_a restricted to the variables associated with p_i.

The meaning of the above rule is the following: if there exists an interaction a such that all its ports are enabled in the current state and its guard evaluates to `true`, then the interaction can be fired. When a is fired, all involved components evolve according to the interaction and uninvolved components remain in the same state.

Notice that several distinct interactions can be enabled at the same time, thus introducing non-determinism in the product behavior. One can add priorities to reduce non-determinism. In this case, one of the interactions with the highest priority is chosen non-deterministically.[1]

Definition 7 (Priority). *Let $S = (Q, \gamma, \longrightarrow)$ be the behavior of the composite component $\gamma(\{B_1, \ldots, B_n\})$. A priority model π is a strict partial order on the set of interactions A. Given a priority model π, we abbreviate $(a, a') \in \pi$ by $a \prec_\pi a'$ or $a \prec a'$ when clear from the context. Adding the priority model π over $\gamma(\{B_1, \ldots, B_n\})$ defines a new composite component $\pi\big(\gamma(\{B_1, \ldots, B_n\})\big)$ noted $\pi(S)$ and whose behavior is defined by $(Q, \gamma, \longrightarrow_\pi)$, where $\longrightarrow_\pi$ is the least set of transitions satisfying the following rule:*

$$\frac{q \xrightarrow{a} q' \qquad \neg\big(\exists a' \in A, \exists q'' \in Q : a \prec a' \wedge q \xrightarrow{a'} q''\big)}{q \xrightarrow{a}_\pi q'}$$

An interaction a is enabled in $\pi(S)$ whenever a is enabled in S and a is maximal according to π among the active interactions in S.

Finally, we consider systems defined as a parallel composition of components together with an initial state.

Definition 8 (System). *A BIP system $\mathcal{S}$ is a tuple $(B, Init, v)$ where B is a composite component, $Init \in B_1.L \times \ldots \times B_n.L$ is the initial state of B, and $v \in \mathbf{X^{Init}}$ where $X^{Init} \subseteq \cup_{i=1}^n B_i.X$.*

Given a port p from the system $\mathcal{S}$, we denote by (1) $interaction(p)$ to be the set of interactions that are connected to p; (2) $component(p)$ to be the component to which the port p belongs; (3) $transitions(p)$ to be the set of transitions labeled by p.

---

[1] The BIP engine implementing this semantics chooses one interaction at random, when faced with several enabled interactions.

7

3   One loop program ($\mathcal{OLP}$)

A *one loop program* ($\mathcal{OLP}$) ranges over boolean, integer and array variables. A variable can be either a register denoting a memory element or a wire denoting a functional macro. An $\mathcal{OLP}$ starts with the variable declarations followed by the wire variable definitions. Then memory variables are initialized simultaneously using the `do-together` construct. After initialization, an infinite loop keeps updating the value of the memory variables simultaneously. The listings in Figure 2 shows the syntax of an $\mathcal{OLP}$.

The wires are defined in a list of assignment statements `wiredef-list`. Each wire can be the target of at most one assignment statement. If a wire is not assigned then it is a non-deterministic *primary input* with a new non-deterministic value at each iteration of the loop.

The list of statements `init-list` assigns initial values to the register variables. All the assignment statements within `init-list` execute simultaneously as indicated with the `do-together` construct.

Similarly, the `next-list` list of statements updates the values of the register variables.

Each assignment statement has a left hand side target term which is either a variable or an access operator to an array element. The right hand side of an assignment is a combinational expression ranging over the program variables, Boolean and arithmetic operators, and a ternary choice operator. The ternary choice (`a? b : c`) returns b if a is `true` and c otherwise.

```
decl-list

wiredef-list

do-together {
  init-list
}

while(true) {
  do-together {
    next-list
  }
}
```

```
type: bool | int | bool [NUM] | int [NUM];
declaration: wire type id; | type id;

expr: term | uop expr| expr bop expr | expr ? expr : expr;
term: id | id[expr];

decl-list: declaration+
wiredef-list: (term = expr)*

init-list: (term = expr)*
next-list: (term = expr)*
```

Fig. 2. $\mathcal{OLP}$ Syntax

8

4 Sequential Circuit

The ABC synthesis and model checker reasons about And-Inverted-Graph representation of a sequential circuit.

Definition 9 (Sequential circuit). A *sequential circuit* is a tuple $((V, E), G, O)$. The pair (V, E) represents a directed graph on vertices V and edges $E \subseteq V \times V$ where E is a totally ordered relation. The function $G : V \mapsto types$ maps vertices to $types$. There are three disjoint types: *primary inputs*, *bit-registers* (which we often simply refer to as *registers*), and logical *gates*. Registers have designated *initial values*, as well as *next-state functions*. Gates describe logical functions such as the conjunction or disjunction of other vertices. A subset O of V is specified as the *primary outputs* of V. We will denote the set of primary input variables by I, and the set of bit-register variables by R.

Definition 10 (Fanins). We define the direct *fanin*s of a gate u to be $\{v \mid (v, u) \in E\}$ the set of source vertices connected to u in E. We call the *support* of u $\{v \mid (v \in I \lor v \in R) \land (v, u) \in *E\}$ all source vertices in R or I that are connected to u with $*E$, the transitive closure of E.

For the sequential circuit to be syntactically well-formed, vertices in I should have no fanins, vertices in R should have 2 fanins (the next-state function and the initial-value function of that register), and every cycle in the sequential circuit should contain at least one vertex from R. The initial-value functions of R shall have no registers in their support. All sequential circuits we consider will be well-formed.

The ABC analyzer reasons about And-Inverted-Graph (AIG) sequential circuits which are sequential circuits with only NAND gates restricted to have 2 fanins. Since NAND is functionally complete, this is not a limitation.

4.1 Semantics of sequential circuits

Definition 11 (State). A *state* is a Boolean valuation to vertices in R.

Definition 12 (Trace). A *trace* is a mapping $t : V \times \mathbb{N} \to \mathbb{B}$ that assigns a valuation to all vertices in V across time *steps* denoted as indexes from $\mathbb{N}$. The mapping must be consistent with E and G as follows. Term u_j denotes the source vertex of the j-th incoming edge to v, implying that $(u_j, v) \in E$. The value of

9

gate v at time i in trace t is denoted by $t(v, i)$.

$$
t(v, i) = \begin{cases}
s_v^i & : v \in I \text{ with sampled value } s_v^i \\
t(u_1, 0) & : v \in R, i = 0, u_1 := \text{ initial-state of } v \\
t(u_2, i - 1) & : v \in R, i > 0, u_2 := \text{ next-state of } v \\
G_v\big(t(u_1, i), ..., t(u_n, i)\big) & : v \text{ is a combinational gate with function } G_v
\end{cases}
$$

The semantics of a sequential circuit are defined with respect to semantical traces. Given an input valuation sequence and an initial state, the resulting trace is a sequence of Boolean valuations to all vertices in V which is consistent with the Boolean functions of the gates. We will refer to the transition from one valuation to the next as a *step*. A node in the circuit is justifiable if there is an input sequence which when applied to an initial state will result in that node taking value true. A node in the circuit is valid if its negation is not justifiable. We will refer to targets and invariants in the circuit; these are simply vertices in the circuit whose justifiability and validity is of interest respectively. A sequential circuit can naturally be associated with a finite state machine (FSM), which is a graph on the states. However, the circuit is very different from its FSM; among other differences, it is exponentially more succinct in almost all cases of interest [14].

4.2 Translation from $\mathcal{OLP}$ to AIG circuits

The translation of an $\mathcal{OLP}$ into an AIG circuit first constructs registers, wires, and primary input variables for each $\mathcal{OLP}$ variable. It then recursively traverses the right hand side of the assignment statements in `wiredef-list`, `init-list`, and `next-list` to build corresponding combinational circuits. It connects the outputs of the combinational circuits built for the `init-list` and `next-list` assignment expressions to the initial value and next state value input pins of the corresponding registers, respectively. Finally, it connects the outputs of the combinational circuits built for the `wiredef-list` to the wires referring to the variables declared as wire variables in `decl-list`.

Variables. We consider each variable not declared as a wire in `decl-list`. We instantiate a corresponding vector of AIG registers with an adequate bit width. The width of the bit vector can be selected by the user, or can be set to match the default width of the declared type. Typically the default values for the bit width are 32 bits for an integer, one bit for a Boolean, and a finite two dimensional bit vector for an array. In our case, and for $\mathcal{OLP}$ programs generated from BIP systems, we will not have arrays of register variables and we will

10

only have fixed size arrays of Boolean wires as discussed in Section . We consider variables declared as wires in `decl-list` and that do not have a corresponding assignment statement in `wiredef-list` as non-deterministic. For each non-deterministic variable we instantiate a corresponding vector of primary inputs with an adequate bit-width. We consider variables declared as wires in `decl-list` with a corresponding assignment statement in `wiredef-list` as functional macros. For each functional macro we instantiate a vector of identity gates (a sequence of two negation gates) where the fanouts correspond to the wire variable and the fanins correspond to the expression defining the wire variable in `wiredef-list`. We denote the gates corresponding to each variable by the function `vargates`.

```
variables(decl−list, wiredef−list)
  foreach variable v in decl−list
    if (v is not a wire)
      vargates(v) = instantiate-registers(v,type(v))
    elseif (v is not assigned in wiredef−list)
      vargates(v) = instantitate-primary-inputs(v,type(v))
    else
      vargates(v) = instantiate-identity-gates(v,type(v))
    endif
  endfor
```

Assignment statements. We consider each assignment statement in `wiredef-list`, `init-list`, and `next-list` and traverse the right hand side expressions of each assignment with the `traverse` routine. The traversal of an expression runs recursively. If the expression refers to a variable v (base case), then the traversal returns `vargates(v)`. If the expression is a logical, conditional, or arithmetic expression, then the `library` routine looks it up in a complete table of circuits with the adequate bit width. For example, if the expression is a ternary conditional statement of the form $b?$ e_1 : e_2, then `library` instantiates a multiplexer, connects its two data fanins to the nodes corresponding to e_1 and e_2, connects its control fanins to the nodes corresponding to b, and returns its fanouts.

```
traverse(exp)

  if (exp is a variable)
    return vargates(exp)
  endif

  foreach i[1 .. exp.operands.size()]
    wirevec[i] = traverse(exp.operands[i])
  endfor
```

11

```
    return library(exp.operation, wirevec)
```

Connections. Finally, we connect the nodes corresponding to the right hand side expressions of the assignment statements in the `init-list` and `next-list` expressions to the initial value and next value fanins of the corresponding register gates, respectively. We connect the nodes corresponding to the right hand side expressions of the assignment statements in the `wiredef-list` expressions to the fanins of the corresponding wire identity gates.

5   BIP to $\mathcal{OLP}$

Given a BIP system $\mathcal{S} = (B, Init, v)$, $\mathcal{BipSV}$ calls function BIP-to-OLP to translate $\mathcal{S}$ into an $\mathcal{OLP}$ program with its own customized execution engine. It calls four functions that fill `decl-list`, `wiredef-list`, `init-list`, `next-list`. All these function use the `append` call to add code fragments to lists.

```
BIP-to-OLP(B, Init, v)
  generateDeclartionList()
  generateWireDefList()
  generateInitList()
  generateNextList()
```

Function `generateDeclartionList()` fills `decl-list` as follows. It creates three arrays of wires to denote interaction semantics. Array ie elements denote whether all logical constraints except priority rules are met for a given interaction. Array ip elements denote whether a given interaction is enabled after applying priority rules. Array is elements denote whether an enabled interaction is selected for execution. Currently, one interaction is selected to avoid executing conflicting interactions. Two interactions are conflicting if they involve same components. The *selector* wire is a non-deterministic primary input that is used to select one of the enabled interactions. The *cycle* boolean register is used to denote whether the system is executing actions corresponding to either interaction or transition. The function also declares two wires ($B_i.p_j.e$ and $B_i.p_j.s$) for each port p_j. Wire $B_i.p_j.e$ denotes whether the port is enabled and wire $B_i.p_j.s$ denotes whether the port is selected by the interaction for execution. Moreover, for each component B_i the function declares a register variable $B_i.\ell$ denoting the current location of B_i. Similarly, the function declares a variable register $B_i.x_j$ for each variable x_j in component B_i.

12

```
generateDeclartionList()
  // interaction enablement wires
  append wire bool ie[|J|] to decl−list
  // interaction priority wires
  append wire bool ip[|J|] to decl−list
  // interaction selected wires
  append wire bool is[|J|] to decl−list
  // non-deterministic priority selector wire
  append wire int selector to decl−list
  // cycle denotes transition or interaction mode
  append bool cycle to decl−list


  foreach i ∈ [1..|I|]
    foreach j ∈ [1..|B_i.P|]
      // port enablement
      append wire bool B_i.p_j.e to decl−list
      // port selected
      append wire bool B_i.p_j.s to decl−list
    endfor


    // location registers
    append int B_i.ℓ to decl−list


    foreach j ∈ [1..|B_i.X|]
      // variable registers
      append int B_i.x_j to decl−list
    endfor
  endfor
```

Function `generateWireDefList()` fills `wiredef-list` with functional macro definitions as follows. The enable wire $B_i.p_j.e$ is `true` when there exists a transition τ labeled with port p, its source ($\tau.src$) is the current location ($B_i.\ell$), and its guard is `true`.

Array element $ie[j]$ corresponding to interaction a_j is evaluated to `true` when the guard of a_j is `true` and all its ports are enabled. Array element $ip[j]$ is evaluated to `true` when $ie[j]$ is `true` and a_j has higher priority than other enabled interactions. Array element $is[j]$ is evaluated to `true` when $ip[j]$ is true either a_j is selected ($selector$ equals to j) or the selected interaction is not enabled and j is the first enabled interaction greater with an index greater than j. The use of non-deterministic selector is added for fairness. The selected wire $B_i.p_j.s$ is `true` when there exists a selected interaction a_k (i.e., $is[k]$ is `true`) involving $B_i.p_j$.

13

```
generateWireDefList()
  // iterate over components
  foreach i ∈ [1..|I|]
    // iterate over component ports
    foreach j ∈ [1..|B_i.P|]
      append B_i.p_j.e := ⋁_{τ∈transitions(B_i.p_j)} τ.guard ∧ B_i.ℓ = τ.src to wiredef−list
    endfor
  endfor

  // iterate over interactions
  foreach j ∈ [1..|J|]
    append ie[j] := a_j.guard ∧ ⋀_{p∈a_i.P} component(p).p.e to wiredef−list
    append ip[j] := ie[j] ∧ (∀k ≠ j : ie[k] ⇒ a_k < a_j) to wiredef−list
    append is[j] := ip[j] ∧ (selector = j ∨ (¬ip[selector] ∧ ∀k > j : ¬ip[k]) to wiredef−list
  endfor

  // iterate over components
  foreach i ∈ [1..|I|]
    // iterate over component ports
    foreach j ∈ [1..|B_i.P|]
      append B_i.p_j.s := ⋁_{a_k∈interactions(B_i.p_j)} is[k] to wiredef−list
    endfor
  endfor
```

Function generateInitList() fills init-list with initial value definitions taken from $Init$ for location variables ($B_i.\ell$) and v for component variables ($B_i.x_j$). Register variable $cycle$ is initialized to zero to denote an interaction execution mode.

```
generateInitList()
  // initialize to interaction mode
  append cycle := 0 to init−list
  foreach i ∈ [1..|I|]
    append B_i.ℓ := Init.B_i to init−list
    foreach j ∈ [1..|B_i.X|]
      // v is the initial valuation
      append B_i.x_j := v(B_i.x_j) to init−list
    endfor
  endfor
```

14

Function `generateNextList()` fills `next-list` with the next state value definitions of register variables. Each component variable can be modified either in an interaction action or in a transition action. The value of variable $cycle$ makes this distinction.

In the interaction mode ($cycle$ equals to zero), the function considers each assignment statement σ from the action of interaction a_j. The function appends a conditional clause requiring the a_k to be selected for execution so that the target variable $B_i.x_j$ of σ is assigned to the expression of σ ($\sigma.expr$). The sequence of conditional clauses form a nested ternary conditional expressions where the last expression retains the previous value of the variable.

Similarly, in the transition mode ($cycle$ equals to one), the function considers each assignment statement σ from the action of transition τ. The function appends a conditional clause requiring the port of the transition τ to be selected for execution and the location of the component to be equal to the source of the transition. The target variable $B_i.x_j$ of σ is assigned to the expression of σ ($\sigma.expr$).

In the transition mode, the function considers the current location of each component $B_i.\ell$ and appends a conditional clause requiring the transition source to be equal to the current location and the port of the transition to be selected. The expression corresponding to the conditional clause updates the current location to be the destination of the transition ($\tau.dest$). In the interaction mode, the location retains its value. Finally, the $cycle$ variable is toggled.

```
generateNextList()
  // iterate over components - interaction-mode
  foreach  i ∈ [1..|I|]
    // iterate over variables, where B_i.X = {x_1,...,l_{|B_i.X|}}
    foreach  j ∈ [1..|B_i.X|]
      // interaction mode
      append  B_i.x_j := cycle = 0?  to  var−st
      // iterate over interactions
      foreach  k ∈ [1..|J|]
        // iterate over interaction assignments
        foreach  σ ∈ a_k.action
          if (B_i.x_j = σ.term)
            append  is[k]? σ.expr :  to  var−st
          endif
        endfor
      endfor
      // interaction mode and no data transfer for B_i.x_j
      append  B_i.x_j :  to  var−st
```

```
      // iterate over component transitions - transition-mode
      append  B_i.ℓ := cycle = 0? B_i.ℓ :  to  loc−st
      foreach  τ ∈ B_i.T
        // iterate over transition assignments
        foreach  σ ∈ τ.action
          if (B_i.x_j = σ.term)
              append  (B_i.port(τ).s ∧ τ.src = B_i.ℓ)? σ.expr :  to  var−st
          endif
        endfor
        append  (B_i.port(τ).s ∧ τ.src = B_i.ℓ)? τ.dest :  to  loc−st
      endfor

      append  B_i.x_j  to  var−st
      append  var−st  to   next−list

      append  B_i.ℓ  to  loc−st
      append  loc−st  to   next−list

    endfor
    // switch cycle
    append  cycle := ¬cycle  to  next−list
  endfor
```

5.1 Illustrative Example

Figure 3 shows a traffic light controller system modeled in BIP. It is composed of two atomic components, `timer` and `light`. The timer counts the amount of time for which the light must stay in a specific state (i.e. a specific color of the light). The light component determines the color of the traffic light. Additionally, it informs the timer about the amount of time to spend in each location through a data transfer on the interaction between the two components.

Figure 4 shows $\mathcal{OLP}$ code generated that corresponds to the to the traffic light controller BIP system shown in Figure 3.

5.2 One cycle optimization

Recall that an interaction specifies a strong synchronization among its involved components. Data transfer can take place during such synchronization. The operational semantic of BIP requires to (1) first execute
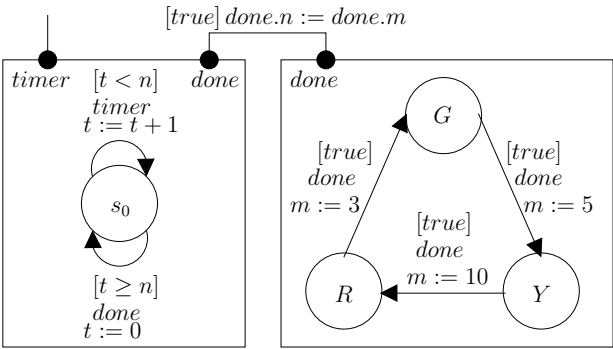
16

Fig. 3. Traffic light in BIP

data transfer of the selected interaction and then (2) execute the functions of the corresponding transitions of atomic components. For this, in the above translation, we used *cycle* boolean register to denote whether the system is executing actions corresponding to either interaction or transition. However, in some cases data transfers of all interactions modify some variables that are not assigned in the corresponding transitions of those interactions. This can be detected by doing a static data dependency between interactions and their transitions. This may drastically improve the performance of the system since data transfers as well as functions of transitions may be executed in one cycle. Note that, our implementation supports this optimization.

6   Implementation and Evaluation

6.1 $\mathcal{B}ip\mathcal{SV}$

$\mathcal{B}ip\mathcal{SV}$ (BIP Synthesis Verification) is a Java implementation of the translation from BIP to $\mathcal{OLP}$ described in Section 5, and, is part of the BIP distribution [38].

$\mathcal{B}ip\mathcal{SV}$ takes as input a BIP system and a set of invariants and generates the corresponding $\mathcal{OLP}$ program with a system-specific execution framework. The $\mathcal{OLP}$ program can be directly compiled for runtime verification (simulation) where primary inputs are set to random values at each iteration. The resulting binary can be concurrent by replacing the `do-together` construct with the corresponding openmp directives.

For sequential synthesis, we synthesize an AIG circuit that can be used as an FPGA implementation of the system and pass it to ABC. We use ABC synthesis and reduction algorithms to reduce the area and the critical time of the AIG circuit by removing latches and logic gates using techniques such as retiming [26],

17

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

```
/*** decl-List ***/
int timer.t;
int timer.n;
int light.m;
int timer.ℓ;
int light.ℓ;
bool cycle;

wire int selector;
wire bool timer.timer.e;
wire bool timer.timer.s;
wire bool timer.done.e;
wire bool timer.done.s;
wire bool light.done.e;
wire bool light.done.s;
wire bool ie[2];
wire bool ip[2];
wire bool is[2];
```

```
/*** wiredef-list ***/
timer.timer.e = (0 == timer.ℓ) && (timer.t < timer.n);

timer.done.e  = (0 == timer.ℓ) && (timer.t == timer.n);
light.done.e  = (0 == light.ℓ) || (1 == light.ℓ) || (2 == light.ℓ);

ie[0] = timer.timer.e;
ie[1] = (light.done.e && timer.done.e);

ip[0] = ie[0];
ip[1] = ie[1];

is[0] = (ip[0] && ( selector == 0 || (!ip[selector]  && !ip[1]);
is[1] = (ip[1] && ( selector == 1 || (!ip[selector]);

timer.timer.e = is[0];
timer.done.e  = is[1];
light.done.e  = is[1] ;
```

```
do-together {
  /*** init-list ***/
  timer.t = 0;
  timer.n = 10;
  timer.ℓ = 0;

  light.m = 5;
  light.ℓ = 0;

  cycle = true;
}/* end do-together */
```

```
while(true) {
  do-together {
    /*** next-list ***/
    timer.n = cycle? is[1]? light.m : timer.n : timer.n;

    timer.ℓ = cycle? timer.ℓ: timer.timer.e && timer.ℓ == 0? 0 : timer.
        timer.e && timer.ℓ == 0? 0 : timer.ℓ;

    timer.t = cycle? timer.t : timer.ℓ == 0 && timer.timer.e? (timer.t +
        1) : timer.ℓ == 0 && timer.done.e? 0 :
            timer.t;

    light.ℓ = cycle? light.ℓ : light.ℓ == 2 && light.done.e? 0: light.ℓ ==
        1 && light.done.e? 0 : light.ℓ == 0
            && light.done.e? 1 : light.ℓ;

    light.m = cycle? light.m : light.ℓ == 0 && light.done.e? 3: light.ℓ ==
        1 && light.done.e? 10: light.ℓ == 2
            && light.done.e? 5 : light.m;

    cycle = !cycle;
  } /*end do-together*/
} /*end while(true)*/
```

Fig. 4. Sample of $\mathcal{OLP}$ generated code of traffic light system

18

redundancy removal [30, 27, 11, 1], logic rewriting [10], interpolation [29], and localization [39]. The reduced AIG circuit is equivalent to the original circuit and can be used as a reduced FPGA implementation.

For verification, ABC uses the sequential synthesis techniques above to reduce the AIG circuit and render it amenable for decision algorithms. Then ABC uses decision algorithms such as symbolic model checking, bounded model checking, induction, interpolation, circuit SAT solving, and target enlargement [31, 32, 24, 6, 28] to verify the correctness of the circuit with respect to the BIP system invariants. It either proves correctness or produces a counter example where the system violates the property. $\mathcal{B}ip\mathcal{SV}$ provides a debugging mechanism where the counter example is mapped back to the original BIP system. The debugging tool is integrated with wave form visualization tool [15].

$\mathcal{B}ip\mathcal{SV}$ is equipped with a command line interface that accepts a set of configuration options. It takes the name of the input BIP file and optional flags.

```
java -jar bip-to-abc.jar [options] input.bip output.abc [property.txt]
```

We evaluated $\mathcal{B}ip\mathcal{SV}$ against two industrial benchmarks, an *Automatic Teller Machine* (ATM) [16] and the *Quorum* consensus protocol [22]. We report on the size of the generated AIGs before and after reduction, and on the time taken by the ABC solver to reduce and verify the benchmarks. We compare the results for the verification of the ATM benchmark with the NuSMV [17] model checker.

6.2   The ATM benchmark

Automatic Teller Machine (ATM) is a computerized system that provides financial services for users in a public space. Figure 5 shows a structured BIP model of an ATM system adapted from the description provided in [16]. The system is composed of four atomic components: (1) the User (2) the ATM (3) the Bank Validation and (4) the Bank Transaction. ATM component handles all interactions between the users and the bank. No communication between the users and the bank is allowed.

The ATM starts from an idle location and waits for the user to insert the card and enter the confidential code. The user has 5 time units to enter the code before the counter expires and the card is ejected by the ATM. Once the code is entered, the ATM checks with the bank validation unit for the correctness of the code. If the code is invalid, the card is ejected and no transaction occurs. If the code is valid, the ATM waits for the user to enter the desired amount of money for the transaction. The time-out for entering the amount of money is of 6 time units.
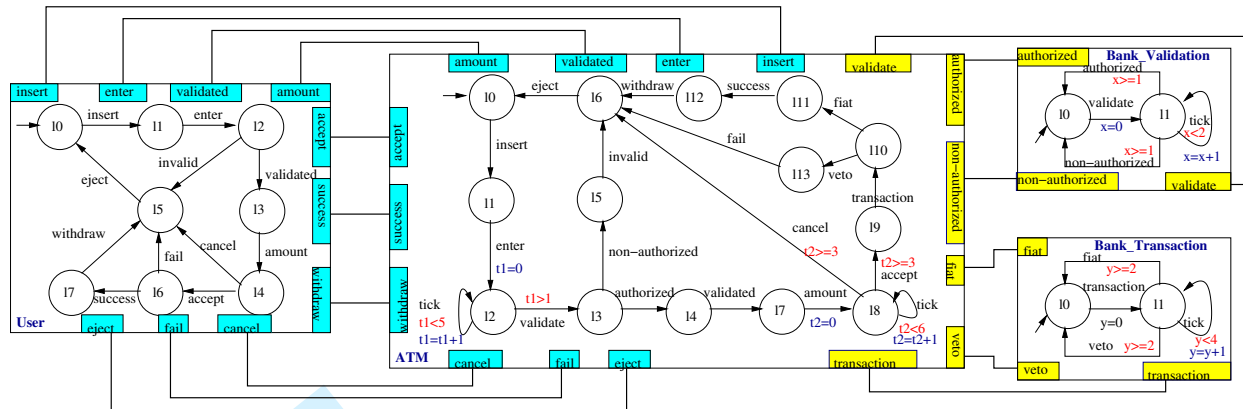
19

Fig. 5. Modeling of ATM system in BIP

	Original			After reduction			Time(s)	
ATMs	latches	AND-gates	levels	latches	AND-gates	levels	$\mathcal{B}ip\mathcal{SV}$	NuSMV
2	78	2308	125	37	552	25	26.1	1.4
3	102	3689	197	50	804	29	32.65	142.6
4	146	5669	234	63	1036	29	597	3361

Table 1. ATM results

Once the user enters the desired transaction amount, the ATM checks with the bank whether the transaction is allowed or not by communicating with the bank transaction unit. If the transaction is approved, the money is transferred to the user and the card is ejected. If the transaction is rejected, the user is notified and the card is ejected. In all cases, the ATM goes back to the idle location waiting for any additional users. In our model, we assume the presence of a single bank and multiple ATMs and users.

Table 1 shows the improvement obtained by using $\mathcal{B}ip\mathcal{SV}$ to verify the deadlock freedom of the ATM system, as compared to using the NuSMV model checker [17]. The first column of the table shows the number of clients and ATMs in the system. The table contains the number of latches, AND gates and logic levels in the AIG generated by $\mathcal{B}ip\mathcal{SV}$ before and after applying reduction techniques, respectively. We report on the verification time taken by the ABC solver to check the generated AIG, and the total taken to perform both synthesis (reduction) and verification, in addition to the time taken by NuSMV to perform verification.

With the increase in the number of users and ATMs in the system, $\mathcal{B}ip\mathcal{SV}$ outperforms NuSMV in terms of total verification time, reaching a speedup of 5.6 for 4 users and ATMs. Additionally, $\mathcal{B}ip\mathcal{SV}$ allows developers to make use of several reduction techniques that are able to reach an average of 50% reduction in the size of the AIG. Note that for 2 ATMs and users, NuSMV outperforms $\mathcal{B}ip\mathcal{SV}$. This is due to the fact

20

that when performing verification, ABC tries multiple verification and reduction algorithms before reaching a conclusive result. However, the advantage that $\mathcal{B}ip\mathcal{SV}$ presents can be clearly seen for larger number of ATMs and users.

6.3   The Quorum protocol

The *Quorum* protocol is a consensus protocol proposed in [22] as complementary to the Paxos consensus protocol [21] under perfect channel conditions. *Consensus* allows a set of communicating processes (clients and servers in our case) to agree on a common value. Each of clients proposes a value and receives a common decision value. The authors in [22] propose to use Quorum when no failures occur (perfect channel conditions) and Paxos when less than half of the servers may fail.

The Quorum protocol operates as follows.

1. Upon proposal, a client c broadcasts its proposed value v to all servers. It also saves v in its local memory and starts a local time t_c.

2. When a server receives a value v from a client c, it performs the following check.

   – It if has not sent any accept messages, it sends an accept message $accept(v)$ to the client c.

   – If it has already accepted value v', it sends an accept message $accept(v')$ to the client c.

3. If a client c receives two different accept messages, it switches to the backup phase $switch - backup(proposal_c)$.

4. If a client c receives the same accept messages $accept(v)$ from all the servers, it decides on the value v.

5. If a client's timer t_c expires, it waits for at least one accept message $accept(v')$ from a server, or chooses a value v' from an already received $accept(v')$ message, and then switches to the backup phase with the value v'.

6. The *backup* phase is an implementation of the Paxos algorithm. Quorum in this case has decided that the channel is not perfect.

We implemented the Quorum protocol in BIP, and we used $\mathcal{B}ip\mathcal{SV}$ to verify two invariants as defined in [22].

1. $Invariant_1$: If a client c decides on a value v, then all clients $c' \neq c$ that have switched, either before or after c, switch with the value v.

2. $Invariant_2$: If a client c decides on a value v, then all clients $c' \neq c$ who decide, do so with the same value v.

21

Design	Original			After reduction			Time (s)	
	latches	AND-gates	levels	latches	AND-gates	levels	$\mathcal{B}ip\mathcal{SV}$	NuSMV
2-2-e	264	3508	101	65	923	51	0.78	526
2-2-v	264	3614	105	66	641	29	240.6	526
4-2-e	390	6305	145	117	1129	50	0.24	memory-out
4-2-v	390	6453	151	117	1170	30	58 hours	memory-out

Table 2. Quorum results

Fig. 6. Visualization of a counter example using Gtkwave

Table 2 shows the results of using $\mathcal{B}ip\mathcal{SV}$ to verify the Quorum protocol for 2 and 4 clients with 2 servers. The designs are indexed as `num_clients-num_servers-status` where `num_clients` is the number of clients, `num_servers` is the number of servers and `status` is either valid (v) or erroneous (e). A valid design contains no design bugs, while an erroneous design is injected with a bug. We report on the size of the AIG in terms of number of latches, number of AND gates and logic levels before and after applying reduction algorithms.

Using ABC's synthesis and reduction algorithms, $\mathcal{B}ip\mathcal{SV}$ was able to reduce the size of the generated AIGs for all designs by a factor larger than 50%. Furthermore, $\mathcal{B}ip\mathcal{SV}$ was able to give conclusive results about all four designs, unlike NuSMV which failed to give any decision about the designs having 4 clients and 2 servers. For example, $\mathcal{B}ip\mathcal{SV}$ found a counter example for the erroneous design having 4 clients and 2 servers in 0.24 sec while NuSMV failed to do so. Figure 6 shows a snippet of the generated counter example for the erroneous design, visualized using the Gtkwave [15] waveform viewer. The variables presented in the counterexample are the current control locations and the value of the variables of the different components in the design. Red arrows points to the values that implies a violation of the invariant.

22

7   Related work

The overlap between software and hardware design in embedded systems creates more challenges for verification and code generation.

SystemC [34] is a modeling platform based on C++ that provides design abstractions at the *Register Transfer Level* (RTL), behavior, and system levels. It aims at providing a common design environment for embedded system design and hardware-software co-design. SystemC designers write their systems in C++ using SystemC class libraries that provide implementations for hardware specific objects such as concurrent modules, synchronization constructs, and clocks. Therefore the input systems can be compiled using standard C++ compilers to generate binaries for simulation. SystemC allows for the communication between different components of a system through the usage of ports, interfaces and channels.

Metropolis [2, 19] is an embedded system design platform based on formal modeling and separation of concerns for an effective design process. A Metropolis process is a sequence of events representing functionality, and different processes communicate via ports of interfaces. An interface includes methods that processes can use to communicate. Metropolis uses SIS for synthesis, SystemC and Ptolemey for runtime verification, and SPIN for model checking. While BIP separates behavior from interaction (synchronization and communication) to simplify correctness by construction and compositional verification, Metropolis separates communication from behavior (computation) and leaves synchronization highly coupled within each of them.

The BIP framework differs from SystemC in that it presents a dedicated language and supporting toolset that describes the behavior of individual system components as symbolic LTS. Communication between components in BIP is ensured through ports and interactions. BIP operates at a higher level than SystemC and does not provide support for circuit level constructs.

Verification techniques for SystemC and BIP make use of symbolic model checking tools. NuSMV2 [17] is a symbolic model checker that employs both SAT and BDD based model checking techniques. It processes an input describing the logical system design as a finite state machine, and a set of specifications expressed in LTL, Computational Tree Logic (CTL) and Property Specification Language (PSL). Given a system $\mathcal{S}$ and a set of specifications P, NuSMV2 first flattens $\mathcal{S}$ and P by resolving all module instantiations and creating modules and processes, thus generating one synchronous design. It then performs a Boolean encoding step to eliminate all scalar variables, arithmetic and set operations and thus encode them as Boolean functions.

23

The work in [33] uses constraint based programming to compute an executable MPI based parallel simulator of an embedded and cyber-physical system written in ForSyDe [35]. ForSyDe is a library of SystemC based parametrized system components with strict constraint specifications and a blocking write FIFO queue modeling a Kan network. The instants of the ForSyDe components are processes that communicate only through signals.

The work in [3] introduces a model checking methodology for LTL specifications of embedded system written in DIVINE [4] over a total store order (TSO) of memory elements. Our method assumes a similarly relaxed memory model since it adopts a cycle based execution model where updated memory values are observable at the next cycle.

In order to avoid the state space explosion problem, NuSMV2 performs a cone of influence reduction [9] step in order to eliminate non-needed parts of the flattened model and specifications. The cone of influence reduction abstraction technique aims at simplifying the model in hand by only referring to variables that are of interest to the verification procedure, i.e. variables that influence the specifications to check [18].

DFinder [8] is an automated verification tool for checking invariants on systems described in the BIP language. Given a BIP system $\mathcal{S}$ and an invariant $\mathcal{I}$, DFinder operates compositionally and iteratively to compute invariants $\mathcal{X}$ of the interactions and the atomic components of $\mathcal{S}$. It then uses the Yices *Satisfiability Modulo Theory* (SMT) solver [20] to check for the validity of the formula $\mathcal{X} \wedge \neg \mathcal{I} = false$. Additionally, DFinder checks the deadlock freedom of $\mathcal{S}$ by building an invariant $\mathcal{I}_d$ that represents the states of of $\mathcal{S}$ in which no interactions are enabled, i.e., a deadlock occurs. It then checks the for the formula $\mathcal{X} \wedge \mathcal{I}_d = false$, i.e., none of the deadlock states are reachable in $\mathcal{S}$.

Techniques based on symbolic model checking for the verification of BIP designs suffer from the state space explosion problem, and often fail to scale with the size and the complexity of the systems. On the other hand, DFinder does not handle data transfer between atomic components, thus limiting the range of practical applications on which it can be applied. Our technique handles data transfers and uses the wide range of synthesis and reduction algorithms provided by ABC to effectively reduce the size and the complexity of the verification problem. Most of these algorithms have no counterpart in symbolic model checking.

Unlike all the methods described above, our method leverages the same semantics for FPGA synthesis, model checking, and runtime verification (simulation).

24

8 Conclusion and Future Work

In this paper we present a method for embedded system synthesis, runtime verification, and model checking with supporting tools for the BIP framework. The method takes a BIP system and generates a concurrent C program with a system specific scheduler embedded therein. The concurrent C program serves as a software runtime verification simulator for the BIP system. The method then take the concurrent C program and generates an AIG circuit which is an FPGA implementation of the BIP system. The method applies synthesis reduction techniques using the ABC framework to simplify and reduce the AIG circuit into a smaller and a less complex circuit that can be readily implemented with an FPGA. The method passes the reduced AIG circuit with a designated output that is `true` when the BIP system invariants are `true` to ABC proof and model checking algorithms. In case ABC finds a counterexample, the methods maps the values from the counterexample to the original ABC system and provides the user with a debug visualization tool. We successfully used the system to verify and debug medium and large case studies.

Currently, the system specific scheduler makes conservative decisions to avoid interaction conflicts. Two interactions are conflicting if they share a port or they are using conflicting ports of the same component. An important extension is to allow parallel execution of non-conflicting interactions using techniques presented in [12].

References

1. A. Aziz, T. Shiple, V. Singhal, R. Brayton, and A. Sangiovanni-Vincentelli. Formula Dependent Equivalence for Compositional CTL Model Checking. *Journal of Formal Methods in System Design*, 21(2):193–224, 2002.

2. Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.

3. J. Barnat, L. Brim, and V. Havel. LTL model checking of parallel programs with under-approximated TSO memory model. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 51–59, July 2013.

4. Jiri Barnat, Lubos Brim, and David Safránek. High-performance analysis of biological systems dynamics with the DiVinE model checker. *Briefings in Bioinformatics*, 11(3):301–312, 2010.

5. Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, 2011.

6. Jason Baumgartner, Andreas Kuehlmann, and Jacob Abraham. Property checking via structural analysis. In *Computer-Aided Verification*, July 2002.

7. Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. Component-based verification using incremental design and invariants. *Software & Systems Modeling*, April 2014.

25

8. Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 614–619. Springer Berlin Heidelberg, 2009.

9. Sergey Berezin, Sérgio Campos, and Edmund M Clarke. *Compositional reasoning in model checking*. Springer, 1998.

10. Per Bjesse and Arne Boralv. DAG-aware circuit compression for formal verification. In *Int'l Conference on Computer-Aided Design*, Nov. 2004.

11. Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, November 2000.

12. Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.

13. R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40. Springer, 2010.

14. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2), 1992.

15. Tony Bybell. Gtkwave electronic waveform viewer, 2010.

16. M.R.V Chaudron, E.M. Eskenazi, A.V. Fioukov, and D.K. Hammer. A framework for formal component-based software architecting. In *OOPSLA*, pages 73–80, 2001.

17. Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

18. Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

19. Abhijit Davare, Douglas Densmore, Liangpeng Guo, Roberto Passerone, Alberto L. Sangiovanni-Vincentelli, Alena Simalatsar, and Qi Zhu. metroii: A design environment for cyber-physical systems. *ACM Trans. Embedded Comput. Syst.*, 12(1s):49, 2013.

20. Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for dpll (t). In *Computer Aided Verification*, pages 81–94. Springer, 2006.

21. Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

22. Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. Speculative linearizability. *Acm Sigplan Notices*, 47(6):55–66, 2012.

23. Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *FM 2006: Formal Methods*, pages 1–15. Springer, 2006.

24. Pei-Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart simulation using collaborative formal and simulation engines. In *Int'l Conference on Computer-Aided Design*, Nov. 2000.

25. G. Holzmann. The model checker SPIN. In *IEEE Transactions on Software Engineering*, May 1997.

26. Andreas Kuehlmann and Jason Baumgartner. Transformation-based verification using generalized retiming. In *Computer-Aided Verification*, July 2001.

27. Andreas Kuehlmann, Malay Ganai, and Viresh Paruthi. Circuit-based Boolean reasoning. In *Design Automation Conference*, pages 232–237, June 2001.

28. H. Mony et al. Scalable automated verification via expert-system guided transformations. In *Formal Methods in Computer-Aided Design*, November 04.

26

29. Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

30. Hari Mony, Jason Baumgartner, Viresh Paruthi, and Robert Kanzelman. Exploiting suspected redundancy without proving it. In *Design Automation Conference*. ACM Press, 2005.

31. In-Ho Moon, Gary D. Hachtel, and Fabio Somenzi. Border-block triangular form and conjunction schedule in image computation. In *Formal Methods in Computer-Aided Design*, Nov. 2000.

32. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Linto Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *ACM Design Automation Conference*, June 2001.

33. Seyed Hosein Attarzadeh Niaki and Ingo Sander. An automated parallel simulation flow for heterogeneous embedded systems. In *Design, Automation and Test in Europe (DATE)*, pages 27–30, 2013.

34. Preeti Ranjan Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM.

35. Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in forsyde. *IEEE Trans. on CAD (TCAD) of Integrated Circuits and Systems*, 23(1):17–32, 2004.

36. Ellen Sentovich, Kanwar Jit Singh, Cho W. Moon, Hamid Savoj, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *ICCD*, pages 328–333. IEEE Computer Society, 1992.

37. M. Sipser. *Introduction to the Theory of Computation*, volume 27. Thomson Course Technology Boston, MA, 2006.

38. Verimag. Rigorous design of component-based systems - the bip component framework. URL: http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=.

39. Dong Wang. *SAT based Abstraction Refinement for Hardware Verification*. PhD thesis, Carnegie Mellon University, May 2003.

27