Reduction and Abstraction Techniques for BIP

Mohamad Noureddine[2(✉)], Mohamad Jaber[2],
Simon Bliudze[1], and Fadi A. Zaraket[2]

[1] Ecole Polytechnique Fédérale de Lausanne, Station 14, 1015 Lausanne, Switzerland
simon.bliudze@epfl.ch
[2] American University of Beirut, Beirut, Lebanon
{man17,mj54,fz11}@aub.edu.lb

Abstract. Reduction and abstraction techniques have been proposed to address the state space explosion problem in verification. In this paper, we present reduction and abstraction techniques for component-based systems modeled in BIP (Behavior, Interaction and Priority). Given a BIP system consisting of several atomic components, we select two atomic components amenable for reduction and compute their product. The resulting product component typically contains constants and branching bisimilar states. We use constant propagation to reduce the resulting component. Then we use a branching bisimulation abstraction to compute an abstraction of the product component. The presented method is fully implemented and scales to large designs not possible to verify with existing techniques.

1 Introduction

As systems become more complex, verifying their correctness becomes harder, especially in the presence of state explosion. Researchers have proposed reduction and abstraction techniques to address this problem [11,14,17]. We discuss and compare to them in Sect. 6.

We target component-based systems (CBS) expressed in the BIP (Behavior, Interaction and Priority) framework [2]. BIP uses a dedicated language and toolset supporting a rigorous design flow. The BIP language allows to build complex systems by coordinating the behavior of a set of atomic components. Behavior is described with Labeled Transition Systems (LTS) where transitions are annotated with data and functions written in C. Coordination between components is layered. The first layer describes the interactions between components that allow synchronization and data transfer. The second layer describes dynamic priorities between the interactions to express scheduling policies. The combination of interactions and priorities characterizes the overall architecture of a system. Moreover, BIP has rigorous operational semantics: the behavior of a composite component is formally described as the composition of the behaviors

Mohamad Noureddine – The presented work was partially realised while this author was at EPFL for a summer internship. The authors are listed alphabetically.

© Springer International Publishing Switzerland 2015
I. Lanese and E. Madelaine (Eds.): FACS 2014, LNCS 8997, pp. 288–305, 2015.
DOI: 10.1007/978-3-319-15317-9_18

of its atomic components. This allows a direct relation between the underlying semantic model and its (automatically synthesized) implementation.

The BIP framework uses: (1) DFinder [3], a compositional and incremental verification tool-set, and (2) NuSMV [8] model checker, to verify BIP system invariants.

In this paper, we present reduction and abstraction techniques for component-based systems modeled in BIP. Given a BIP system consisting of several atomic components, our method automatically selects a pair of candidate components that have a high reduction potential based on their data dependencies and their synchronization. Our method uses component selection heuristics similar to [9,10,19] and provides a user-defined selection API. Then, our method computes the product of the selected pair of atomic components which renders the interaction data transfer operations into transition data transfer operations. This often uncovers opportunities for constant propagation in the product component that were hidden before. Moreover, the product operation results in product transitions of two types: (1) *non-observable* transitions involved only in a singleton interaction (i.e., a singleton interaction involves only one component) and with no actions; (2) *observable* transitions involved in either multiparty interactions or in a singleton interaction but with actions. Non-observable transitions form a branching bisimilar equivalence relation that partitions the state space. Our method detects and merges equivalence classes into representative states resulting in an abstraction of the product component.

The presented techniques are fully implemented. We evaluate our method using (1) traffic light controller case study and (2) medium to large configurations of an Automatic Teller Machine (ATM). The results show that our method drastically reduces the state space and enables verifying invariants more efficiently. In this paper, we make the following contributions:

- We abstract a BIP system with a branching bisimulation equivalence that we formalize and that leverages observable and non-observable transitions in the BIP context.
- We provide structural heuristics for selecting candidate pairs of components amenable for reduction and abstraction.
- We also provide an API for user-defined component selection criteria.
- We formalize the product operation between two BIP components and embed constant propagation in it.

The rest of the paper is structured as follows. Section 2 discusses needed background information about BIP. We present the merging and constant propagation techniques in Sect. 3, and the branching bisimulation reduction in Sect. 4. Section 5 illustrates the results obtained from verifying models reduced using our method. We summarize related work in Sect. 6 and conclude in Sect. 7.

2 BIP - Behavior, Interaction, Priority

We recall the necessary concepts of the BIP framework [2]. BIP allows to construct systems by superposing three layers of modeling: Behavior, Interaction,

and Priority. The *behavior* layer consists of a set of atomic components represented by transition systems. Atomic components are Labeled Transition Systems (LTS) extended with C functions and data. Transitions are labeled with sets of communication ports. The *interaction* layer models 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.

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*. Below, we will denote by $\mathbb{B}[X]$ the set of boolean predicates over X and by $Exp[X]$ the set of assignment statements of the form $X := f(X)$.

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

Definition 2 (Atomic component). *An atomic component B is defined as a tuple (P, L, X, T), where P is a set of ports,*[1] *L is a set of control locations, X is a set of variables and $T \subset L \times P \times \mathbb{B}[X] \times Exp[X] \times L$ is a transition relation, such that, for each transition $\tau = (l, p[X_p], g_\tau, f_\tau, l') \in T$, $g_\tau \in \mathbb{B}[X]$ is a Boolean guard over X and $f_\tau \in Exp[X]$ is a partial mapping associating to some $x \in X$ the corresponding statement $f_\tau^x(X)$.*

For $\tau = (l, p[X_p], g_\tau, f_\tau, l') \in T$ a transition of the 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. Transition τ 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 $\tau = (l, p[X_p], g_\tau, f_\tau, l')$, $\tau.src$, $\tau.port$, $\tau.guard$, $\tau.func$, 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 $\mathrm{var}(f_\tau) = \{x \in X \mid x := f^x(X) \in f_\tau\}$. 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. We denote by $\mathbf{X}$ the set of valuations of the variables X.

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

Definition 3 (Semantics of Atomic Components). *The semantics of the atomic component $B = (P, L, X, T)$ is defined as the LTS $S_B = (Q_B, P_B, \rightarrow)$,*[2] *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 and $\rightarrow = \big\{\big((l, v), p(v_p), (l', v')\big) \mid$*

[1] All sets are finite.

[2] Here and below, we omit the index on $\rightarrow$, since it is always clear from the context.

$\exists \tau = (l, p[X_p], g_\tau, f_\tau, l') \in T : g_\tau(v) \wedge v'$ *is equal to the value of* $f_\tau(v_p/v)\big\}$*, where* v_p *is a valuation of the variables* X_p*.*

Transition $(l, v) \xrightarrow{p(v_p)} (l', v')$ is possible iff there exists a transition $(l', p[X_p], g_\tau, f_\tau, l)$, such that $g_\tau(v') = \texttt{true}$. As a result, the valuation of variables X is updated to $v' = f_\tau(v_p/v)$, i.e. the values of variables X_p are updated to v_p before the application of f.

Creating Composite Components. Atomic components interact by synchronizing transitions. Upon synchronization, data values can be transferred between components.

Definition 4 (Interaction). *An interaction* a *is a tuple* (P_a, G_a, F_a)*, where* $P_a \subseteq \bigcup_{i=1}^{n} B_i.P$ *is a nonempty set of ports that contains at most one port of every component, that is,* $\forall i : 1 \leq i \leq n : |B_i.P \cap P_a| \leq 1$*. Denoting by* $X_a = \bigcup_{p \in P_a} X_p$ *the set of variables available to* a*,* $G_a \in \mathbb{B}[X_a]$ *is a boolean guard and* $F_a : \mathbf{X_a} \rightarrow \mathbf{X_a}$ *is an update function.* P_a *is the set of connected ports called the support set of* a*.*

Definition 5 (Semantics of Composite Components). *Let* $\mathcal{B} = \{B_1, \ldots, B_n\}$ *be a set of atomic components with their respective semantic LTS* $S_{B_i} = (Q_{B_i}, P_{B_i}, \rightarrow)$ *(recall, Definition. 3, that the states* Q_{B_i} *and labels* P_{B_i} *comprise valuations of data variables); let* γ *be a set of interactions. The composition of* $\mathcal{B}$ *with* γ *is the LTS* $\gamma(\mathcal{B}) = (Q, \gamma, \rightarrow)$*, where* $Q = Q_{B_1} \times \ldots \times Q_{B_n}$ *and* $\rightarrow$ *is the least set of transitions satisfying the following rule*

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

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

The meaning of the above rule is the following: if there exists an interaction a such that all its ports are enabled in the current state and its guard evaluates to `true`, then the interaction can be fired. When a is fired, all involved components evolve according to the interaction and not involved components remain in the same state.

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

Definition 6 (System). *A BIP system* $\mathcal{S}$ *is a tuple* $(B, Init, v)$ *where* B *is a composite component,* $Init \in B_1.L \times \ldots \times B_n.L$ *is the initial state of* B*, and* $v \in \mathbf{X^{Init}}$ *where* $X^{Init} \subseteq \bigcup_{i=1}^{n} B_i.X$*.*

3 Merging and Constant Propagation

In this section we present two techniques *merging* and *constant propagation* in order to reduce the state space of the system. First, we select candidate components for merging based on a set of heuristics. Then, we apply a constant propagation technique [16] that will reduce the state space by removing some data variables.

3.1 Merging Components

Throughout this paper, we assume that the input BIP systems have no priority rules and that all the automata in the BIP atomic components are complete. An automaton is complete iff in any location, the disjunction of guards of the outgoing transitions evaluates to true.

Let $\mathcal{B}$ be a set of atomic components and γ be an interaction model. Consider two atomic components $B_1, B_2 \in \mathcal{B}$ and denote $I(B_1, B_2) = \{a \in \gamma \mid P_a \subseteq B_1.P \cup B_2.P\}$ the set of interactions involving only B_1 and B_2; $P_{12} = \bigcup_{a \in I(B_1,B_2)} P_a$ the set of all ports that are part of some interaction $a \in I(B_1, B_2)$ and p_a a new port corresponding to an interaction $a \in I(B_1, B_2)$. Assume further that all ports involved in interactions between B_1 and B_2 are not involved in interactions with any other atomic components, i.e., for any $a \in \gamma$, either $a \in I(B_1, B_2)$, or $P_a \cap P_{12} = \emptyset$.

Definition 7 (Product Component). *Let B_1 and B_2 be two atomic components as above. Their product is an atomic component $prod(B_1, B_2) = (P, L, X, \rightarrow)$, where:*

- $P = B_1.P \cup B_2.P \cup \{p_a \mid a \in I(B_1, B_2)\} \setminus P_{12}$,
- $X = B_1.X \cup B_2.X$,
- $\rightarrow$ *is the minimal transition relation induced by the following rules,*

$$\frac{a \in I(B_1, B_2) \quad P_a = \{p_1, p_2\} \quad l_1 \xrightarrow{(p_1, g_1, f_1)} l_1' \quad l_2 \xrightarrow{(p_2, g_2, f_2)} l_2'}{(l_1, l_2) \xrightarrow{\left(p_a,\ g_1 \wedge g_2 \wedge G_a,\ (f_1 \cup f_2) \circ F_a\right)} (l_1', l_2')},$$

$$\frac{\{i, j\} = \{1, 2\} \quad p \in B_i.P \setminus P_{12} \quad l_i \xrightarrow{(p, g, f)} l_i' \quad l_j' = l_j}{(l_1, l_2) \xrightarrow{(p, g, f)} (l_1', l_2')},$$

- $L = \left\{(l_1, l_2) \in B_1.L \times B_2.L \mid \exists (l_1', l_2') \in B_1.L \times B_2.L : (l_1, l_2) \rightarrow (l_1', l_2')\right\}$.

Informally, the product component operation is a syntactic analogue of the semantic *parallel composition* operation of two atomic components as defined in Definition 5.

Definition 8 (Component Merging). *Let $\gamma(\mathcal{B})$ be a composite component and B_1, B_2 be atomic components as above. We define the component merging operation $merge(\gamma(\mathcal{B})) = \gamma'(\mathcal{B}')$ where*

– $\mathcal{B}' = \mathcal{B} \cup \{prod(B_1, B_2)\} \setminus \{B_1, B_2\}$, *and*
– $\gamma' = \big(\gamma \setminus I(B_1, B_2)\big) \cup \big\{ \big(\{p_a\}, \texttt{true}, \texttt{id}\big) \,\big|\, a \in I(B_1, B_2) \big\}$

By Definition 5 system $\gamma'(\mathcal{B}')$ is semantically equivalent to $\gamma(\mathcal{B})$ since the product component $prod(B_1, B_2) \in \mathcal{B}'$ from Definition 7, where B_1 and B_2 are atomic components in $\mathcal{B}$, is semantically equivalent to a *parallel composition*.

Several heuristics for selecting candidate components have been presented for LTS systems with no data transfer [9,10,19]. These heuristics consider merging pairs of components and favor the ones that result in smaller components. Our method *iteratively* selects and merges candidate components for merging based on a set of heuristics that take data transfer and component synchronization into consideration.

The first heuristic favors the pairs of components with the largest amount of data transfer. Intuitively, larger data transfer operations offer more room for stuck at constant variables that we detect and eliminate using constant propagation. The width of the data transfer considers the type of data variables (integers are wider than Boolean variable). The second heuristic favors the components that are highly synchronized since they produce more compact products. Intuitively, the product of highly synchronized components results in a large unreachable state space that is easily detected. Both are structural analysis heuristics and take polynomial running time with respect to the size of the system.

Our method also supports a merge selection API that users can implement to rank candidate component pairs. The API passes a pair of components to the user implementation and the implementation evaluates the pair and returns a merging rank value. The pair with highest rank is considered for merging.

Product components can be very large which is a problem to compositional model checking techniques in DFinder and NuSMV. Our method considers the product of components B_1 and B_2 if the maximum number of possible transitions of $B_1 \times B_2$ is smaller than a threshold value $n_1 \times n_2 \leq n_t$ where n_1 and n_2 are the number of transitions in B_1 and B_2, respectively.

3.2 Constant Propagation

We remove stuck-at-constant data variables by following the basic definition of constant propagation from [16,20]. A variable is stuck-at-constant if it is constant on all possible control locations of an atomic component. Applying constant propagation as an intermediate step in compositional verification tools is not novel and is a well established technique [12,15]. To the best of our knowledge, we are the first to apply it in the context of BIP systems.

Given a composite component $\gamma(\{B_i\}_{i \in I})$ and an atomic component B_k such that $k \in I$, we construct the composite component $\gamma(\{B_j\}_{j \in J}, B'_k)$ where $J = I \setminus \{k\}$ and B'_k is an atomic component such that $B'_k.L = B_k.L$, $B'_k.X = B_k.X \setminus X_c$ where $X_c \subseteq B_k.X$ is the set of stuck-at-constant variables in B_k, and $B'_k.T = T'$ where T' is T with each constant variable $x \in X_c$ is replaced by its constant value appropriately.

Algorithm 1. Algorithm for building CFG of an atomic component

$l_0 \leftarrow$ initial location of B
Create vertex v_{init} s.t. $v_{init}.l = l_0$ and $v_{init}.f =$ initial valuation of $B.X$
Create vertex v_{l_0} s.t. $v_{l_0}.l = l_0$ and $v_{l_0}.f = \phi$
Create vertex (v_{init}, v_{l_0})
$Vertices[l_0] \leftarrow v_{l_0}$
$stack.push(l_0)$
while $\neg stack.isEmpty()$ **do**
  $l \leftarrow stack.pop()$
  Set l as visited
  $v_l \leftarrow Vertices[l]$
  for all τ s.t. $\tau.src = l$ **do**
    Create vertex v s.t. $v.l = l$ and $v.f = \tau.f$
    Create edge (v_l, v)
    if $\tau.dest$ visited **then**
      Create edge $(v, Vertices[\tau.dest])$
    else
      Create vertex v_{dest} s.t. $v.dest.l = \tau.dest$ and $v_{dest}.f = \phi$
      Create edge (v, v_{dest})
      $Vertices[\tau.dest] \leftarrow v_{dest}$
      $stack.push(\tau.dest)$
    end if
  end for
end while

Definition 9 (Control Flow Graph). *The control flow graph (CFG) of an atomic component B is a directed graph (V, E) where:*

- *V is a set of vertices, each representing a control location $l \in B.L$ and a set of computational steps in B. We denote by $v.l$ and $v.f$ the control location and the set of computational steps in $v \in V$, respectively.*
- *E is a set of edges, such that $(v_1, v_2) \in E$ iff $\exists(\tau \in B.T).\ (v_1.l = \tau.src \wedge v_2.l = \tau.dest)\ \wedge\ (v_1.f = \tau.f \vee v_2.f = \tau.f)$.*

Listing 1 shows the algorithm used for constructing the CFG of an atomic component B. We create an empty vertex for each control location in B. We denote by v_l the empty vertex corresponding to control location $l \in B.L$. Then we perform a depth first traversal of the LTS of B. For each control location l, for every outgoing transition τ, we create a vertex v_τ such that $v_\tau.l = l$ and $v_\tau.f = \tau.f$, and create edges (v_l, v_τ) and $(v_\tau, v_{\tau.dest})$. The empty vertices can be easily discarded, but we keep them to simplify the constant propagation step. Intuitively, this step is conservative as it abstracts away guards and may not consider some potential stuck-at-constant variables. Alternatively, symbolic computation can be used to take advantage of that.

Definition 10 (Lattice element). *A lattice element is a representation of static knowledge of the value of a variable x during the execution of a constant propagation algorithm [20]. A lattice element can have one of three types:*

– $\top$*: x is likely to have a yet to be determined constant value.*
– $\bot$*: x's value cannot be determined statically.*
– c_i*: x has the value i.*

Definition 11 (Lattice element meet). *The meet* ($\sqcup$) *operation of two lattice elements is an operation such that: (1)* $\top \sqcup any = any$*; (2)* $\bot \sqcup any = \bot$*; (3)* $c_i \sqcup c_j = c_i$ *if* $i = j$*; and (4)* $c_i \sqcup c_j = \bot$ *if* $i \neq j$*.*

Listing 2 shows the constant propagation algorithm. Given an atomic component B, we start by constructing the CFG $G(B) = (V, E)$. At each vertex $v \in V$, a variable is associated with two lattice elements, an entry element and an exit element. We initialize all variables to have the $\top$ lattice element. Variables that take part in any interaction are directly assigned the $\bot$ element since their values cannot be predicted from the component itself. Visiting a vertex v consists of computing the entry lattice elements for each variable $x \in var(v.f)$, where $var(v.f)$ is the set of variables referenced in $v.f$. This is done by performing a meet operation on the exit lattice elements of all vertices v' such $(v', v) \in E$. We then evaluate $v.f$ based on the new entry elements. The rules for the evaluation of the addition operator on lattice elements are: (1) $\top + (\top \text{ or } c_i) = \top$; (2) $\bot + any = \bot$; and (3) $c_i + c_j = c_{i+j}$. The rules for the rest of operators follow similarly.

If the evaluation of $v.f$ causes a change in the exit lattice element of any variable $x \in B.X$, all vertices v'' such that $(v, v'') \in E$ are marked for visiting. A fixed point is reached once no further exit elements are changes and no vertices are still marked for visiting. After reaching the fixed point, we form the set of stuck at constant variables $X_c = \{x : x \in B.X and\ \forall v \in V.\ Entry[v][x]\ is\ constant\}$ that have constant lattice elements at the entry of every vertex. Finally, we construct $T' = \{(l, p, g'_\tau, f'_\tau, l') \mid (l, p, g_\tau, f_\tau, l') \in T\}$ where g'_τ and f'_τ are the new guards and actions. We substitute the X_c variables with their corresponding constant values in the guards $g'_\tau = g_\tau[x \in X_c/Entry[v][x]]$ where $v.f = f_\tau$. We do the same for the actions $f'_\tau = (f_\tau \setminus \{x := f^x(X) \mid x \in X_c\})[x \in X_c/Entry[v][x]]$ but after removing the assignment statements corresponding to X_c variables.

4 Branching Bisimulation Abstraction

To cope with the increase in the number of control locations introduced by the component merging process, we apply a *branching bisimulation* based abstraction [13]. A branching bisimulation equivalence relation partitions the control locations into disjoint sets of locations that are branching bisimilar [21]. We recall the definition of branching bisimulation for LTS systems from [5,13] and apply it in the BIP context.

Definition 12 (Partition of control locations). *Given an atomic component* B*,* $\pi \subseteq 2^{B.L}$ *is partition of the set of control locations* $B.L$ *iff (1)* $\bigcup_{L \in \pi} L = B.L$*; and (2)* $\forall L', L'' \in \pi,\ L' \neq L'' \Rightarrow L' \cap L'' = \emptyset$*.*

We denote by $\pi(l)$ the block $L \in \pi$ containing the control location l.

Algorithm 2. Constant propagation algorithm

```
G ← CFG(B)
for all v ∈ G.V do
    for all x ∈ var(v.f) do
        Entry[v][x] ← Exit[v][x] ← ⊤
    end for
end for
v0 ← G.v_init
stack.push(v0)
while ¬stack.isEmpty() do
    v ← stack.pop()
    for all x ∈ var(v.f) do
        Entry[v][x] ← meet(Exit[v'][x]  ∀v' ∈ G.V s.t. (v', v) ∈ G.E)
    end for
    for all x ∈ var(v.f) do
        Exit[v][x] = evaluate(v.f)
        if Exit[v][x] changed then
            for all v'' s.t. (v, v'') ∈ E do
                stack.push(v'')
            end for
        end if
    end for
end while
```

Definition 13 (Non-observable transition). *Given a composite component* $\gamma(\{B_i\}_{i\in I})$*, an atomic component* B_k *for* $k \in I$*, a transition* $\tau = (l, p, g_\tau, f_\tau, l') \in B_k.T$ *is a non-observable transition iff (1)* $f_\tau = \emptyset$*; and (2)* $\forall a = (P_a, G_a, F_a) \in \gamma,\ p \in P_a \Rightarrow (P_a = \{p\} \wedge F_a = \emptyset)$*.*

Informally, non-observable transitions involved only in a singleton interaction and with no actions. Non-observable transitions form a branching bisimilar equivalence relation that partition the state space.

Let $\varepsilon = \{(l, l') \,|\, \exists \tau = (l, p, g_\tau, f_\tau, l') \in B_k.T \textit{ and } \tau \text{ is non-observable}\}$. The set ε^* denotes the reflexive transitive closure of ε. We use the notation $l \xrightarrow{p} l'$ for $\tau = (l, p, g_\tau, f_\tau, l') \in B.T$.

Definition 14 (Branching bisimilarity relation). *Given a composite component* $\gamma(\{B_i\}_{i\in I})$*, an atomic component* B_k *for* $k \in I$*, a relation* $\mathfrak{B} = B_k.L \times B_k.L$ *is a branching bisimilarity relation on* B_k *iff:*

- $\mathfrak{B}$ *is symmetric*
- *Given* $l, \ell \in B_k.L$*,* $(l, \ell) \in \mathfrak{B}$ *iff* $\forall l, l_1,\ l \xrightarrow{p} l_1$*,*

$$\begin{cases} (l, l_1) \in \varepsilon \wedge (l_1, \ell) \in \mathfrak{B} \\ \vee \\ \exists (l_2, \ell_1 \in B_k.L),\ (\ell_1 \xrightarrow{p} l_2) \wedge (\ell, \ell_1) \in \varepsilon^* \wedge (l, \ell_1) \in \mathfrak{B} \wedge (l_1, l_2) \in \mathfrak{B} \end{cases}$$

We write $l_1 \sim l_2$ when $(l_1, l_2) \in \mathfrak{B}$. We denote by π^b the partition under the branching bisimilarity equivalence.

Definition 15 (Quotient branching bisimilar component). *Given a composite component $\gamma(\{B_i\}_{i\in I})$, and an atomic component B_k for $k \in I$, let $\mathfrak{B}$ be the largest branching bisimulation relation over B_k, an atomic component B is the quotient branching bisimilar component of B_k iff B is the atomic component with the smallest number of states such that*

1. $B.L = \pi^b$
2. $B.T = \{(\pi^b(l), p, g, f, \pi^b(l')) \mid (l, p, g, f, l') \in B_k.T \wedge \big((l \sim l') \Rightarrow ((l, l') \notin \varepsilon)\big)\}$
3. $B.P = \{p \mid \exists \tau = (l, p, g, f, l') \in B.T\}$.

Definition 16 (Branching bisimulation abstraction). *Given a composite component $\gamma(\{B_i\}_{i\in I})$ and an atomic component B_k for $k \in I$, we define the branching bisimulation abstraction operation*

$$abstract(k, \gamma(\{B_i\}_{i\in I})) = \gamma(\{B_i\}_{i\in I})|_{B_k=B}$$

where B_k is replaced by its quotient branching bisimilar component B.

Construction. We follow the signature refinement approach for branching bisimulation abstraction as presented in [5,21]. It is based on computing a *signature* for each control location $l \in B.L$. At the end of the algorithm, control locations with the same signature $sig(l)$ are bisimilar with respect to the branching bisimilarity relation $\mathfrak{B}$. Given an atomic component B, we start from an initial partition $\pi^0 = \{B.L\}$. We then keep refining the partition π w.r.t. $\mathfrak{B}$ until a fixed point is reached and we are left with a minimal partition π^b of $B.L$.

Definition 17 (Branching bisimulation signature function). *Given an atomic component B and a partition π of $B.L$, the branching bisimulation signature function of a control location $l \in B.L$ is defined as: $sig(l) = \{(p, \pi(l_1)) \mid \exists l_2 \in B.L \text{ s.t. } (l, l_2) \in \varepsilon^* \ \wedge\ l_2 \xrightarrow{p} l_1 \ \wedge\ \big((l_2, l_1) \notin \varepsilon \vee \pi(l_1) \neq \pi(l)\big)\}$.*

Listing 3 shows the algorithm we used for computing the minimal partition π^b of the control locations $B.L$; it is a direct adaptation of the single threaded algorithm presented in [5]. Constructing the quotient atomic component from the computed partition is a direct translation of Definition 15.

Correctness. As noticed in [4], a straightforward consequence of Bloom's results [6] is that composition with sets of interactions (Definition 5), called "BIP glue operators" in [4], preserves bisimilarity. Since the only transitions that we consider non-observable in this paper do not modify the data variables of atomic component and do not participate in any of the interactions, it follows that branching bisimilarity is preserved by composition with interactions.

Algorithm 3. Branching bisimulation abstraction algorithm

```
BranchingBisimilarityAbstraction(B)
π′ ← π ← B.L
repeat
    π ← π′
    π′ ← refinePartition(B, π)
until π′ ← π
```

```
refinePartition(B, π)
for all l ∈ B.L do
    sig ← ∅
    for all l′ ∈ B.L s.t. (l, p, g, f, l′) ∈ B.T do
        if ((l, p, l′) ≠ ε) ∨ (π(l′) ≠ π(l)) then
            sig ← sig ∪ (p, π(l′))
        end if
    end for
    insertSignature(B, π, l, sig)
end for
return {{l′ ∈ B.L | sig(l′) = sig(l)} | l ∈ B.L}
```

```
insertSignature(B, π, l, sig)
sig(l) ← sig(l) ∪ sig
for all l′ ∈ B.L s.t. l′ -ε-> l do
    if π(l) = π(l′) then
        insertSignature(B, π, l′, sig)
    end if
end for
```

The branching bisimulation abstraction introduces new behaviors as follows. Observable transitions are allowed to introduce changes to the state of the component by changing the values of the internal variables. The branching bisimilarity relation only considers ports as transition labels and ignores differences in actions. Thus grouping locations and building the quotient component may introduce new sequences of transitions, especially in the cases where guards on the transitions are not mutually exclusive. Nevertheless, the interactions between the different components in the system are preserved, i.e., synchronization between the components is not affected.

5 Results

We illustrate our method using a traffic light controller case study and evaluate it using several configurations of medium to large ATM systems. We use the NuSMV [8] model checker to verify deadlock freedom of the BIP systems before and after reduction. We report on the number of BDD nodes allocated, and on the execution time taken to perform the verification. Moreover, we also report on the execution time taken by DFinder [3] to prove the deadlock freedom of

the ATM design. All experiments are run on a machine with an Intel Core i7 processor and 4 GB of physical memory. We set a time-out for verification of 5000 seconds, and do not set a limit on the memory usage other than the physical limit of the machine. We use the default configuration of NuSMV and do not add any further optimizations. We use the command `check_fsm` to verify deadlock freedom of the designs.

5.1 Traffic Light Controller

Figure 1 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 a between the two components.

The interaction a between the components creates a data dependence between the two. This data dependence hides the fact that the variable n has a constant value at each location in the timer component. Figure 2a shows the product component of the light and timer components. Since the `done` ports of the two components are synchronized, we replace them by a single port `donedone`. Subsequently, the interaction a is replaced by an interaction a' based solely on the newly created port. Note that this synchronization between the two `done` ports renders some transitions in the product automaton obsolete, i.e. they can never be taken and are thus removed.

Performing constant propagation on the resulting product yields to detecting that both the variables n and m are constants at each control location. We replace these variables by their constant values at each control location, and remove them from the component as shown in Fig. 2b.

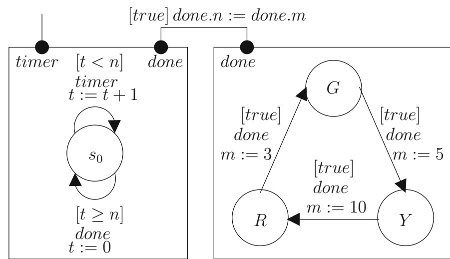
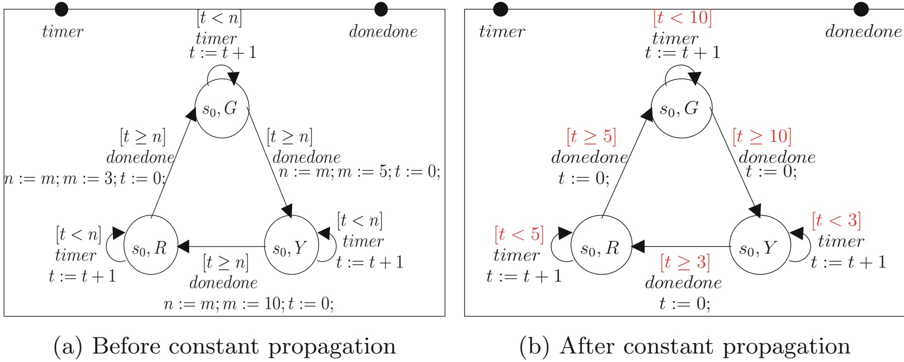


Fig. 1. Traffic light in BIP

Table 1 shows the results of running NuSMV on the translated BIP models before and after applying our reduction techniques. The *Locations* and *Transitions* columns show the total number of control locations and transitions in the



(a) Before constant propagation (b) After constant propagation

Fig. 2. The product of the timer and light components

Table 1. Results for traffic light controller

Before reduction				After reduction			
Locations	Transitions	BDD nodes	Time(s)	Locations	Transitions	BDD nodes	Time(s)
4	5	8589	0.0088	1	6	1425	0.0016

BIP system, respectively. The *BDD Nodes* and *Time* columns show the number of allocated BDD nodes and the time taken for verification, respectively. Using branching bisimulation reduction, we are able to reduce the number of control locations from 4 to a single location. Although the component merging operation introduced an increase in the number of transitions, this addition did not affect neither the number of allocated BDD nodes nor the verification time. Our method reduced the verification time by a factor of 5 and the number of allocated BDD nodes by a factor of 6.

5.2 Automatic Teller Machine

An ATM is a computerized system that provides financial services for users in a public space. Figure 3 shows a structured BIP model of an ATM system adapted from the description provided in [7]. The system is composed of four atomic components: (1) the User, (2) the ATM, (3) the Bank Validation and (4) the Bank Transaction. It is the job of the ATM component to handle 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 his card and enter the confidential code. The user has 5 time units to enter the code before the counter expires and the card is ejected by the ATM. Once the code is entered, the ATM checks with the bank validation unit for the correctness of the code. If the code is invalid, the card is ejected and no transaction occurs. If the code is valid, the ATM waits for the user to enter the desired amount of

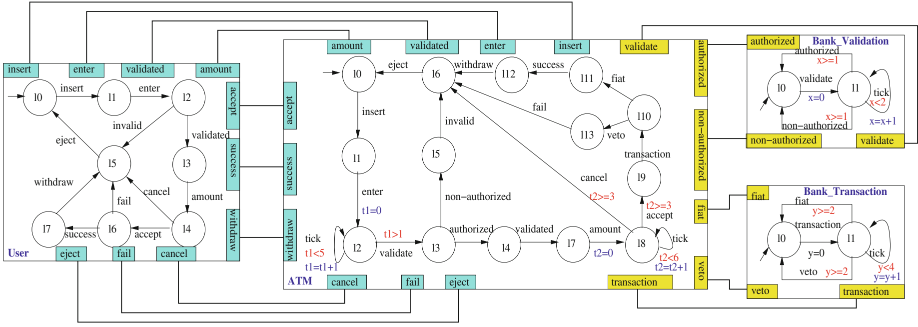


Fig. 3. Modeling of ATM system in BIP

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 any additional users. In our model, we assume the presence of a single bank and multiple ATMs and users.

Table 2 shows the improvement obtained by applying our reduction method on the ATM design for a number of ATMs ranging from 2 to 50. We show the number of control locations and transitions before and after applying our reduction method. We also present the number of allocated BDD nodes and the verification time in seconds in each case for NuSMV, and the verification time taken by DFinder to prove deadlock freedom. Note that in all cases, the results were conclusive and no spurious counter-examples were generated. Our method reduced by 3 times the number of control locations in the design and by 2 times the number of transitions. Under NuSMV, it introduced large improvements in both the number of allocated BDD nodes and the verification time, achieving

Table 2. Results for ATM system

						NuSMV				DFinder	
	Locations		Transitions		Reduction	BDD nodes		Time (s)		Time (s)	
ATMs	Orig.	Red.	Orig.	Red.	Time (s)	Orig.	Red.	Orig.	Red.	Orig.	Red.
2	50	18	68	32	0.066	977,712	542,901	1.4	0.2	3	2
3	73	25	98	44	0.073	6,183,118	921,076	142.6	10.3	4	3
4	96	32	128	56	0.079	18,630,028	1,893,192	3,360.9	281.3	6	4
5	119	39	158	68	0.086	N/A	N/A	N/A	N/A	7	5
10	234	74	308	128	0.133	N/A	N/A	N/A	N/A	24	8
50	1,154	354	1,508	608	0.472	N/A	N/A	N/A	N/A	267	37

10 times reduction for the case of 4 ATMs and 4 users. For number of ATMs and users higher than 4, NuSMV reached the time-out limit for both designs. As for DFinder, our method achieved high improvement reaching a speedup of 10 in the case of 50 ATMs and users. Note that in all cases, the time needed to reduce the designs is negligible as shown in Table 2.

6 Related Work

Much work has been done on the automatic compositional reduction of communicating processes [1,9,10,19]. The techniques revolve around incrementally composing and minimizing individual components of an input system modulo an equivalence relation. Most of the techniques focus on finding heuristics for selecting components to be composed in a way that minimizes the size of the largest intermediate composed component.

The work in [9] presents a comparative study of three component selection heuristics. The first is proposed in [19] and aims at finding components such that the number of transitions that can be removed (hidden) after their parallel composition is as high as possible. The authors in [9] improve on the heuristics defined in [19] by introducing metrics to estimate the number of the transitions that can be removed after parallel composition. Our transformations can make use of the aforementioned heuristics to select candidate components for merging. In fact, our supporting tool provides an easy to use programming interface for adding and testing selection heuristics.

The work in [10] uses the concept of networks of LTSs introduced in [18] to support compositional reduction using different compositional operators. The authors use a heuristic similar to the ones presented in [9] to estimate the number of internal transitions that can be removed after applying the composition operators, and compare the obtained metric for possible compositions. Our technique differs from the work in [10] in that our transformations are solely targeted towards BIP systems, and need not be as general as the techniques presented in [10].

Additionally, the idea of computing the product of communicating finite state machines and then reducing them using a notion of state equivalence is presented in [1]. The authors propose a method to iteratively multiply the components of a given design and reduce the product at each iteration using a notion of input-output equivalence. This leads to the construction of a minimal product finite state machine representing the entire input system, on which verification is to be performed. We follow a similar approach to that presented in [1], but we do not compute the product of the entire system, component merging is based on a set of user defined heuristics and is done while considering the state explosion introduced by the product operation.

In [11], the authors address the problem of using program analysis in order to assist reduction techniques, mainly *symmetry reduction*, in limiting state-space explosion in systems composed of multiple communicating processes. Such a

system is symmetric if its transition relation is invariant under some given permutations of the communicating processes. States in the system that are identical up to these permutations are considered equivalent, and lead to generating a reduced system that is bisimilar to the original system. The authors argue that symmetry reduction is affected by local state explosion in the each of the processes, and propose the usage of static analysis techniques such as *static local reachability analysis* in order to benefit the efficiency of symmetry reduction. In our work, we also make use of constant propagation, a static program analysis technique, in the benefit of reducing the number of internal variables and thus help the model checker in deciding the problem.

Graf and Steffen [14] focus on presenting a compositional minimization technique for finite state concurrent systems. This technique makes use of *interface specifications* to remove unreachable transitions of the system. Interface specifications are provided by the user and are used to define sets of observable sequences at the interfaces between communicating processes. The authors present a method that takes interface specifications into consideration when performing iterative composition and minimization, thus avoiding the state-space explosion at the intermediate composition levels. We resemble the aforementioned approach in that we consider port synchronization between components when performing merging, thus leading to removing unreachable transitions from the product component.

Compositional minimization via static analysis (CMSA) [22] selects candidate components for minimization using a mincut based algorithm such that the number of component outputs is significantly smaller that the number of inputs. CMSA then partitions the state space into equivalence classes relevant to the outputs, selects representative states of the equivalence classes, and computes a reduced circuit using a bisimulation based transformation that targets state space reduction. CMSA is applicable to circuit designs only. Our method differs in that it works on BIP systems, and resembles CMSA in that it considers merged components as candidate components, and applies a branching bisimulation abstraction with respects to the ports of the resulting product component.

7 Conclusion

Our work makes contributions to efficiently verify component-based systems modeled in BIP. First, we select pairs of components amenable for reduction and abstraction using structural heuristics. Then we merge the selected components using a product operation and we reduce the resulting component using constant propagation. Finally, we use abstraction techniques based on merging branching bisimilar states in order to reduce the size of the system and its complexity. Spurious counterexamples are detected by translating the counterexample to the original system and simulating it. Our contributions are complementary to tools that are used to verify BIP systems such as DFinder and BIP-to-NuSMV. Our reduction and abstraction techniques are completely implemented in a supporting tool that provides an API to specify user defined merging heuristics.

In the future, we plan to extend our work to handle priorities in the BIP context. We also plan to define feedback guidance to refine the abstraction in case a spurious counterexample is generated. We also plan to use the component selection heuristics defined in the literature in our tool, compare them on different designs and propose new heuristics that are targeted towards the efficient compositional reduction of BIP systems.

References

1. Aziz, A., Singhal, V., Swamy, G., Brayton, R.K.: Minimizing interacting finite state machines: a compositional approach to language to containment. In: ICCD, pp. 255–261 (1994)
2. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the bip framework. IEEE Softw. **28**(3), 41–48 (2011)
3. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: D-Finder: a tool for compositional deadlock detection and verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009)
4. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008)
5. Blom, S., Orzan, S.: Distributed branching bisimulation reduction of state spaces. Electron. Notes Theor. Comput. Sci. **89**(1), 99–113 (2003). PDMC 2003, Parallel and Distributed Model Checking (Satellite Workshop of CAV 2003)
6. Bloom, B.: Ready simulation, bisimulation, and the semantics of CCS-like languages. Ph.D. thesis, Massachusetts Institute of Technology (1989)
7. Chaudron, M., Eskenazi, E., Fioukov, A., Hammer, D.: A framework for formal component-based software architecting. In: OOPSLA, pp. 73–80 (2001)
8. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
9. Crouzen, P., Hermanns, H.: Aggregation ordering for massively compositional models. In: 2010 10th International Conference on Application of Concurrency to System Design (ACSD), pp. 171–180. IEEE (2010)
10. Crouzen, P., Lang, F.: Smart reduction. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 111–126. Springer, Heidelberg (2011)
11. Emerson, E.A., Wahl, T.: Efficient reduction techniques for systems with many components. Electr. Notes Theor. Comput. Sci. **130**, 379–399 (2005)
12. Garavel, H., Sifakis, J.: Compilation and verification of lotos specifications. PSTV **10**, 359–376 (1990)
13. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996)
14. Graf, S., Steffen, B.: Compositional minimization of finite state systems. In: CAV, pp. 186–196 (1990)
15. Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. In: Ponse, A., Verhoef, C., van Vlijmen, S.F.M. (eds.) Algebra of Communicating Processes, pp. 26–62. Springer, London (1995)

16. Kildall, G.A.: A unified approach to global program optimization. In: POPL 1973, pp. 194–206. ACM, New York (1973)
17. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 846–862. Springer, Heidelberg (2013)
18. Lang, F.: Exp.Open 2.0: a flexible tool integrating partial order, compositional, and on-the-fly verification methods. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
19. Tai, K.C., Koppol, P.V.: Hierarchy-based incremental analysis of communication protocols. In: Proceedings of the 1993 International Conference on Network Protocols, 1993, pp. 318–325. IEEE (1993)
20. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. **13**(2), 181–210 (1991)
21. Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref – a symbolic bisimulation tool box. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 477–492. Springer, Heidelberg (2006)
22. Zaraket, F.A., Baumgartner, J., Aziz, A.: Scalable compositional minimization via static analysis. In: ICCAD, pp. 1060–1067 (2005)