Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

From High-Level Modeling Towards Efficient and Trustworthy Circuits

Fadi A. Zaraket[1], Mohamad Jaber[1], Mohamad Noureddine[2], Yliès Falcone[3]

[1] American University of Beirut, Beirut, Lebanon
e-mail: {fz11,mj54}@aub.edu.lb
[2] University of Illinois at Urbana-Champaign, Performability Engineering Research Group, Urbana, IL, USA
e-mail: nouredd2@illinois.edu
[3] Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France
e-mail: Ylies.Falcone@univ-grenoble-alpes.fr

The date of receipt and acceptance will be inserted by the editor

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. BIP framework comes with a runtime engine and a suite of verification tools that use D-Finder and NuSMV as model-checkers.

In this paper, we provide a method and a supporting tool that take a BIP system and a set of invariants and compute a reduced sequential circuit with a system-specific scheduler and 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 circuit implementation of the system that can be readily translated into FPGA or ASIC implementations, and to (2) verify the system and debug it in case a counterexample is found. Moreover, we generate a concurrent C implementation of the circuit that can be directly used for runtime verification.

We evaluated our method with two benchmark systems, and our results show that, compared to existing techniques, our method is faster and scales to larger sizes.

1 Introduction

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 intellectual property (IP) components of *heterogeneous* computational nature, i.e., some might be implemented as software processor executables while some others as real-time logic circuits. *Field-programmable gate array* (FPGA) logic circuits are popular logic circuit implementations of embedded system components because they are amenable for reconfiguration and can perform several computational tasks simultaneously. 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 [36]. Partitioning, often done manually, is used to decide whether a component is to be implemented as a programmed process or as a real-time logic circuit. A plethora of software, behavioral, and logic compilation and synthesis techniques are used in the process [28]. The end result implementation is then subject to functional verification including model-checking and runtime verification.

The design flow faces three important challenges of relevance to this paper.

– Model-checking faces the state-space explosion problem which often renders the results of model-checking inconclusive.
– The logical capacity of a reconfigurable FPGA board is limited. Thus, the size of the logic circuit implementations corresponding to IP components decide 1) how many components can be loaded simultaneously on the board, and 2) whether IP swapping is needed or not at runtime. Moreover, the critical depth of the logic circuit implementation decides how fast the board can be clocked.
– Runtime verification of embedded systems with general-purpose runtime verification engines exhibits expensive runtime overhead.

Behavior-Interaction-Priority (BIP) is a framework for the design of *Component-Based Systems* (CBSs). BIP uses a dedicated language and tool-set to support a rigorous and layered design flow for embedded systems. BIP is currently being used in academy and in industry in projects such as ASCENS, COMBEST, PRO3D, SMECY, ACROSS, MARAE, GOAC, MIND and CHAPI [1]. BIP allows to build complex systems by coordinating the behavior of a set of atomic components [8]. BIP makes use of (1) D-Finder [12], a compositional and incremental verification tool-set, and (2) NuSMV [26] to model-check the correctness of BIP systems.

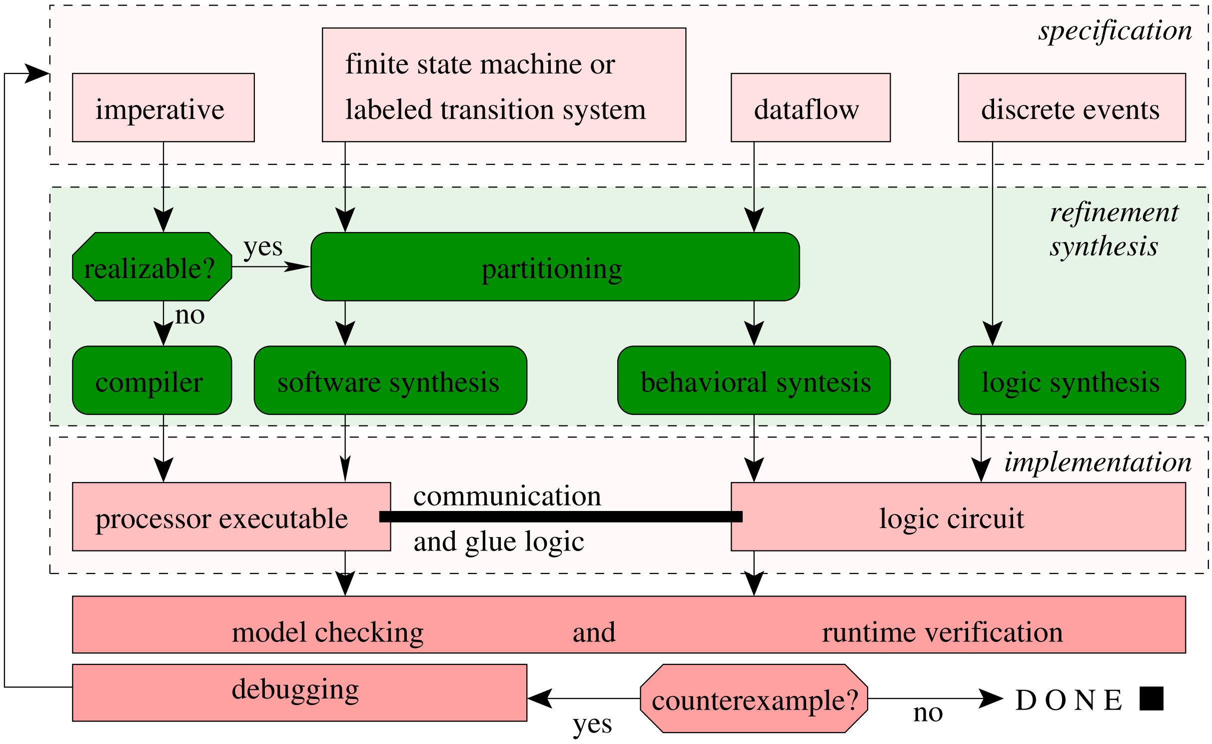

Fig. 1. Embedded system specification, refinement, and implementation stages

However, D-Finder [11] does not handle data transfer between components [56], and the available online version only supports deadlock-freedom check. Additionally, for complex systems, NuSMV often suffers from the state-explosion problem [59], and fails to perform its verification tasks.

ABC [21] is a transformation-based verification framework [41] that operates on And-Inverter Graphs (AIG); semi-canonical Boolean netlists with memory elements. It employs iteratively and synergistically: (1) powerful reduction, (2) abstraction, and (3) decision algorithms; such as retiming [41], redundancy removal [47,42,17,4], logic rewriting [15], interpolation [44], and localization [60], symbolic model-checking, bounded model-checking, induction, interpolation, circuit SAT solving, and target enlargement [49,50,37,10,43].

In this paper, we present a method and a supporting tool ($\mathcal{BipSV}$) 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 intermediate C-like *one loop program* ($\mathcal{OLP}$). The translation to $\mathcal{OLP}$ is necessary to allow for runtime verification and for the use of ABC verification algorithms. We consider invariant properties which are Boolean expressions over atomic propositions on components (e.g., constraints on the current locations and values of variables).
2. The method then translates the $\mathcal{OLP}$ program to an AIG circuit with an output therein that holds iff the system is deadlock free, and satisfies the system invariants. The method passes the generated AIG circuit to ABC for reduction and verification. The method drives the ABC reduction and verification algorithms and 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 D-Finder and NuSMV.
3. $\mathcal{BipSV}$ 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 [24].
4. The method generates a *field-programmable gate array* (FPGA) implementation of the BIP system with a system-specific execution framework. An FPGA implementation is a configuration of memory elements and lookup-up tables (LUT) provided with an FPGA board that implements a specific logical function and appropriately performs the desired computation. FPGA implementations are directly mapped to other integrated circuit representations such as *application-specific integrated circuits* (ASIC).
$\mathcal{BipSV}$ constructs the FPGA implementation from the reduced AIG circuit to benefit from the area and critical-time reduction algorithms of the ABC framework. The reduction algorithms remove redundant latches and logic gates. To the best of our knowledge, we are the first to directly synthesize an FPGA from a BIP system.
5. The method translates the $\mathcal{OLP}$ program into a concurrent C implementation of the BIP system. The implemen-

tation can be used for runtime verification as well as a direct software implementation. Moreover, in case the design was to be partitioned into software and hardware, parts of the implementation are readily available to execute on CPUs.

Our results show that $\mathcal{B}ip\mathcal{SV}$ successfully verifies large systems that are not possible to verify with existing techniques. The method also achieves significant reductions in FPGA size and depth reported as the number of gates and logic levels before and after the reductions.

BIP is based on the generation of modular code and a dedicated platform, the so-called BIP *engine*, which interprets the BIP semantics and orchestrates the computation of atomic components. This modularity favors the clarity of models but implies a prohibitive inefficiency. The main loop of the BIP 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 schedules their transitions for execution.

Compared to the BIP engine, our method differs in that it directly embeds a system-specific scheduler represented by a bit vector of interactions in the implementation. The value of the interaction bit vector 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 (see Section 8 for a detailed comparison with related work). Metropolis [5,28] is a design framework that takes a Metropolis Meta Model description of an embedded system and generates a SystemC [54] based simulator of the system. It uses the SIS toolset [58] for synthesis and the SPIN model-checker for verification [38]. SystemC [54] 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 [57] restrict the expressiveness to enable formal verification tools to handle the system. In 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. Our method simplifies 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 3, we recall the necessary concepts of the BIP framework. Section 4 defines one loop programs ($\mathcal{OLP}$). Section 5 formalizes sequential circuits and shows how to translate a sequential circuit into an $\mathcal{OLP}$. Section 6 shows how to translate a BIP system into an $\mathcal{OLP}$. Section 7 describes $\mathcal{B}ip\mathcal{SV}$, a full implementation of our framework and some benchmarks. Section 8 discusses related work. Section 9 draws some conclusions and perspectives.

2 Preliminaries and notation

We introduce some preliminary concepts and notations.

Functions. For two functions $v \in [X \to Y]$ and $v' \in [X' \to Y']$, the substitution function, noted v/v', with $v/v' \in [X \cup X' \to Y \cup Y']$, is defined as: $v/v'(x) = v'(x)$ if $x \in X'$ and $v(x)$ otherwise.

Transition systems. Labeled Transition Systems (LTS) are used to define the semantics of BIP systems. An LTS defined over an alphabet Σ is a 3-tuple $(\mathrm{Lab}, \mathrm{Sta}, \mathrm{Trans})$ where Lab is a set of labels, Sta is a non-empty set of states and $\mathrm{Trans} \subseteq \mathrm{Sta} \times \mathrm{Lab} \times \mathrm{Sta}$ is the transition relation. A transition $(s, e, s') \in \mathrm{Trans}$ means that the LTS can move from state s to state s' by consuming label e. We abbreviate $(s, e, s') \in \mathrm{Trans}$ by $s \xrightarrow{e}_{\mathrm{Trans}} s'$ or by $s \xrightarrow{e} s'$ when clear from the context. Moreover, $s \xrightarrow{e}$ is a short for $\exists s' \in \mathrm{Sta} : s \xrightarrow{e} s'$.

3 BIP - Behavior interaction priority

We recall the necessary concepts of the BIP framework [8]. BIP allows to construct systems by superposing three layers of design: 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.

3.1 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.

3.1.1 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).

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 `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, $\mathbf{X} \in [X \to \mathrm{Data}]$, where Data is the set of all values possibly taken by variables in X.

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:

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

A state 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. T_B is the set of transitions. The evolution of states $(l, v) \stackrel{p(v_p)}{\to} (l', v')$, where v_p is a valuation of the variables attached to port p, is possible if there exists a transition $(l, p[x_p], g_\tau, f_\tau, l')$, such that $g_\tau(v) = \mathtt{true}$. In this case, we say that p is *enabled* in state (l, v). Execution of port p results in updating the valuation of v to $v' = f_\tau(v/v_p)$.

Note that the valuation of the variables attached to port p are further instantiated when composing components with respect to data transfer functions of interactions [40,33].

3.1.2 Composing atomic components

Assuming some available atomic components $B_1, \ldots, B_n$, we show how to connect a subset $\{B_i\}_{i\in I}, I \subseteq [1, n]$, of the components 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 $X_a = \cup_{p\in P_a} x_p$ the set of variables available to interaction 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 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 interactions 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 rule defined in Figure 2. In this rule, 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. μ_i denotes the valuation of the variables attached to port p_i after executing function F_a of interaction a.

The meaning of the rule defined in Figure 2 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

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

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

Fig. 2. Semantics Rule of Composite Component

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:

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

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, $\mathit{Init} \in B_1.L \times \ldots \times B_n.L$ is the initial state of B, and $v \in \mathbf{X}^{\mathbf{Init}}$ where $X^{\mathit{Init}} \subseteq \cup_{i=1}^{n} B_i.X$.

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

We define the function `index` that assigns for each interaction $a \in \gamma$ a positive integer in $\big[0, |\gamma| - 1\big]$, i.e., `index` : $\gamma \rightarrow \big[0, |\gamma| - 1\big]$.

Definition 9 (Trace). A trace t of length ℓ of a system (B, Init, v) is the sequence of global states $q_0 \cdot q_1 \cdots q_{\ell-1}$ such that: $q_0 = (\mathit{Init}, v)$, and $\forall i \in [0, l-1] : q_i \in Q \wedge \exists a_i \in A : q_i \xrightarrow{a_i}_\pi q_{i+1}$. That is, a_i is an interaction enabled on q_i and its execution results in state q_{i+1}. We denote by $t[i]$ the i^{th} state in the trace, i.e., state q_i.

Example 1. 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.

4 One loop programs ($\mathcal{OLP}$) - syntax and semantics

This section introduces the syntax and semantics of one loop programs.

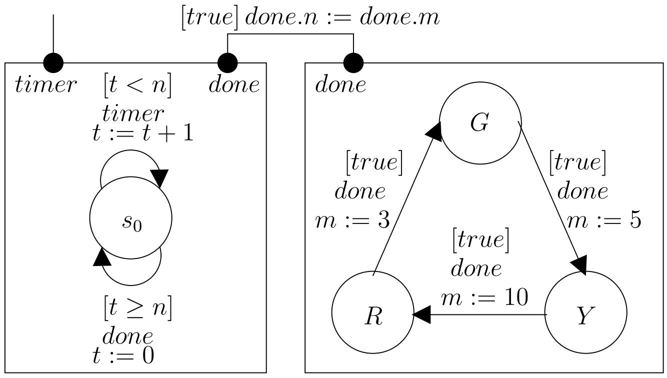

Fig. 3. Traffic light in BIP

4.1 Syntax of one loop Programs

Figure 5 illustrates the syntax of a *one loop program* ($\mathcal{OLP}$). An $\mathcal{OLP}$ starts with a list of variable declarations `decl-list`. $\mathcal{OLP}$ declarations allow Boolean, integer, array of Boolean and array of integer types. The `wire` modifier keyword is used to denote that the variable is a wire. Otherwise, the variable is a register variable. A register variable represents a data storage/memory element. A wire represents a functional macro, which is used to connect different elements (wires or registers).

Definition 10 ($\mathcal{OLP}$ variables). The set of variables V of an $\mathcal{OLP}$ is defined to be the set of all non-wire, i.e., register variables declared in `decl-list`. Function *type* : $V \mapsto \{\mathit{int}, \mathit{Boolean}, \mathit{int}[1], \mathit{Boolean}[1], \mathit{int}[2], \mathit{Boolean}[2], \ldots\}$ maps a variable $v \in V$ to its declared type.

The `wiredef-list` follows `decl-list` and is a list of assignment statements where the target term is a wire variable. An assignment has a left-hand side `term` and a right-hand side expression `expr`. The term is either an identifier `id` or an array access expression `id[expr]` where `id` is the name of the array and `expr` is an expression. $\mathcal{OLP}$ expressions are built with terms, expressions with unary operators (`-`, `!`), expressions with binary operators (`+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=`, `==`, `&&`, `||`), or expressions with a ternary choice operator (`?` `:`). Let s be a `wiredef-list` assignment with target t and expression e; expression e must not refer to target t.

The `init-list` and the `next-list` are lists of assignment statements where the target terms are register variables. The `init-list` is embodied in a `do-together` construct, which implies concurrent execution of all its statements. Expressions in `init-list` assignments must not refer to non-wire variables. The `next-list` is embodied in a `do-together` construct which is in turn embodied in a `while(true)` loop construct. The loop makes sure that the design runs indefinitely.

Definition 11 (Well-formed $\mathcal{OLP}$). An $\mathcal{OLP}$ is *well-formed* when both the `init-list` and the `next-list` contains one assignment per non-wire variable and the `wiredef-list` contains at most one assignment per wire.

Hereafter, we consider only well-formed $\mathcal{OLP}$.

Definition 12 (Non-deterministic wires). We define the set of *non-deterministic wires* of an $\mathcal{OLP}$ to be the set of wire variables that are not targets of assignment statements in `wiredef-list`.

Definition 13 (init and next state functions). Consider $v \in V$ and consider s_{init} and s_{next} the assignment statements where v is the target term in `init-list` and `next-list`, respectively. We define functions `init-state(v)` and `next-state(v)` to be the functions corresponding to the right-hand side expressions of s_{init} and s_{next}, respectively.

Example 2 (Well-formed $\mathcal{OLP}$). Figure 4 shows an $\mathcal{OLP}$ that corresponds to the BIP system for the traffic-light controller shown in Figure 3.

4.2 Semantics of one loop programs

Recall that a variable can be either a register denoting a memory element, or a wire denoting a functional macro. Memory variables are initialized simultaneously using the `do-together` construct. After initialization, an infinite loop keeps updating the value of memory variables simultaneously. The listings in Figure 5 shows the syntax of an $\mathcal{OLP}$.

If a wire is not assigned, then it is said to be a non-deterministic *primary input*. It takes a new non-deterministic value at each iteration of the loop. The list of statements `init-list` assigns initial values to the register variables. Similarly, the `next-list` list of statements updates the values of the register variables. The semantics of $\mathcal{OLP}$ expressions are defined by the typical valuation rules of the corresponding unary and binary operators. The ternary choice `(a? b : c)` returns b if a is `true` and c otherwise.

The formal semantics of $\mathcal{OLP}$ is given in terms of $\mathcal{OLP}$ state and trace as follows. For this purpose, we consider an $\mathcal{OLP}$ P ranging over a set of non-wire variables $V = \{v_1, v_2 \ldots, v_n\}$.

Definition 14 ($\mathcal{OLP}$ state). The state of P is defined as the valuation $\sigma : V \to D$. The valuation σ maps variables in V to $D = \mathbb{B} \cup Data \cup \mathbb{B}^k \cup Data^k$ such that $\sigma(v_i) \in \mathbb{B}$ (resp. $Data$, $\mathbb{B}^k$, $Data^k$) when $type(v_i)$ is `Boolean` (resp. `int`, `Boolean[k]`, and `int[k]`), where $1 \leqslant i \leqslant n$ and $k > 0$.

Definition 15 (`do-together` semantics). All the assignment statements `init-list` and `next-list` can execute simultaneously as indicated with the `do-together` construct.

Definition 16 ($\mathcal{OLP}$ trace). A trace π of length ℓ of P is a sequence of $\mathcal{OLP}$ states $\sigma_0, \sigma_1, \ldots, \sigma_{\ell-1}$. State σ_0 is defined as the valuation given by the `init-state(`v_i`)` functions, with $1 \leqslant i \leqslant n$. State σ_{k+1} corresponds to the valuations given by functions `next-state(`v_i`)` where references to variables $v_j \in V$ are substituted by the corresponding valuations from σ_k, $0 \leqslant k \leqslant \ell$.

In Section 6, we shall see how to automatically translate a BIP system into $\mathcal{OLP}$.

5 From $\mathcal{OLP}$ to sequential circuits

We define the translation of $\mathcal{OLP}$ to sequential circuits.

Definition 17 (Sequential circuit). A *sequential circuit* is a tuple $((V, E), G, O)$. Pair (V, E) represents a directed graph on vertices V and edges $E \subseteq V \times V$, where E is a total order. Function $G : V \to Types$ 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 denote the set of primary input variables by I, and the set of bit-register variables by R.

Definition 18 (Fanins and fanouts). The direct *fanin*s of a gate u are defined as $\{v \in V \mid (v, u) \in E\}$, i.e., the set of source vertices connected to u in E.

The direct *fanout*s of a gate u are defined as $\{v \mid (u, v) \in E\}$, i.e., the set of sink vertices connected to u in E. The *support* of u is $Fanins(u) \cap (I \cup R)$, i.e., the set of all source vertices that are either primary inputs or registers that are connected to u.

For a 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 register in their support. In the following, we consider only well-formed sequential circuits which can be verified by a structural check that is linear in the size of the sequential circuit.

The ABC synthesis and model-checker framework reasons about the And-Inverted-Graph (AIG) representation of a sequential circuit which are sequential circuits with only NAND gates and with exactly two fanins [21].

We describe useful reduction and verification ABC algorithms in Appendix A.

```
/*** 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.s = is[0];
timer.done.s  = is[1];
light.done.s  = 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.s && timer.ℓ == 0)?
             (0) : (timer.ℓ)));

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

    light.ℓ = (cycle)? (light.ℓ) : ((light.ℓ == 2 && light.done.s)?
             (0) : ((light.ℓ == 1 && light.done.s)?
                (0) : ((light.ℓ == 0 && light.done.s)?
                   (1) : (light.ℓ))));

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

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

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

```
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+
assignment: term = expr
wiredef-list: (assignment)*

init-list: (assignment)*
next-list: (assignment)*
```

Fig. 5. $\mathcal{OLP}$ Syntax

5.1 Semantics of sequential circuits

The semantics of a sequential circuit is defined in terms of its states and traces.

Definition 19 (AIG state). An AIG *state* $\sigma : R \to \mathbb{B}$ is a Boolean valuation of vertices in R.

Definition 20 (AIG full trace). An AIG *full trace* is a mapping $t : V \times \mathbb{N} \to \mathbb{B}$ that gives a value to vertices in V across time *steps* denoted as indexes from $\mathbb{N}$: The mapping must be consistent with E and G in the following sense. The value of gate v at time i in full trace t is denoted by $t(v, i)$ as defined in Figure 6.

The well-formedness constraint guarantees the absence of combinational cycles in the AIG. Therefore, given a sequence of input valuations and an initial state, Figure 6 defines the resulting trace as a sequence of Boolean valuations to all ver-

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

Fig. 6. Semantics of sequential circuits given in terms of full traces. $t(v, i)$ denotes the valuation of gate v at step i in trace t. Term u_j denotes the source vertex of the j-th incoming edge to v, that is, $(u_j, v) \in E$.

tices in V which is consistent with the Boolean functions of the gates.

Definition 21 (AIG trace). An *AIG trace* of length ℓ is a sequence of AIG states $\rho = s_0, s_1, \ldots, s_{\ell-1}$. Given a full AIG trace t, we can compute $\rho = s_0, s_1, \ldots, s_{\ell-1}$ where $s_i = \{(r_0, b_0^i), \ldots, (r_{|R|-1}, b_{|R|-1}^i)\}$, $r_j \in R$, $b_j^i \in \mathbb{B}$, $0 \leqslant i < \ell$ and $0 \leqslant j < |R|$ and $((r_j, i), b_j^i) \in t$.

We will refer to the transition from one valuation to the next one as a *step*. A vertex in the circuit is said to be *justifiable* if there is an input sequence which, when applied to an initial state, will result in that vertex taking value `true`. A vertex 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 reachable states. However, the circuit is very different from its FSM; among other differences, it is exponentially more succinct in almost all cases of interest [22].

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

Algorithm **olp-to-aig** shown in Figure 7 takes an $\mathcal{OLP}$ P as input and constructs an *equivalent* AIG. An illustration example is provided in Figure 10. The steps of the algorithm are as follows.

1. It first instantiates AIG registers, wires, and primary inputs that correspond to $\mathcal{OLP}$ variables using the **variables** routine.
2. It then calls the recursive routine **traverse** to translate the right-hand side expressions of the assignment statements in `wiredef-list`, `init-list`, and `next-list` into AIG combinational circuits.
3. It connects the resulting vertices of the combinational circuits of the right-hand side expressions to the fanins of the registers corresponding to the left-hand side target variables.
   (a) The vertices corresponding to the `init-list` right-hand side expressions are connected to the initial value fanins of the registers.
   (b) Similarly, those of the `next-list` are connected to the next state value fanins.
   (c) Finally, it connects the vertices of the combinational circuits built for the `wiredef-list` expressions to the corresponding wires referring to the variables declared as wire variables in `decl-list`.

```
// P is an OLP program
olp-to-aig(P)
  // instantiate aig variables and construct
  // vargates
  variables(P.decl-list);

  // s is of the form term = expr
  foreach assignment s ∈ init-list
    next-state(s.term) = traverse(s.expr);
  endfor

  foreach assignment s ∈ wiredef-list
    vargates(s.term) = traverse(s.expr);
  endfor

  foreach assignment s ∈ next-list
    next-state(s.term) = traverse(s.expr);
  endfor
```

Fig. 7. $\mathcal{OLP}$ to AIG transformation

Variables. We consider each variable not declared as a wire in `decl-list` (see Figure 8). 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 butl only have fixed-size arrays of Boolean wires as discussed in Section 6. We say that a variable declared as wire in `decl-list` is non-deterministic when it does not have a corresponding assignment statement in `wiredef-list`. 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 v by the function `vargates`(v).

Assignment statements. We consider each assignment statement in `wiredef-list`, `init-list`, and `next-list` and traverse the right-hand side expressions of each assign-

```
variables(decl−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)
      // non-deterministic input
      vargates(v) = instantiate-primary-inputs(
                              v,type(v))
    endif
  endfor
```

Fig. 8. Routine variables

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

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

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

Fig. 9. Routine traverse

ment with the recursive `traverse` routine (see Figure 9). 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 finds an equivalent circuit for it with the adequate bit width in a complete table of circuits. For example, if the expression is a ternary conditional statement of the form $b?\ e_1 : e_2$, then routine `library` instantiates a multiplexer, connects its two data fanins to the vertices corresponding to e_1 and e_2, connects its control fanins to the vertices corresponding to b, and returns its fanouts.

Invariants. A special variable in the $\mathcal{OLP}$ program denotes the conjunction of all the invariants of the system in addition to the deadlock-freedom property. This variable is the designated output of the resulting AIG circuit. ABC verifies that the designated output is always `true`.

Definition 22 (AIG $\mathcal{OLP}$ state equivalence). An $\mathcal{OLP}$ state $\sigma = \{(v_0, d_0), \ldots, (v_{|V|-1}, d_{|V|-1}\}$ and an AIG state $s = \{(r_0, b_0), \ldots, (r_{|R|-1}, b_{|R|-1})\}$, are said to be equivalent iff $s(vargates(v_i))$ is equal to the binary representation of $\sigma(v_i)$, for each $0 \leqslant i < |V|$.

We are now ready to state the (trace) equivalence between AIG and $\mathcal{OLP}$.

Theorem 1 (AIG $\mathcal{OLP}$ trace equivalence). *Let P be an $\mathcal{OLP}$ and A be the AIG circuit generated from it (i.e., A = `olp-to-aig(P)`). Let I be the set of non-deterministic wires of P. Set I also corresponds to the set of corresponding primary inputs of A. Given a sequence ρ of length ℓ of input valuations of I, traces $\pi_p = \sigma_0^p, \sigma_1^p, \ldots, \sigma_{\ell-1}^p$ and $\pi_a = \sigma_0^a, \sigma_1^a, \ldots, \sigma_{\ell-1}^a$ produced by P and A, respectively, are equivalent, i.e., σ_i^p and σ_i^a are equivalent for all $0 \leqslant i < \ell$.*

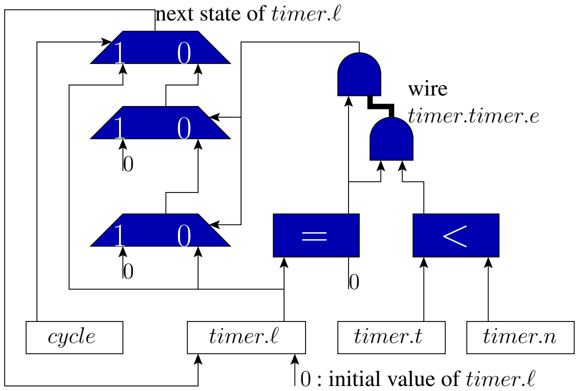

Fig. 10. Sample circuit for the $timer.\ell$ registers; the $<$, $=$, and multiplexer gates can be easily implemented using NAND gates.

Proof. The proof is by induction on the length of traces.

Base case: The initial states σ_0^p and σ_0^a are equivalent since the initial state functions of registers R in A and the right-hand side expressions of the corresponding assignments in `init-list` of P are equivalent by construction.

Inductive step: Similarly, the next state functions of the registers R in A and the right hand side expressions of the corresponding assignments in `next-list` of P are equivalent by construction. It follows from the induction hypothesis that states σ_i^p and σ_i^a are equivalent for a given $0 \leqslant i < \ell - 1$. Since all the next state functions of A evaluate simultaneously in one step, and similarly all the assignment statements in the `next-list` execute simultaneously in one iteration of the sole loop in P, the resulting states σ_{i+1}^p and σ_{i+1}^a are equivalent. Thus, π_p and π_a are equivalent and therefore P and A are trace equivalent.

Example 3 (Generated AIG circuit). Figure 10 shows a circuit generated by traversing the right-hand side expressions of the initial value and next state function assignment corresponding to variable $timer.\ell$. The sample circuit shows only the AND, $=$, $<$ and multiplexer gates for simplicity; all those gates can be readily implemented using NAND gates. A multiplexer takes a Boolean control input, and uses its value to choose one of its two data inputs.

The next state function depends on variables $cycle$, $timer.t$, and $timer.n$ and on the wire variable $timer.timer.e$. The registers of the variables are connected directly to the circuit. The circuit for the wire variable $timer.timer.e$ is constructed by traversing the right-hand side expression of its assignment in the wire definition list. Then, the constructed circuit is connected to the input of the corresponding AND gate.

Note that (1) the initial value of $timer.\ell$ registers is 0, and (2) the next state fanins of $timer.\ell$ are connected to a multiplexer whose data inputs are equal to 0 or $timer.\ell$. Thus, the constant propagation algorithm replaces $timer.\ell$ with 0 and propagates that effect.

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

Fig. 11. Translation of BIP system into an $\mathcal{OLP}$ program.

6 BIP to $\mathcal{OLP}$

Given a BIP system $\mathcal{S} = (B, Init, v)$, $\mathcal{BipSV}$ calls function `BIP-to-OLP` (see Figure 11) to translate $\mathcal{S}$ into an $\mathcal{OLP}$ including an encoding of the semantics of interactions and priorities. It calls four functions that fill `decl-list`, `wiredef-list`, `init-list`, `next-list`. All these functions use the `append` call to add code fragments to lists.

1. Function `generateDeclarationList()` (see Figure 12) fills `decl-list` as follows. It creates three arrays of wires to denote interaction semantics. The elements of array ie denote whether all logical constraints except priority rules are met for a given interaction. The elements of array ip denote whether a given interaction is enabled after applying priority rules. The elements of array is 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 the same components. In order to avoid concurrently-executing conflicting interactions, BIP provides centralized, multithreaded and distributed implementations. In the centralized implementation, the engine executes only one interaction at a time. In the multi-threaded implementation [9], the involved components in the non-conflicting interactions execute simultaneously with no overhead except the classical thread synchronization overhead. However, while each component executes in a separate thread, the engine executes in a single-engine thread. The single-engine thread is responsible for sequentially (1) selecting an interaction for execution, (2) executing the corresponding action, and (3) signaling the static threads associated with the involved components for execution.
Alternatively, BIP allows the generation of distributed implementations [18] where non-conflicting interactions can be simultaneously executed. However, an additional layer is added to resolve conflicts. This may introduce significant overhead due to communication between the layers. The overhead may drastically increase when interactions do not involve heavy computations, which is the case in general since most interactions involve data transfer.
In our case, it is possible to add a circuit that identifies and enables all non-conflicting interactions with a simple modification that merges the consecutive execution cycles of non-conflicting interactions into one cycle. Such transformation is available for free with the retiming algorithm in ABC [39] and allows us to compare well with the distributed implementation in [18].
Furthermore, the selection of non-conflicting interactions can happen simultaneously with $\mathcal{BipSV}$ As opposed to sequential as in the multi-threaded BIP implementation [9]. The multi-threaded implementation does not have to wait for all non-conflicting interactions to complete before executing new interactions. $\mathcal{BipSV}$ is currently restricted to a cycle implementation since this is a necessary constraint for generating code that can be synthesized into FPGAs. This constraint can be relaxed by allowing longer interactions to span multiple cycles and introducing a busy state in the involved components. As for software execution, in the current model of execution, the one loop in $\mathcal{BipSV}$ iterates once all the assignments inside it are done. We can relax that to have the loop iterate when the first interaction is done and guard the assignments involved in the busy interactions with the necessary conditional logic.
Wire $selector$ is a non-deterministic primary input used to select one of the enabled interactions. Boolean register $cycle$ is used to denote whether the system is executing actions corresponding to either interaction or transitions. Function `generateDeclarationList()` also declares two wires ($B_i.p_j.e$ and $B_i.p_j.s$) for each port p_j. For a port p_j, wire $B_i.p_j.e$ indicates whether the port is enabled and wire $B_i.p_j.s$ indicates 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.

```
generateDeclarationList()
  // 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
  append bool b[|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
```

Fig. 12. `generateDeclarationList()` function.

```
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_j.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
```

Fig. 13. `generateWireDefList()` function.

```
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
  // iterate over interactions
  foreach  j ∈ [1..|J|]
    append  b[j] = is[j]  to  init−list
  endfor
```

Fig. 14. `generateInitList()` function.

2. Function `generateWireDefList()` (see Figure 13) 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 holds.
   Array element $ie[j]$, corresponding to interaction a_j, evaluates to `true` when the guard of a_j holds 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 and either (1) a_j is selected ($selector$ equals to j), or (2) the selected interaction is not enabled and all interactions with index greater than j are not enabled.
   The Boolean bit-vector b is redundant with wire is and is declared to simplify the proof of Theorem 2. The use of a 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 $B_i.p_j$.
3. Function `generateInitList()` (see Figure 14) 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 `false` to denote an interaction execution mode.
4. Function `generateNextList()` (see Figure 15) 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 (when $cycle$ is equal to `false`), the function considers each assignment statement σ from the action of interaction a_j. The function appends a conditional clause requiring 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 forms a nested ternary conditional expression where the last expression retains the previous value of the variable.
   Similarly, in the transition execution mode ($cycle$ equals to `true`), the function considers each assignment 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, variable $cycle$ is toggled.

6.1 Correctness

Given a BIP system $\mathcal{S}$ and its corresponding $\mathcal{OLP}$ program P = `BIP-to-OLP`$(\mathcal{S})$. Let Tr_s be the set of traces of $\mathcal{S}$ and let Tr_p be the set of traces of P. Consider T' the projection of Tr_p constrained by omitting the states where `cycle` is equal to `false`. Formally, $T' = \{t' \mid t'[i] = t[2 \times i] \wedge t \in Tr_P \wedge i \in \mathbb{N}\}$. Intuitively, T' represents the semantics of the original BIP model regardless of the built-in scheduler details (i.e., the enable exchange, interaction selection, data transfer details).

Theorem 2 (BIP $\mathcal{OLP}$ equivalence). *The BIP system S is semantically equivalent to P: $Tr_s = T'$.*

Proof. The proof is done by induction on the length of traces and on the structure of S and P.

Left case: $Tr_s \subseteq T'$. Consider $t \in Tr_s$, there exists $t' \in T'$ and $t = t'$.

- *Induction basis.* Consider the initial state: $t[0]$ and $t'[0]$. `generateInitList()` sets `cycle` to `true` and assigns each location to $Init$ and each variable to its initial value. Thus, $t[0]$ is equal to $t'[0]$.

```
generateNextList()
  // iterate over interactions
  foreach j ∈ [1..|J|]
    append b[j] = is[j] to next−list
  endfor

  // 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
```

Fig. 15. `generateNextList()` function.

– Let $t[0..k]$ be the prefix of t of length $k+1$ where $k \geqslant 0$, using the induction hypothesis, there exists at least one trace $t'[0..k] \in T'$ and $t[0..k] = t'[0..k]$.
Consider valuations $t[k]$ and $t'[k]$ that correspond to the firing interactions. Since they are equal, the next state of locations and data variables will be the same at $t[k+1]$ and $t[k+1]$ as enforced by `generateNextList()`.
In case no interactions are enabled at step $k+1$, both P and S will preserve the same state at step $k+1$ and thus $t[0..k+1] = t'[0..k+1]$. Otherwise, let a_j be an interaction that is enabled according to state $t[k+1]$, then according to BIP semantics, interaction a_j must be also enabled and of the highest priority. This means that a_j is also enabled and of the highest priority under $t'[k]$. The primary input `selector` wire variable is a nondeterministic variable and can assume all indexes on interactions including the value j, thus setting the enabled interaction a_j in $t'[k+1]$. Therefore, there exists a setting for `selector` at step $k+1$ such that P executes a_j.
Without loss of generality, let that nondeterministic setting be the one in $t'[k+1]$. Therefore, $t[k+1] = t'[k+1]$.

Right case: $T' \subseteq Tr_s$**.** Consider $t' \in T'$ there exists $t \in Tr_s$ and $t = t'$.

– *Base case.* The base case is as before and $t[0]$ is equal to $t'[0]$.
– *Induction case.* Let $t'[0..k]$ be the prefix of t' of length $k+1$ where $k \geqslant 0$, using the induction hypothesis, there exists a trace $t[0..k] \in Tr_s$ and $t[0..k] = t'[0..k]$. As for the previous proof, `generateNextList()` guarantees that the state of the locations and the data variables are equal in the next states $t[k+1]$ and $t'[k+1]$ of S and P, respectively.
Furthermore, the two states are equivalent if no interaction was enabled in $t'[k+1]$.
In case an interaction a_j is selected to execute in P at step $k+1$ as set in state $t'[k+1]$, then it must be enabled and of high priority. Thus, it must also be enabled and of high priority in S according to state $t[k+1]$ as guaranteed by `generateNextList()`. The semantics of BIP allows the nondeterministic selection of one of the enabled and high priority interactions, and without loss of generality let that selection be a_j in $t[k+1]$. Therefore $t[k+1] = t'[k+1]$.

Consequently, the claim holds as expected.

6.2 Embedding the built-in scheduler

The logic for the builtin scheduler is coupled with the port enable, port select, interaction enable, and interaction select circuits. It involves the priority settings of the interactions which translate to constant wires in the AIG. It also involves a `selector` primary input that assumes a nondeterministic value. The logic of the scheduler computes the enabled ports, then computes the enabled interactions. In case priority was not enough to select one interaction to execute, the logic of the scheduler uses the non-deterministic value of the `selector` to give higher priority to the interaction with the nearest index to the `selector`.

6.3 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 semantics of BIP requires to (1) first execute the data transfer of the selected interaction, and then to (2) execute the functions of the corresponding transitions of atomic components. For this purpose, in the above translation, we used the *cycle* Boolean register to indicate 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 analysis 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. Moreover, it is possible to do source-to-source transformations to compose the effect of data transfer, and hence one cycle-based implementations could be always generated.

7 Implementation and evaluation

7.1 $\mathcal{B}ip\mathcal{SV}$

$\mathcal{B}ip\mathcal{SV}$ (BIP Synthesis Verification) is an implementation of our method with several modules. The implementation is available at http://research-fadi.aub.edu.lb/dkwk/doku.php?id=biptoabc.

- The first module is a Java implementation of the translation from BIP to $\mathcal{OLP}$ described in Section 6. It takes as input a BIP system and a set of invariants, and generates the corresponding $\mathcal{OLP}$ with a system-specific execution framework.
- The second module generates a concurrent runtime verification executable from the $\mathcal{OLP}$ program that uses the OpenMP API to perform runtime verification (simulation) of the BIP system. The module sets the primary inputs to random values at each iteration, and replaces the `do-together` constructs with OpenMP directives to have the resulting binary running concurrently.
- The third module is a C++ implementation that transforms the $\mathcal{OLP}$ to an AIG circuit after preforming word-level constant propagation and cone of influence reductions. The module also passes the generated AIG circuit to the ABC framework, and drives the synthesis reduction algorithms, and then the verification algorithms. $\mathcal{B}ip\mathcal{SV}$ automatically selects the ABC algorithms to run based on structural AIG metrics. Alternatively, the user can interactively guide the reduction and verification processes.
- Finally, in case a counterexample is found by either the concurrent runtime verification module or by the ABC verification algorithms, a C++ module takes the counterexample, translates it back to BIP, and provides a user-friendly interface to visualize the counterexample and debug the system with an integrated open source wave form viewer tool [24].

$\mathcal{B}ip\mathcal{SV}$ uses ABC synthesis and reduction algorithms to reduce the area and the critical time of the AIG circuit by removing redundant latches and logic gates. Examples of reduction algorithms are retiming [41], redundancy removal [47,42,17,4], logic rewriting [15], interpolation [44], and localization [60]. The reduced AIG circuit is equivalent to the original circuit and $\mathcal{B}ip\mathcal{SV}$ can readily translate it into an 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 [49,50,37,10,43] 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.

$\mathcal{B}ip\mathcal{SV}$ 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 (e.g., debugging).

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

Moreover, $\mathcal{B}ip\mathcal{SV}$ takes as input a property to be verified expressed by pre and post conditions over atomic propositions. Atomic propositions are conditions on components (e.g., a condition on the lastly-executed port, current locations of atomic components, values of variables). The pre and post conditions are stored in a file to be parsed by $\mathcal{B}ip\mathcal{SV}$. For instance, the following snippet defines a property where: (1) the precondition is the condition that always holds (i.e., `true`); and (2) the post condition requires that when component $comp_1$ is at location s_0 and component $comp_2$ is at location s_1, then the variable x of $comp_1$ should be equal to the variable y of $comp_2$.

```
@pre inv { true; }
@post inv {((comp1_currentState == comp1_state_s0) && (
     comp2_currentState == comp2_state_s1)) -> (
     comp1_var_x == comp2_var_decidedValue);}
```

Additionally, we have built some predefined patterns such as deadlock expressed as an invariant denoting the set of the states from which all interactions are disabled.

We evaluated $\mathcal{B}ip\mathcal{SV}$ against two benchmarks used to evaluate BIP verification techniques, an *Automatic Teller Machine* (ATM) [25] and the *Quorum* consensus protocol [35]. 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 against another solution that uses a bisimulation-based abstraction for reduction [53] and NuSMV [26] as a model-checker.

7.2 The ATM benchmark

Automatic Teller Machine (ATM) is a computerized system that provides financial services for users in a public space. Figure 16 shows a structured BIP model of an ATM system adapted from the description provided in [25]. The system is composed of four atomic components: (1) the User (2) the ATM (3) the Bank Validation and (4) the Bank Transaction. The 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

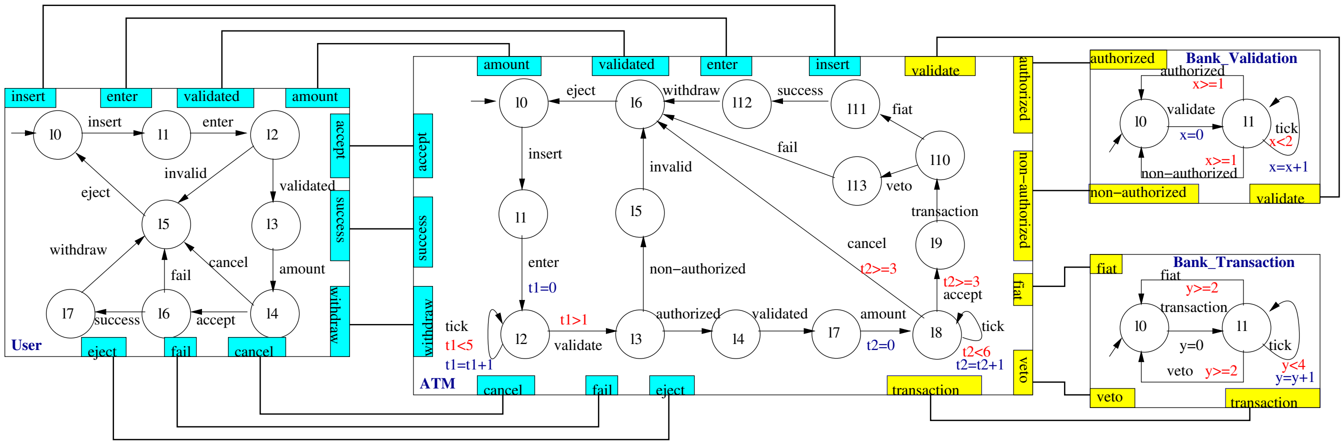

Fig. 16. Modeling of ATM system in BIP

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.

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 another users. In our model, we consider 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 [26]. The first column shows the number of clients and ATMs in the system. The table contains the number of latches, NAND 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 total time taken to perform synthesis (reduction) and verification by $\mathcal{B}ip\mathcal{SV}$, in addition to the time taken by NuSMV to perform verification. Note that the time to perform synthesis was negligible.

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 that when performing verification, ABC tries multiple verification and reduction algorithms before reaching a conclusive result. However, the advantage of $\mathcal{B}ip\mathcal{SV}$ is clearly that it scales with the number of ATMs and users.

7.3 The Quorum protocol

The *Quorum* protocol is a consensus protocol proposed in [35] complementary to the Paxos consensus protocol [34] under perfect channel conditions. *Consensus* allows a set of communicating processes (clients and servers in our case) to agree on a common value. Each client proposes a value and receives a common decision value. The authors in [35] propose to use Quorum when no failure occurs (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 timer t_c.
2. When a server receives a value v from a client c, it performs the following check.
   - If it 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 [35].

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 to value v.

	Original			After reduction			Time(s)	
ATMs	latches	NAND-gates	levels	latches	NAND-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

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.

Table 2 shows the verification time and the size of the circuit when 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 NAND gates and logic levels before and after applying reduction algorithms. The FPGA corresponding to the reduced circuit uses the same number of latches, and a proportional number of LUT connections to the NAND gates.

Using ABC's synthesis and reduction algorithms, we reduced the size of the generated AIGs (from $\mathcal{B}ip\mathcal{SV}$) 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 17 shows a snippet of the generated counter example for the erroneous design, visualized using the Gtkwave [24] 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.

8 Related Work

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

SystemC [54] 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.

The BIP framework differs from SystemC in that it presents a dedicated language and supporting tool-set 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.

Metropolis [5,28] 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.

Verification techniques for SystemC and BIP make use of symbolic model-checking tools. NuSMV [26] 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, NuSMV 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 encodes them as Boolean functions. In Section 7, we benchmark $\mathcal{B}ip\mathcal{SV}$ verification tasks against verification tasks using the NuSMV model-checker. The $\mathcal{OLP}$ translation differs from the NuSMV translation as follows.

- Only BIP variables and locations are encoded into registers in $\mathcal{OLP}$ and all other elements such as interaction and port enablement are encoded using wires. The NuSMV translation uses registers for all BIP elements; thus, implying a larger state space. Performing the same encoding in NuSMV requires the use of redundant expressions, which may cause redundant logic.
- $\mathcal{OLP}$ programs generated from BIP systems can be straightforwardly translated into concurrent C implementations with a minor modification (e.g. replacing the

	Original			After reduction			Time (s)	
Design	latches	NAND-gates	levels	latches	NAND-gates	levels	$\mathcal{BipSV}$	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

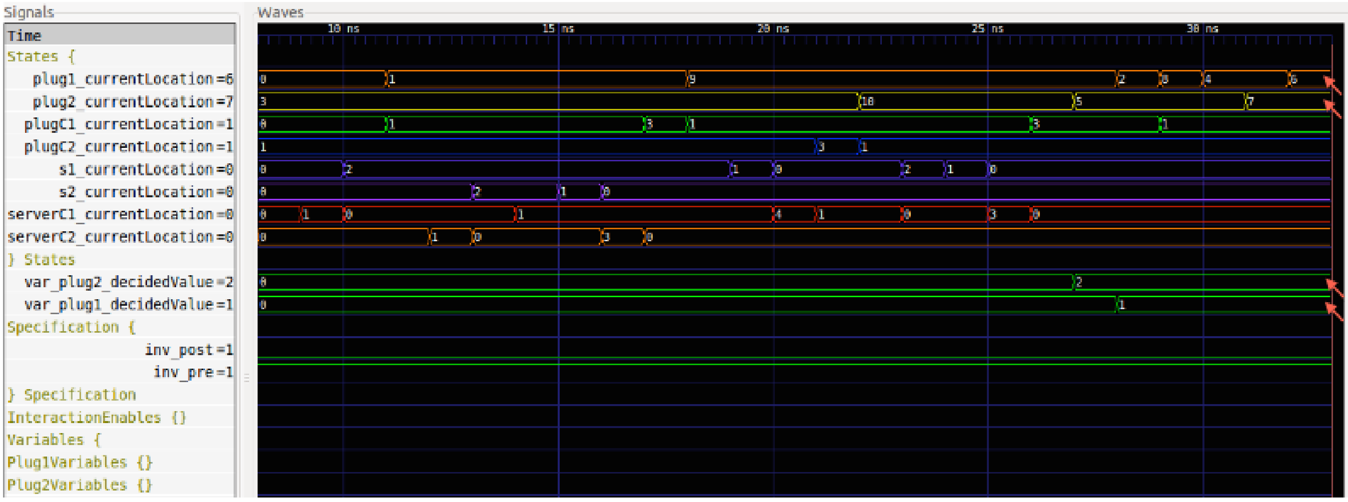

Fig. 17. Visualization of a counter example using Gtkwave

`do-together` directives with OpenMP API directives. The implementation can be used for runtime verification (c.f. [32]) as well as a direct software implementation. Moreover, in case the design was to be partitioned into software and hardware, parts of the implementation are readily available to execute on CPUs. Performing the same with the NuSMV implementation would require developing a new source-to-source translator.

The work in [53] uses bisimulation-based abstraction to reduce the state-space and then uses NuSMV for model-checking. Our technique can directly benefit from the abstraction of [53]. However, our experiments show that counterpart bit-level transformations were more effective. Moreover, our $\mathcal{OLP}$ to AIG transformation uses compact timing since it implements the built-in scheduler in the AIG circuit; while in [53], the transformation from the abstracted model to the NuSMV model enumerates all symbolic states. That is, with $\mathcal{BipSV}$ bounded model-checking can use lower time bound than [53]. Moreover, our method enables the use of a plethora of reduction and abstraction algorithms readily available at bit-level [21]. Since our transformation is time-exact the $\mathcal{OLP}$ program and the AIG circuit we generate can be used for runtime verification as well as real implementations.

The work in [55] takes a design specified in Esterel and translates it to a sequential circuit specified in Verilog or BLIF. Esterel and BIP differ in several ways. For example, Esterel is less expressive as it does not allow for multiparty interactions with non-deterministic behaviors while BIP does. In addition, our translation transforms a high level BIP model directly into a bit-level circuit by embedding built-in scheduling into the design. Moreover, it embeds the given properties into the generated circuits as designated outputs. This avoids the use of compilers to interpret models in Verilog.

The work in [52] uses constraint-based programming to compute an executable MPI-based parallel simulator of an embedded and cyber-physical systems written in ForSyDe [57]. ForSyDe is a library of SystemC based parametrized system components with strict constraint specifications and a blocking write FIFO queue modeling a Kahn network. The instances of the ForSyDe components are processes that communicate only through signals.

The work in [6] introduces a model-checking methodology for LTL specifications of embedded systems written in DIVINE [7] 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, NuSMV performs a cone of influence reduction [13] step in order to eliminate non-needed parts of the flattened model and specifications. The cone of influence reduction technique aims at simplifying the model at hand by only referring to variables that are of interest to the verification procedure, i.e. variables that influence the specifications to check [27].

D-Finder [12] is an automated verification tool for checking invariants on systems described in the BIP language. Given a BIP system $\mathcal{S}$ and an invariant $\mathcal{I}$, D-Finder 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 [29] to check for the validity of the formula $\mathcal{X} \wedge \neg\mathcal{I} = \texttt{false}$. Additionally, D-Finder checks the deadlock-freedom of $\mathcal{S}$ by building an invariant $\mathcal{I}_d$ that represents the states of $\mathcal{S}$ in which no interactions are enabled, i.e., a deadlock occurs. It then checks for the formula $\mathcal{X} \wedge \mathcal{I}_d = \texttt{false}$, 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, the compositional and incremental methods provided by D-Finder are limited to systems without data transfer over interactions. In [51], the authors proposed a method that transforms a system with data transfer into equivalent system without data transfer on which the compositional method can be applied. Nonetheless, the proposed method remains theoretical and not integrated into D-Finder. This limitation hampers the practical application of D-Finder and of the BIP framework, since data transfer is necessary and common in the design of practical applications.

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).

9 Conclusion and future work

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 takes 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 several case studies.

For future work, we consider several research directions. Currently, the system-specific scheduler makes conservative decisions to avoid interaction conflicts. Two interactions conflict if they share a port or they use conflicting ports of the same component. An important extension is to allow for the parallel execution of non-conflicting interactions using techniques presented in [18]. Another interesting direction is to generate correct and efficient sequential circuit given real-time software (i.e., with real-time constraints) modeled using the real-time version of BIP [2]. Finally, we will study the efficiency and the effectiveness of the generated $\mathcal{OLP}$ programs aligned with automated test case generation techniques such as [23].

References

1. BIP Website. Available at `http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html`.
2. Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Rigorous implementation of real-time systems - from theory to application. *Mathematical Structures in Computer Science*, 23(4):882–914, 2013.
3. Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P Kurshan, and Kenneth L McMillan. An analysis of sat-based model checking techniques in an industrial environment. In *Correct hardware design and verification methods*, pages 254–268. Springer, 2005.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems - FORTE 2008, 28th IFIP WG 6.1 International Conference, Tokyo, Japan, June 10-13, 2008, Proceedings*, pages 116–133, 2008.
10. Jason Baumgartner, Andreas Kuehlmann, and Jacob Abraham. Property checking via structural analysis. In *Computer-Aided Verification*, July 2002.

11. 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.
12. 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.
13. Sergey Berezin, Sérgio Campos, and Edmund M Clarke. *Compositional reasoning in model checking*. Springer, 1998.
14. Armin Biere. *Handbook of satisfiability*, volume 185. IOS Press, 2009.
15. Per Bjesse and Arne Boralv. DAG-aware circuit compression for formal verification. In *Int'l Conference on Computer-Aided Design*, Nov. 2004.
16. Per Bjesse and Arne Boralv. Dag-aware circuit compression for formal verification. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 42–49. IEEE Computer Society, 2004.
17. Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, November 2000.
18. 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.
19. Aaron R Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.
20. Aaron R Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Formal Methods in Computer Aided Design, 2007. FMCAD'07*, pages 173–180. IEEE, 2007.
21. R. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40. Springer, 2010.
22. 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.
23. Jacob Burnim and Koushik Sen. Heuristics for Scalable Dynamic Test Generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 443–446. IEEE, 2008.
24. Tony Bybell. Gtkwave electronic waveform viewer, 2010.
25. 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.
26. 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.
27. Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.
28. 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.
29. Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for dpll (t). In *Computer Aided Verification*, pages 81–94. Springer, 2006.
30. Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125–134. IEEE, 2011.
31. Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.
32. Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.
33. Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. *Software and System Modeling*, 14(1):173–199, 2015.
34. Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.
35. Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. Speculative linearizability. *Acm Sigplan Notices*, 47(6):55–66, 2012.
36. Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *FM 2006: Formal Methods*, pages 1–15. Springer, 2006.
37. 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.
38. G. Holzmann. The model checker SPIN. In *IEEE Transactions on Software Engineering*, May 1997.
39. Aaron P Hurst, Alan Mishchenko, and Robert K Brayton. Fast minimum-register retiming via binary maximum-flow. In *Formal Methods in Computer Aided Design, 2007. FMCAD'07*, pages 181–187. IEEE, 2007.
40. Mohamad Jaber. *Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP. (Implémentations Centralisée et Répartie de Systèmes Corrects par construction à base des Composants par Transformations Source-à-source dans BIP)*. PhD thesis, Joseph Fourier University, Grenoble, France, 2010.
41. Andreas Kuehlmann and Jason Baumgartner. Transformation-based verification using generalized retiming. In *Computer-Aided Verification*, July 2001.
42. Andreas Kuehlmann, Malay Ganai, and Viresh Paruthi. Circuit-based Boolean reasoning. In *Design Automation Conference*, pages 232–237, June 2001.
43. H. Mony et al. Scalable automated verification via expert-system guided transformations. In *Formal Methods in Computer-Aided Design*, November 04.
44. 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.
45. Alan Mishchenko, Michael Case, Robert Brayton, and Stephen Jang. Scalable and scalably-verifiable sequential synthesis. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 234–241. IEEE, 2008.
46. Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting a fresh look at combinational logic

synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 532–535. ACM, 2006.

47. Hari Mony, Jason Baumgartner, Viresh Paruthi, and Robert Kanzelman. Exploiting suspected redundancy without proving it. In *Design Automation Conference*. ACM Press, 2005.
48. Hari Mony, Jason Baumgartner, Viresh Paruthi, and Robert Kanzelman. Exploiting suspected redundancy without proving it. In *Proceedings of the 42nd annual Design Automation Conference*, pages 463–466. ACM, 2005.
49. 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.
50. 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.
51. Thanh-Hung Nguyen. *Constructive Verification for Component-based Systems*. University of Grenoble, 2010.
52. 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.
53. Mohamad Noureddine, Mohamad Jaber, Simon Bliudze, and Fadi A. Zaraket. Reduction and abstraction techniques for bip. In *Formal Aspects of Component Software (FACS)*, 2014.
54. 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.
55. Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, 2007.
56. Wang Qiang and Simon Bliudze. Verification of component-based systems via predicate abstraction and simultaneous set reduction. In *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Madrid, Spain, August 31 - September 1, 2015 Revised Selected Papers*, pages 147–162, 2015.
57. 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.
58. 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.
59. M. Sipser. *Introduction to the Theory of Computation*, volume 27. Thomson Course Technology Boston, MA, 2006.
60. Dong Wang. *SAT based Abstraction Refinement for Hardware Verification*. PhD thesis, Carnegie Mellon University, May 2003.

A ABC reduction and verification techniques

The ABC framework provides a set of algorithms that can be applied iteratively to (1) reduce the AIG into an equivalent AIG and (2) verify that a designated output of an AIG is always true. In what follows we provide brief descriptions of several reduction and verification ABC algorithms.

A.1 Structural register sweep (SRS)

SRS detects registers that are stuck-at-constant and eliminates them from a given sequential AIG circuit. The technique starts by zeroing up all initial values of registers in the circuit. It then uses the ternary simulation algorithm in order to detect stuck-at-constant registers. The algorithm starts from the initial values of the registers and simulates the circuit using x values for the circuit's primary inputs. The simulation algorithm stops when a new ternary state is equal to a previously computed ternary state. In this case, any register having the same constant value at each reachable ternary state will be declared to be stuck-at-constant and thus eliminated. The structural sweeping algorithm stop when no further reduction in the number of registers is possible [45].

A.2 Signal correspondence (Scorr)

Scorr uses k-step induction in order to detect and merge sets of classes of sequentially-equivalent nodes [45]. The base case for this algorithm is that the equivalence between the classes holds for the first k frames, and the inductive case is that given the base case, starting from any state, the equivalence holds in the $(k+1)^{st}$ state. Key to the signal correspondence algorithm is the way the candidate equivalences are assumed for the base case. Abc implements speculative reduction, originally presented in [48], which merges, but does not remove, any node of an equivalence class onto its representative, in each of the first k time frames. Instead of removing the merged node, a constraint is added to assert that the node and its representative are equal. This technique is claimed to decrease the number of constraints added to the SAT solved for induction.

A.3 Rewriting

Rewriting aims at finding nodes in a Directed Acyclic Graph (DAG) where by replacing subgraphs rooted at these nodes by pre-computed subgraphs can introduce important reductions in the DAG size, while keeping the functionality of these nodes intact. The algorithm traverses the DAG in depth-first post-order and gives a score for each root node. The score represents the number of nodes that would result from performing a rewrite at this node. If a rewrite exists such that the size of the DAG is decreased, such a rewrite is performed and scores are recomputed accordingly. Rewriting has been proposed initially in [16], targeted for Reduced Boolean Circuits (RBC); it was later implemented and improved for ABC in [46].

A.4 Retiming

Retiming a sequential circuit is a standard technique used in sequential synthesis, aiming at the relocation of the registers in the circuit in order to optimize some of the circuit characteristics. Retiming can either targets the minimization of the delay in the circuit, or the minimization of the number of registers given a delay constraint, or the unconstrained minimization of the number of registers in the circuit. It does so while keeping the output functionality of the circuit intact [39]

A.5 Property directed reachability (Pdr)

The Pdr algorithm aims at proving that no violating state is reachable from the initial state of a given AIG network. It maintains a trace representing a list of over-approximations of the states reachable from the initial state, along with a set of *proof-obligations*, which can be a set of bad states or a set of states from which a bad state is reachable. Given the trace and the set of obligations, the Pdr algorithm manipulates them and keeps on adding facts to the trace until either an inductive invariant is reached and the property is proved, or a counter example is found (a bad state is proven to be reachable). The algorithm was originally developed by Aaron Bradley in [19,20] and was later improved by Een et. al in [30].

A.6 Temporal induction

Temporal induction carries an inductive proof of the property over the time steps of a sequential circuit. Similar to a standard inductive proof, it consists of a base case and an inductive hypothesis. These steps are typically expressed as SAT problems to be solved by traditional SAT solvers. k-step induction strengthens simple temporal inductive proofs by assuming that the property holds for the first k time steps (states), i.e. a longer base case needs to be proven [31]. Since the target is to prove unsatisfiability (proving that the negation of the property is unsatisfiable), if the base case is satisfiable, a counter-example is returned. Otherwise, the induction step is checked by assuming that the property holds for all the states except the last one (the $(k+1)$'th state) [14].

A.7 Interpolation

Given an unsatisfiable formula $A \wedge B$, an interpolant I is a formula such that $A \implies I$, $I \wedge B$ is unsatisfiable and I contains only common variables to A and B. Given a system M, a property p and a bound k, interpolation based verification starts by attempting bounded model-checking (BMC) with the bound k. If a counter-example is found, the algorithm returns. Otherwise, it partitions the problem into a prefix pre and a suffix suf, such that the problem is the conjunction of the two. Then the interpolant I of pre and suf is computed, it represents an over-approximation of the set of states reachable in one step from the initial state of the algorithm. If I contains no new states, a fixpoint is reached and the property is proved. Otherwise, the algorithm reiterates and replaces the initial states with new states added by I [3].