Formal Verification of Infinite-State BIP Models

Simon Bliudze[1], Alessandro Cimatti[2], Mohamad Jaber[3], Sergio Mover[2], Marco Roveri[2], Wajeb Saab[1], and Qiang Wang[1(✉)]

[1] École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
qiang.wang@epfl.ch
[2] Fondazione Bruno Kessler, Trento, Italy
[3] American University of Beirut, Beirut, Lebanon

Abstract. We propose two expressive and complementary techniques for the verification of safety properties of infinite-state BIP models. Both our techniques deal with the full BIP specification, while the existing approaches impose considerable restrictions: they either verify finite-state systems or they do not handle the transfer of data on the interactions and priorities.

Firstly, we propose an instantiation of the ESST (Explicit Scheduler Symbolic Thread) framework to verify BIP models. The key insight is to apply symbolic reasoning to analyze the behavior of the system described by the BIP components, and an explicit-state search to analyze the behavior of the system induced by the BIP interactions and priorities. The combination of symbolic and explicit exploration techniques allow to benefit from abstraction, useful when reasoning about data, and from partial order reduction, useful to mitigate the state space explosion due to concurrency.

Secondly, we propose an encoding from a BIP model into a symbolic, infinite-state transition system. This technique allows us to leverage the state of the art verification algorithms for the analysis of infinite-state systems.

We implemented both techniques and we evaluated their performance against the existing approaches. The results show the effectiveness of our approaches with respect to the state of the art, and their complementarity for the analysis of safe and unsafe BIP models.

1 Introduction

BIP [2,4] is a framework for the component-based design of complex concurrent systems that is being actively used in many industrial settings [3,5]. The verification of BIP plays a crucial role in the *Rigorous System Design* methodology [28], where a correct implementation of the system is obtained by a series of transformations from its high-level model; proving that a property holds in the model will ensure that it holds in the implementation.

This work was carried out within the D-MILS project, which is partially funded under the European Commission's Seventh Framework Programme (FP7).

B. Finkbeiner et al. (Eds.): ATVA 2015, LNCS 9364, pp. 326–343, 2015.
DOI: 10.1007/978-3-319-24953-7_25

Despite the importance of verifying BIP models, the existing approaches (e.g. implemented in tools like DFinder [7],VCS [20] and Bip2Uppaal [29]) impose considerable restrictions on the models that can be analyzed. In particular, only DFinder verifies models with infinite-state data variables. However, DFinder does not consider the data transfer on interactions, an essential feature to express that data is exchanged among the components (consider that in BIP the components cannot share variables), and the priorities among interactions.

In this paper, we focus on the safety property verification of infinite-state BIP models, and we propose two techniques that are: (i) expressive enough to capture *all* the features of infinite-state BIP models (e.g. data transfer, priorities); (ii) complementary, with respect to the performance, for verifying safe and unsafe models.

The first solution is a novel verification algorithm based on the *Explicit Scheduler, Symbolic Threads* (ESST) framework [16]. The ESST extends lazy predicate abstraction and refinement [21,22] to verify concurrent programs formed by a set of cooperative threads and a non-preemptive scheduler; the main characteristic of the approach is to use lazy predicate abstraction to explore threads and explicit-state techniques to explore the scheduler. The choice of ESST is motivated by the clear separation of computations and coordination in the BIP language, which is similar to the separation of threads and scheduler in the ESST, and by the ESST efficiency, since the ESST outperforms the verification techniques based on sequentialization (e.g. in the context of SystemC and Fair Threads programs [16]). In our work, we show an *efficient* instantiation of the ESST framework in an algorithm that verifies BIP models (ESST_{BIP}). The instantiation is not trivial, and consists of defining a suitable interaction model between the threads and the scheduler, the consequent mapping of BIP components into threads, and of implementing the scheduler. Moreover, we improve the performance of our approach with several optimizations, which are justified by the BIP semantic.

In our second solution we explore a conceptually simple, but still novel, encoding of a BIP model into an infinite-state transition system. This alternative flow is motivated by the recent advancements in the verification of infinite-state systems (e.g. see [14,23]). Also this technique supports all the BIP features, like priorities and data transfer on interactions.

We provide an implementation of both approaches: the ESST_{BIP} is implemented in the Kratos [13] software model checker; the translational approach is performed using the BIP framework and then verified with the nuXmv [12] model checker. We performed a thorough experimental evaluation comparing the performance of the two techniques and of DFinder (in this case, only on the models without data transfers). The results show that the proposed approaches always perform better than DFinder, and that ESST_{BIP} and the translational approach using nuXmv are complementary, with ESST_{BIP} being more efficient in finding counterexamples for unsafe models, while the translational approach using nuXmv is more efficient in proving correctness of safe models.

This paper is structured as follows. We first provide the background of the BIP language in Sect. 2. Then in Sect. 3 we describe the ESST_{BIP} algorithm, as well as its optimizations. In Sect. 4 we show the encoding of BIP into a symbolic transition system. Then, in Sect. 5 we review the related work and in Sect. 6 we present the experimental evaluation. Finally, in Sect. 7 we draw some conclusions and outline directions for future work.

2 The BIP Model

We denote by Var a set of variables with domain $\mathbb{Z}$[1], (i.e. for all $x \in Var$, $Dom(x) = \mathbb{Z}$). An *assignment* is of the form $x := exp$, where $x \in Var$ and exp is a linear expression over Var. An *assumption* is of the form $[bexp]$, where $bexp$ is a Boolean combination of predicates over Var. Let $BExp(Var)$ be the set of assumptions and $Exp(Var)$ be the set of assignments. Let $Ops(Var) = BExp(Var) \cup Exp(Var) \cup \{skip\}$ be the set of edge operations, where *skip* denotes an operation without effects on Var. A *state* $s : Var \to \mathbb{Z}$ is a mapping from variables to their valuations; we use *State* to denote the set of all possible states. We define an evaluation function $[\![\cdot]\!]_{\mathcal{E}} : exp \to (State \to \mathbb{Z})$ for assignments and $[\![\cdot]\!]_{\mathcal{B}} : bexp \to (State \to \{true, false\})$ for assumptions. We refer to [16] for the definition of $[\![\cdot]\!]_{\mathcal{E}}$ and $[\![\cdot]\!]_{\mathcal{B}}$. We denote by $s[x := e]$ the substitution of x by e in expression s.

The BIP Syntax. An *atomic component* is a tuple $B_i = \langle Var_i, Q_i, P_i, E_i, l_{0_i} \rangle$ where Var_i is a set of variables, Q_i is a set of locations, P_i is a set of ports, $E_i \subseteq Q_i \times P_i \times BExp(Var_i) \times Exp(Var_i) \times Q_i'$ is a set of edges extended with guards and operations and $l_{0_i} \in Q_i$ is the initial location.

We assume that, for each location, every pair of outgoing edges labeled with the same port has disjoint guards. This can be achieved by simply renaming the ports and imposes no restrictions on the BIP expressiveness. We also identify a set of *error locations*, $Q_{err_i} \subseteq Q_i$ to encode the safety property[2].

Let $\mathcal{B} = \{B_1, \ldots, B_n\}$ be a set of atomic components. An *interaction* γ for $\mathcal{B}$ is a tuple $\langle Act, g, op \rangle$ such that: $Act \subseteq \bigcup_{i=1}^{n} P_i$, $Act \neq \emptyset$, and for all $i \in [1, n]$, $|Act \cap P_i| \leq 1$, $g \in BExp(\bigcup_{Bj \in \gamma_{\mathcal{B}}} Var_j)$ and $op \in Exp(\bigcup_{Bj \in \gamma_{\mathcal{B}}} Var_j)$, where $\gamma_{\mathcal{B}} = \{Bj | Bj \in \mathcal{B}, Act \cap P_j \neq \emptyset\}$.

We assume that the sets of ports of the components in a BIP model and the sets of local variables are disjoint (i.e. for all $i \neq j$, $P_i \cap P_j = \emptyset$ and $Var_i \cap Var_j = \emptyset$). For a port $\alpha \in P_i$, we identify with $id(\alpha)$ the index i of the component B_i.

Let Γ be a set of interactions, a *priority model* π of Γ is a strict partial order of Γ. For $\gamma_1, \gamma_2 \in \Gamma$, γ_1 has a lower priority than γ_2 if and only if $(\gamma_1, \gamma_2) \in \pi$. For simplicity, we write $\gamma_1 < \gamma_2$ in this case.

[1] We also consider finite domain variables (e.g. Boolean), which can be easily encoded in $\mathbb{Z}$.

[2] We can express any safety property using additional edges, interactions and error locations.

A BIP *model* $\mathcal{P}_{\text{BIP}}$ is a tuple $\langle \mathcal{B}, \Gamma, \pi \rangle$, where $\mathcal{B} = \langle B_1, \ldots, B_n \rangle$ is a set of atomic components, Γ is a set of interactions over $\mathcal{B}$ and π is a priority model for Γ.

We assume that each component B_i of $\mathcal{P}_{\text{BIP}}$ has at most an error location, without outgoing edges and such that the port of all its incoming edges is $error_i$; each $error_i$ appears in a unique singleton interaction and all such interactions have the highest priority in $\mathcal{P}_{\text{BIP}}$. Any BIP model can be put into such form (see [10]).

The BIP Semantics. A *configuration* c of a BIP model $\mathcal{P}_{\text{BIP}}$ is a tuple $\langle \langle l_1, s_1 \rangle, \ldots, \langle l_n, s_n \rangle \rangle$ such that for all $i \in [1, n]$, $l_i \in Q_i$ and $s_i : Var_i \to \mathbb{Z}$ is a state of B_i. Let $\mathcal{P}_{\text{BIP}} = \langle \mathcal{B}, \Gamma, \pi \rangle$ be a BIP model and $c = \langle \langle l_1, s_1 \rangle, \ldots, \langle l_n, s_n \rangle \rangle$ be a configuration. The interaction $\gamma = \langle Act, g, op \rangle \in \Gamma$ is *enabled* in c if, for all the components $B_i \in \mathcal{B}$ such that $Act \cap P_i \neq \emptyset$, there exists an edge $\langle l_i, Act \cap P_i, g_i, op_i, l_i' \rangle \in E_i$ and $[\![g_i]\!]_{\mathcal{B}}(s_i) = true$, and $[\![g]\!]_{\mathcal{B}}(s_1, \ldots, s_n) = true$.

A BIP model $\mathcal{P}_{\text{BIP}} = \langle \mathcal{B}, \Gamma, \pi \rangle$ can take an edge from the configuration $c = \langle \langle l_1, s_1 \rangle, \ldots, \langle l_n, s_n \rangle \rangle$ to the configuration $c' = \langle \langle l_1', s_1' \rangle, \ldots, \langle l_n', s_n' \rangle \rangle$ if there exists an interaction $\gamma = \langle Act, g, op \rangle$ such that: (i) γ is enabled in c; (ii) there does not exist an enabled interaction $\gamma' \in \Gamma$ in c such that $\gamma' > \gamma$; (iii) for all $B_i \in \mathcal{B}$ such that $Act \cap P_i \neq \emptyset$, there exists $\langle l_i, Act \cap P_i, g_i, op_i, l_i' \rangle \in E_i$ and if $op = x := exp$, $op_i = y := exp_i$ then $s_i'' = s_i[x := [\![exp]\!]_{\mathcal{E}}(s_i)]$, $s_i' = s_i''[y := [\![exp_i]\!]_{\mathcal{E}}(s_i'')]$; (iv) for all $B_i \in \mathcal{B}$ such that $Act \cap P_i = \emptyset$, $l_i' = l_i$ and $s_i' = s_i$.

We use the notation $c \xrightarrow{\gamma} c'$ to denote that there exists an edge from the configuration c to the configuration c' on the interaction γ. A configuration $c_0 = \langle \langle l_1, s_1 \rangle, \ldots, \langle l_n, s_n \rangle \rangle$ is an *initial configuration* if, for some $i \in [1, n]$, $l_i = l_{0_i}$ and, for all $i \in [1, n]$, s_i is a valuation for Var_i[3]. A configuration c is *reachable* if and only if there exists a sequence of configurations $c_0 \xrightarrow{\gamma_1} c_1 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_k} c_k$, such that c_0 is an initial configuration and $c_k = c$. A BIP model is *safe* if no error locations are reachable.

3 ESST for BIP (ESST_{BIP})

3.1 The ESST Framework

In this subsection we provide the necessary background on the ESST framework, following the presentation of [16,17].

Programming Model. The ESST framework analyzes a multi-threaded program $\mathcal{P} = \langle \mathcal{T}, \text{SCHED} \rangle$, consisting of a set of *cooperative threads* $\mathcal{T} = \langle T_1, \ldots, T_n \rangle$ and a *non-preemptive scheduler* SCHED. A non-preemptive scheduler cannot interrupt the execution of a thread, while a cooperative thread is responsible for suspending its execution and releasing the control to the scheduler.

[3] While we did not add initial predicates for Var_i, this can be encoded with an additional initial location and an edge that has as guard the initial predicates.

A *thread* $T_i = \langle G_i, LVar_i \rangle$ is a sequential program with a set of local variables $LVar_i$ and is represented by a control-flow graph (CFG) $G_i = (L_i, E_i, l_{0_i}, L_{err_i})$, where: (i) L_i is the set of locations; (ii) $E_i \subseteq L_i \times Ops(LVar_i) \times L_i$ is the set of edges; (iii) $l_{0_i} \in L_i$ is the entry location; (iv) $L_{err_i} \subseteq L_i$ is the set of *error locations*.

A *scheduler* $\textsc{Sched} = \langle SVar, F_S \rangle$ has a set of variables $SVar$ and a scheduling function F_S. For each thread T_i, the scheduler maintains a variable $st_{T_i} \in SVar$ to keep track of its status (i.e. $Running, Runnable, Waiting$). A *scheduler state* $\mathbb{S}$ is an assignment to all the variables $SVar$. Given a scheduler state $\mathbb{S}$ where no thread is $Running$, $F_S(\mathbb{S})$ generates the set of scheduler states that describes the next thread to be run. We denote by $SState$ the set of all possible scheduler states, and by $SState_{One}$ the set of scheduler states, where only one thread is $Running$. A thread can change the scheduler state by calling a *primitive function*. For example, the call to a primitive function can change the thread status from $Running$ to $Waiting$ to release the control to the scheduler.

The intuitive semantics of a multi-threaded program is the following: the program executes the thread in the $Running$ status (note that there is at most one running thread); the running thread T_i can suspend its execution, setting the variable st_{T_i} to a value different from $Running$, by calling an appropriate primitive function; when there are no running threads, the scheduler executes its scheduling function to generate a set of running threads. The next thread to run is picked non-deterministically. See [10] for a formal definition of the semantic.

The ESST Algorithm. The ESST algorithm [16,17] performs a reachability analysis of a multi-threaded program $\mathcal{P} = \langle \mathcal{T}, F_S \rangle$ using explicit-state techniques to explore the possible executions of the scheduler and lazy predicate abstraction [22] to explore the executions of the threads. In the following, we rely on the extended version of the ESST where the scheduler execution is semi-symbolic [17], since we will need a scheduler that reads and writes the local state of the threads. We provide a concise description of the reachability analysis algorithm, and refer to [16,17] for the details.

The ESST constructs an *abstract reachability forest* (ARF) to represent the reachable states. An *ARF node* is a tuple $\langle \langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, \mathbb{S} \rangle$, where for all $i \in [1, n]$, $l_i \in L_i$ is a location of T_i and φ_i is a local region (a formula over $LVar_i$), φ is a global region[4] (a formula over $\bigcup_{i \in [1,n]} LVar_i$) and $\mathbb{S}$ is a scheduler state. The ARF is constructed by expanding the ARF nodes. An ARF node can be expanded as long as it is not *covered* (no other nodes in the ARF include the set of states denoted by this node) or if it is not an error node (the node does not contain any error location). If ESST terminates and all the nodes in the ARF are covered then $\mathcal{P}$ is *safe*. If the expansion of the ARF reaches an error node, the ESST builds an abstract counterexample (a path in the ARF from the initial node to the error node), which is simulated in the concrete program;

[4] Whereas in the general ESST framework the global region is used to track both local and global variables, we use it to only track the relations among the local variables due to the data transfer on interactions.

if the simulation succeeds, we find a real counter-example and the program is *unsafe*. Otherwise the counter-example is spurious, the ESST refines the current abstraction, and restarts the expansion (see [16] for details).

The node expansion uses three basic operations: the symbolic execution of a thread (based on the *abstract strongest post-condition*), the execution of a primitive function, and the semi-symbolic execution of the scheduling function F_S. The *abstract strongest post-condition* $SP^{\delta}_{op}(\varphi)$ is the *predicate abstraction* of the set of states reachable from any of the states in the region φ after executing the operation *op*, using the set of predicates δ. ESST associates a set of predicates to thread locations ($\delta_{l'_i}$), as well as to the global region (δ). The primitive functions are executed by the primitive executor $\textsc{Sexec} : (SState \times PrimitiveCall) \rightarrow (\mathbb{Z} \times SState)$ to update the scheduler state. The scheduler function F_S is implemented by a function $F_S : ARFNodes \rightarrow (2^{SState_{One} \times LFProg})$, where $SState_{One}$ is the set of scheduler states with only one running thread, *ARFNodes* is the set of *ARF* nodes and *LFProg* is the set of loop-free programs (programs that contains assignments and conditional statements, but not loops) over the variables of $\mathcal{P}$[5]. A ARF node $\eta = (\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, \mathbb{S})$ is expanded by the following two rules:

E1. If $\mathbb{S}(st_{T_i}) = Running$ and there exists an edge $(l_i, op, l'_i) \in E_i$, create a successor node $(\langle l_1, \varphi'_1 \rangle, \ldots, \langle l'_i, \varphi'_i \rangle, \ldots, \langle l_n, \varphi'_n \rangle, \varphi', \mathbb{S}')$, where:

- $\langle \mathbb{S}', \hat{op} \rangle = \begin{cases} \langle \mathbb{S}, op \rangle & \text{if } op \text{ is not a primitive function call} \\ \langle \mathbb{S}'', x := v \rangle & \text{if } op \text{ is } x := f(\boldsymbol{y}) \text{ and } (v, \mathbb{S}'') = \textsc{Sexec}(\mathbb{S}, f(\boldsymbol{y})) \end{cases}$
- $\varphi'_i = SP^{\delta_{l'_i}}_{\hat{op}}(\varphi_i \wedge \varphi)$, $\varphi'_j = \varphi_j$, for $i \neq j$, and $\varphi' = SP^{\delta}_{\hat{op}}(\varphi)$. ($\delta_{l'_i}$ and δ are the precisions associated to the location l_i and to the global region respectively).

E2. If there are no running threads, for each $\langle \mathbb{S}', P^{lf} \rangle \in F_S(\eta)$ create a successor node $(\langle l_1, \varphi'_1 \rangle, \ldots, \langle l_n, \varphi'_n \rangle, \varphi', \mathbb{S}')$, where $\varphi'_j = SP^{\delta_{l'_j}}_{\hat{P^{lf}}}(\varphi_j \wedge \varphi)$, for $j \in [1, n]$ and $\varphi' = SP^{\delta}_{\hat{P^{lf}}}(\varphi)$.

The rule E1 expands the ARF node by unfolding the CFG edge $\langle l, op, l' \rangle$ of the running thread T_i. If the operation *op* is not a primitive function, then the scheduler state is unchanged (i.e. $\mathbb{S}' = \mathbb{S}$). Otherwise, if the operation *op* is a primitive function, (e.g. $x := f(\boldsymbol{y})$), the algorithm executes the primitive executor Sexec to change the scheduler state and collect the return value of the function (i.e. $(v, \mathbb{S}'') = \textsc{Sexec}(\mathbb{S}, f(\boldsymbol{y}))$). In both cases, the state of the running thread and the global region are updated by computing the abstract strongest post condition. The rule E2 executes the scheduling function to create a new ARF node for each output state of the scheduling function when all the threads are not running. A detailed illustration of the execution of scheduling function will be give in Sect. 3.2.

[5] The ESST framework does not allow the scheduler to produce programs with loops.

3.2 Instantiation of ESST for BIP

To instantiate ESST for BIP, there are two naïve approaches. One is to translate a BIP model to a SystemC program, hence relying on the SystemC primitive functions and the SystemC scheduler (i.e. the existing instantiation of ESST [16]). This approach is inefficient, since one has to encode the BIP semantics with additional threads. Another approach is to reuse the SystemC primitive functions as in [16,17], modifying the scheduler to mimic the BIP semantics. This approach is not efficient either, since the primitive functions in SystemC only allow threads to notify and wait for events. This has the effect of introducing additional variables in the scheduler to keep track of the sent and received events, which considerably increases the state space to be explored.

In this paper, we provide a novel instantiation of the ESST framework to analyze BIP models, it consists of: (i) a mapping from BIP to multi-threaded programs and the definition of a new primitive function *wait*() used by threads to interact with the scheduler; (ii) a new semi-symbolic scheduler that respects the BIP operational semantics and preserves the reachability of error locations.

We use the ESST version with a semi-symbolic scheduler, instead of using a purely explicit one, allowing the scheduler to read and write the state of the threads. This feature is important to analyse BIP models because, in BIP, interaction guards and effects are expressed over the global state of the system. Moreover, the semi-symbolic scheduler is also needed to correctly enforce the BIP priorities.

In each scheduling loop, the scheduler performs two tasks: (i) it computes the set of possible interactions and chooses one to be run; (ii) it schedules the execution of each thread that participates in the chosen interaction. When all the threads are in the $Waiting$ state, the scheduler computes the set of possible interactions and chooses one interaction to be run by setting the status of the participating threads to *Runnable*, and by setting the value of a local variable in the thread. The variable is used in the guards of the thread edges and encode the BIP ports. Moreover, the scheduler is also responsible for executing the global effects of the interaction. Whithin each scheduling cycle, the scheduler picks the *Runnable* threads one by one, until no such threads are available.

Primitive Functions and Threads. In our BIP instantiation of ESST we introduce a primitive function *wait()*, which suspends the execution of the calling thread and releases the control back to the scheduler (thus, we have only one primitive function). The function does not change the state of the thread, but changes the status of the thread in the scheduler state to $Waiting$. Since the return value of *wait*() is of no interest, we will write *wait*() instead of $x := wait()$. Formally, the semantics of *wait()* is defined by the primitive executor Sexec, that is $[\![wait()]\!]_{\mathcal{E}}(s, \mathbb{S}) = \textsc{Sexec}(\langle \mathbb{S}, wait() \rangle) = \langle *, \mathbb{S}' \rangle$, where $*$ denotes a dummy return value, s is the state of T_i, and $\mathbb{S}' = \mathbb{S}[st_{T_i} := Waiting]$, if T_i is the caller of *wait*().

Given an *atomic component* $B_i = \langle Var_i, Q_i, E_i, l_{0_i}, Q_{err_i} \rangle$ of $\mathcal{P}_{\text{BIP}}$, we define the *thread* $T_i = \langle G_i, LVar_i \rangle$, where $LVar_i = Var_i \cup \{evt^i\}$, $Dom(evt^i) = \mathbb{Z}$ and $G_i = (L_i, E_i, l'_{0_i}, L_{err_i})$, where:

$$
\begin{aligned}
L_i &= \{l, l_{wait} | l \in Q_i\} \cup \{l_e | e \in E_i, e = \langle l, \alpha, g, op, l' \rangle\};\\
E_i &= \{\langle l, wait(), l_{wait} \rangle | l \in Q_i\} \cup \\
&\quad \{\langle l_{wait}, evt^i = \alpha, l_e \rangle, \langle l_e, op, l' \rangle | e \in E_i, e = \langle l, \alpha, g, op, l' \rangle\};\\
l'_{0_i} &= l_{0_i};\\
L_{err_i} &= Q_{err_i}.
\end{aligned}
$$

We introduce an additional integer variable evt^i for each thread and we associate every port α to a distinct integer value; for clarity, we use the notation $evt^i = \alpha$ instead of $evt^i = i$, where $i \in \mathbb{Z}$ is the value we associated to the port α. The CFG G_i of the thread is obtained from a transformation of the BIP atomic component B_i: (i) adding a location l_{wait} and an edge from l to l_{wait} for each $l \in Q_i$; (ii) for each edge $e = \langle l, \alpha, g, op, l' \rangle \in E_i$, add an intermediate location l_e, and an edge from l_{wait} to l_e, labelled with $evt^i = \alpha$, and an edge from l_e to l', labelled with op[6].

The edge to the location l_{wait} labelled by the primitive function $wait()$ ensures that the thread releases the control to the scheduler, waiting that the scheduler chooses an interaction to be run. The subsequent edge labelled by $evt^i = \alpha$ ensures that the thread only executes the edge chosen by the scheduler and constrained by the value of the variable evt^i. Notice that the edge guard will be taken into account by the BIP scheduler.

Semi-symbolic BIP Scheduler. For analyzing BIP models, we design the semi-symbolic BIP scheduler $\textsc{Sched}(\mathcal{P}_{\text{BIP}}) = \langle F_S, SVar \rangle$, where $SVar = \{st_{T_1}, \ldots, st_{T_n}\}$, and F_S is the scheduling function that respects the BIP semantics. As required by the ESST, the scheduler keeps the status of each thread T_i in a variable st_{T_i}, with values $\{Running, Runnable, Waiting\}$. Initially all st_{T_i} are $Runnable$.

Given an ARF node $\eta = \langle \langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, \mathbb{S} \rangle$, we say that an interaction $\gamma = \langle \{\alpha_1, \ldots, \alpha_k\}, g, op \rangle$ is *enabled* if there exists a set of edges $\{t_{\alpha_1}, \ldots, t_{\alpha_k}\}$ such that, for all $i \in [1, k]$, $t_{\alpha_i} \in E_{id(\alpha_i)}$, $t_{\alpha_i} = \langle l_{id(\alpha_i)}, \alpha_i, g_{t_{\alpha_i}}, op_{t_{\alpha_i}}, l'_{t_{\alpha_i}} \rangle$. In that case, we write $enabled(\eta, \gamma)$ and we denote with $EnabledSet(\eta)$ the set of all the enabled interactions in η. Notice that the concept of enabled interaction on an ARF node is different from the one we had on a BIP configuration: we do not check the satisfiability of the guards in the ARF node to determine the set of enabled interactions. Instead, interaction guards and effects are accounted for by the symbolic execution of the scheduling function.

F_S alternates two different phases: (i) scheduling of new interactions; (ii) execution of edges participating in the chosen interaction. Given an ARF node $\eta = \langle \langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, \mathbb{S} \rangle$, $F_S(\eta)$ is defined as follows:

F1. If for all $st_{T_i} \in SVar$, such that $\mathbb{S}(st_{T_i}) = Waiting$ and $EnabledSet(\eta) = \{\gamma_1, \ldots, \gamma_k\}$, $F_S(\eta) = \{\langle \mathbb{S}_1, P_1^{lf} \rangle, \ldots, \langle \mathbb{S}_k, P_k^{lf} \rangle\}$, where for all $i \in [1, k]$:

[6] Note that, while the formal presentation introduces intermediate locations and edges, in practice these are collapsed in a single edge since we use the *large block encoding* [8].

- $\gamma_i = \langle \{\alpha_i^1, \ldots, \alpha_i^l\}, g_i, op_i \rangle$;
- $\mathbb{S}_i = \mathbb{S}[st_{T_{id(\alpha_i^1)}} := \mathit{Runnable}, \ldots, st_{T_{id(\alpha_i^l)}} := \mathit{Runnable}]$;
- $P_i^{lf} = p; g_i; g_e; op_i; evt^1_{id(\alpha_i^1)} := \alpha_i^1; \ldots; evt^l_{id(\alpha_i^l)} := \alpha_i^l$, where $p = \bigwedge_{\langle \gamma_i, \gamma' \rangle \in \pi, \alpha \in \gamma'} \bigwedge_{\langle l, \alpha, g_\alpha, op_\alpha, l' \rangle \in E_{id(\alpha)}} \neg g_\alpha$ and $g_e = \bigwedge_{\alpha \in \gamma_i} \bigvee_{\langle l, \alpha, g_\alpha, op_\alpha, l' \rangle \in E_{id(\alpha)}} g_\alpha$.

F2. If there exists a thread T_i, such that $\mathbb{S}(st_{T_i}) = \mathit{Runnable}$, then $F_S(\eta) = \{\langle \mathbb{S}[st_{T_i} := \mathit{Running}], \mathit{skip} \rangle\}$.

In rule F1, the formula p encodes the priority constraints (there are no enabled interactions with a higher priority than γ_i), and the formula g_e imposes that, in each thread that participates in the interaction, there is at least one enabled edge labeled with the corresponding interaction port. Thus, the loop free program P_i^{lf} ensures that the interaction γ_i will be scheduled, according to the BIP semantics, and also imposes the correct ports that must be executed by the threads. The rule F2 just picks the next thread to be run.

Correctness of $\mathbf{ESST_{BIP}}$.

Theorem 1. *Let* $\mathcal{P}_{\text{BIP}} = \langle \mathcal{B}, \Gamma, \pi \rangle$ *be a* BIP *model and* $\mathcal{P} = \langle \mathcal{T}, \textsc{Sched}(\mathcal{P}_{\text{BIP}}) \rangle$ *be the corresponding multi-threaded program with semi-symbolic* BIP *scheduler* $\textsc{Sched}(\mathcal{P}_{\text{BIP}})$. *If the* ESST_{BIP} *algorithm terminates on* $\mathcal{P}$*, then the* ESST_{BIP} *returns* safe *iff the* BIP *model* $\mathcal{P}_{\text{BIP}}$ *is* safe.

For lack of space, we provide the proofs in the extended technical report [10].

3.3 Optimizations

In this section we present some optimizations aiming to reduce the number of the ARF nodes that must be explored during the reachability analysis.

Partial Order Reduction for BIP. The application of POR to the ESST_{BIP} is based on the following idea: when the ESST_{BIP} executes the scheduling function F_S on a node η, it creates the successor nodes only for a representative subset of the set of all the enabled interactions $\mathit{EnabledSet}(\eta)$. To compute the independence relation between interactions, we define the following *valid dependence relation* [16] for BIP models: two interactions are *dependent* if they share a common component. This valid dependent relation can be computed statically from the BIP model. We have implemented both persistent set and sleep set POR approaches. The use of POR in ESST_{BIP} is correct since the application of POR to the general ESST framework is sound, provided a valid dependence relation [16].

Simultaneous Execution of the Edges Participating in an Interaction. In the basic ESST_{BIP}, we serialize the edges participating in the same interaction since we use a scheduling function that allows only one thread to run at a time. Consider an ARF node $\eta = \langle\langle l_1, \varphi_1\rangle, \ldots, \langle l_n, \varphi_n\rangle, \varphi, \mathbb{S}\rangle$, and an interaction $\gamma = \langle\{\alpha_1, \ldots, \alpha_k\}, g, op\rangle$ enabled in η. Let $\{t_{\alpha_1}, \ldots, t_{\alpha_k}\}$ be the set of participating edges in γ and $op_{\alpha_1}, \ldots, op_{\alpha_k}$ be their respective effects. When we expand η, we will create the following sequence of successor nodes:

$$\eta \xrightarrow{E2} \eta_1 \xrightarrow{E2} \eta_2 \xrightarrow{\alpha_1} \eta_3 \xrightarrow{op_{\alpha_1}} \eta_4 \xrightarrow{wait()} \eta_5 \xrightarrow{E2} \eta_6 \xrightarrow{\alpha_2} \ldots \xrightarrow{wait()} \eta_{2+4k}$$

where the label E2 denotes the execution of the ESST_{BIP} scheduler function. The intermediate nodes $\eta_1, \ldots, \eta_{2+4k-1}$ are due to the sequentialization of the execution of the edges participating in the interaction. These intermediate nodes increase the complexity of the reachability analysis. They do not correspond to any state reachable in the BIP model, where all the edges involved in an interaction are executed simultaneously, and are an artefact of the encoding of the BIP model into the ESST framework. We can modify the search discussed in Sect. 3 in order to avoid the generation of these intermediate states by (i) extending the primitive execution function Sexec to simultaneously evaluate a sequence of primitive functions, (ii) changing the node expansion rule E1 of ESST as follows:

E1'. If $\mathbb{S}(st_{T_i}) = Running$, let $T_R = \{T_i \in T | \mathbb{S}(st_{T_i}) \neq Waiting\}$ be the set of threads not in the $Waiting$ state. Let $op = op_1; \ldots; op_k$ be a sequential composition (in arbitrary order) of the operations labeling the outgoing edges $(l_i, op_i, l'_i) \in E_i$, for $T_i \in T_R$[7]. The successor node is$(\langle l'_1, \varphi'_1\rangle, \ldots, \langle l'_n, \varphi'_n\rangle, \varphi', \mathbb{S}')$, where:

- $\langle \mathbb{S}', \hat{op}\rangle = \begin{cases} \langle \mathbb{S}, op\rangle & \text{if none of the } op_i \text{ in } op \text{ is a call to } wait() \\ \langle \mathbb{S}'', skip\rangle & \text{if all } op_i \text{ in } op \text{ is a } wait() \text{ and} (*, \mathbb{S}'') = \text{Sexec}(\mathbb{S}, op) \end{cases}$
- $\varphi'_i = SP^{\delta_{l'_i}}_{\hat{op}_i}(\varphi_i \wedge \varphi)$ for each thread $T_i \in T_R$, $\varphi'_j = \varphi_j$ and $l'_j = l_j$ for each thread $T_j \notin T_R$, and $\varphi' = SP^{\delta}_{\hat{op}}(\varphi)$, where $\hat{op}_i$ is the projection of $\hat{op}$ on the instructions of thread T_i.

We remark that, in BIP, we do not have shared variables. Thus, all the op_i are local to the corresponding components, and executing a sequence of op_i altogether will not create any conflict. The correctness of this optimization can be easily justified since it respects BIP operational semantics.

Implicit Primitive Functions. The previous optimization does not remove all the intermediate ARF nodes $\eta_1, \ldots, \eta_{2+4k-1}$ visited by the ESST_{BIP} that do not have a corresponding configurations in the BIP operational semantics. In particular, we can avoid the creation of the intermediate ARF nodes created

[7] Note that there is no non-determinism on the outgoing edge to be executed by each thread T_i after the scheduling of the interaction γ.

by calls to *wait()* noting that: (i) *wait()* is always executed immediately after the execution of some edge t_{α_i} labeled by α_i, i.e. $\eta \xrightarrow{\alpha_i} \eta' \xrightarrow{wait()} \eta''$ (see the description of the sequence of the ARF nodes visited after the scheduling of an interaction in the previous optimization); (ii) *wait()* only modifies the scheduler states of an ARF node. Thus, we can combine the execution of *wait()* with the execution of its preceding edge t_{α_i}. This optimization can be integrated in the ESST_{BIP} framework by modifying rule E1 as follows.

E1". If $\mathbb{S}(st_{T_i}) = Running$, and $\{(l_i, op, l_i^1), (l_i^1, wait(), l_i')\} \subseteq E_i$, then the successor node is $(\langle l_1, \varphi_1' \rangle, \ldots, \langle l_n, \varphi_n' \rangle, \varphi', \mathbb{S}')$, where:
- $\langle \mathbb{S}', \hat{op} \rangle = \langle \mathbb{S}'', op; skip \rangle$ if op is not $wait()$ *and* $(*, \mathbb{S}'') = \textsc{Sexec}(\mathbb{S}, wait())$
- $\varphi' = SP^{\delta}_{\hat{op}}(\varphi)$, $\varphi_i' = SP^{\delta_{l_i'}}_{\hat{op}}(\varphi_i \wedge \varphi)$, and $\varphi_j' = \varphi_j$, for $i \neq j$,

This optimization is correct with respect to BIP semantics since ESST_{BIP} will still visit all the reachable states of the original BIP model $\mathcal{P}_{\text{BIP}}$. To see this, notice that there are no interactions to be scheduled in the intermediate sequence of ARF nodes created while executing an interaction, and after the execution of the edge t_{α_i} the thread T_i will always stop its execution.

We remark that the optimization for the implicit execution of primitive functions and the optimization for the simultaneous execution of the edges of an interaction can be combined together, to further reduce the search space of the basic ESST_{BIP}.

4 Encoding BIP into Transition System

In this section, we show how to encode a BIP model into a Symbolic Transition System, thus enabling a direct application of state-of-the-art model checkers for infinite state systems, such as the nuXmv [12] symbolic model checker.

A *Symbolic Transition System* (STS) is a tuple $S = \langle V, I, Tr \rangle$, where: (i) V is a finite set of variables, (ii) I is a first-order formula over V (called *initial condition*), and (iii) Tr is a first-order formula over $V \cup V'$ (called *transition condition*[8]). The semantic of an STS can be given in terms of an explicit transition systems (see for example [26]).

The encoding of a BIP model $\mathcal{P}_{\text{BIP}} = \langle \mathcal{B}, \Gamma, \pi \rangle$ as an STS $S_{P_{\text{BIP}}} = \langle V, I, Tr \rangle$ is the following. The set of variables is defined as:

$$V = \bigcup_{i=1}^{n} \{loc_i\} \cup \bigcup_{i=1}^{n} x | x \in Var_i \} \cup \bigcup_{i=1}^{n} \{v_\alpha | \alpha \in P_i\} \cup \{v_\Gamma\}$$

where for all $i \in [1, n]$, we preserve the domain of each var $x \in Var_i$, $Dom(loc_i) = Q_i$; for all $\alpha \in P_i$, $Dom(v_\alpha) = \{true, false\}$; and $Dom(v_\Gamma) = \Gamma$.

The initial condition is $I = \bigwedge_i^n (loc_i = l_{0_i})$, since we do not have initial predicates in $\mathcal{P}_{\text{BIP}}$. The transition condition is $Tr = (\bigwedge_{i=1}^{n} (Tr_{e_i} \wedge Tr_{p_i}) \wedge Tr_\Gamma \wedge Tr_\pi$, where Tr_{e_i} encodes the edges of the component B_i, Tr_{p_i} determines when

[8] Hereby and below, we denote with $V' = \{x' | x \in V\}$ the set of primed variables of V.

the variable v_α for port α is true, Tr_Γ encodes when an interaction is enabled, and Tr_π encodes the priorities.

In the following, let $\Gamma_{B_i} = \{\langle Act, g, op\rangle | Act \cap P_i \neq \emptyset\rangle$ be the set of all the interactions on which B_i participates and $\Gamma_e = \{\langle Act, g, op\rangle | e = \langle l_i, \alpha, g_e, op_e, l_i'\rangle, \alpha \in Act\}$, with $e \in E_i$, be the set of interactions that contain the port that labels e.

The encoding of an edge e of a component B_i is defined as:

$$Tr_{e_i} = \bigvee_{e=\langle l_i, Act\cap P_i, g_e, op_e, l_i'\rangle \in E_i} loc_i = l_i \wedge loc_i' = l_i' \wedge g_e \wedge \bigvee_{\gamma \in \Gamma_{B_i}} v_\Gamma = \gamma \wedge$$
$$\bigwedge_{\gamma \in \Gamma_{B_i}} \left(v_\Gamma = \gamma \rightarrow \bigwedge_{x \in Var_i} x' = update(x, e, \gamma)\right) \wedge \bigwedge_{\gamma \notin \Gamma_{B_i}} \left(v_\Gamma = \gamma \rightarrow \bigwedge_{x \in Var_i} x' = x\right)$$

$$update(x, e, \gamma) = \begin{cases} replace(e, \gamma) & \text{if } op_e = x := e \\ replace(x, \gamma) & \text{otherwise} \end{cases}$$

and $replace(e, \gamma)$ is a function that replaces all the occurrences of a variables y in e with e_γ, if $op_\gamma = y := e_\gamma$ and $\gamma = \langle Act, g_\gamma, op_\gamma\rangle$[9]. Tr_{p_i} is defined as $\bigwedge_{\alpha \in \Gamma} \left(v_\alpha \leftrightarrow \bigvee_{\langle l_i, \alpha, g_e, op_e, l_i'\rangle \in E_i} \left(loc_i = l_i \wedge g_e\right)\right)$. Finally, the conditions that constraint the interactions to their ports and the priorities among the interactions are defined as:

$$Tr_\Gamma = \bigwedge_{\gamma = \langle Act_\gamma, g_\gamma, op_\gamma\rangle \in \Gamma} \bigwedge_{\alpha \in Act_\gamma} v_\Gamma = \gamma \rightarrow (v_\alpha \wedge g_\gamma)$$

$$Tr_\pi = \bigwedge_{(\gamma_1, \gamma_2) \in \Gamma, \gamma_1 = \langle Act_{\gamma_1}, g_{\gamma_1}, op_{\gamma_1}\rangle} (g_{\gamma_1} \wedge \bigwedge_{\alpha \in Act_{\gamma_1}} v_\alpha) \rightarrow v_\Gamma \neq \gamma_2$$

Theorem 2. *The transition system $S_{P_{BIP}}$ for a BIP model $\mathcal{P}_{\text{BIP}}$ preserves reachability of any configuration of the BIP model.*

The proof relies on the fact that the state space of the BIP model is preserved. The initial configuration is preserved by formula I, where loc_i is constrained to the initial locations of the corresponding component. The transition relation is also preserved, since the variable v_Γ can be assigned to the value representing an interaction γ, enabling the corresponding edges, if and only if γ is enabled in the corresponding state of the BIP model. The valuations of the additional variables v_α and v_Γ do not alter the state space: their valuations are constrained by formula the Tr to reflect the BIP semantics.

5 Related Work

Several approaches to the verification of BIP models have been explored in the literature. DFinder [7] is a verification tool for BIP models that relies on compositional reasoning for identifying deadlocks and verifying safety properties.

[9] Note that, while in our definition op_γ is a single assignment, the approach can be easily generalized to sequential programs applying a *single-static assignment (SSA)* transformation [18].

The tool has several limitations: it is unsound in the presence of data transfers among components (it assumes that the involved variables do not exchange values); its refinement procedure is not effective for infinite-state systems, since it consists only in removing the found unreachable deadlock states from the next round of the algorithm; finally, it can only handle BIP models with finite domain variables or integers. Our approaches instead are sound in the presence of data transfer, they exploit standard refinement mechanisms (e.g. refinement based on interpolation) and can handle BIP models with real variables.

The VCS [20] tool supports the verification of BIP models with data transfer among components, using specialized BDD- and SAT-based model checking algorithms for BIP. Differently from our approach, VCS is only able to deal with finite domain variables, and priority is ignored.

Our encoding in transition system is related to works in [25,29]. In [29], a timed BIP model is translated into Timed Automata and then verified with Uppaal [6]. The translation handles data transfers, but it is limited to BIP models with finite domain data variables and without priorities. In [25], the authors show an encoding of a BIP models into Horn Clauses. They do not handle data transfers on interactions and do not describe how to handle priorities. We remark that, any transition system can be encoded into Horn Clauses and then verified with tools such as Z3 [23] or Eldarica [24].

With respect to the verification of multi-threaded programs, the works most related to ours are [16,17,30]. In [16,17], the authors present the ESST framework, instantiating it for SystemC [27] and FairThreads [11]. They neither consider instantaneous synchronizations nor priorities among interactions. Instead, in this work we instantiate the ESST framework for the analysis of BIP models, which encompasses instantaneous synchronizations and priorities. The semi-symbolic scheduler in [17] is also different from ours: while they use the semi-symbolic scheduler to handle parameters of the primitive functions, we use it to change the status of the local threads. We also apply and adapt several optimizations sound w.r.t. the BIP operational semantics. The work in [30] combines lazy abstraction and POR for the verification of generic multi-threaded programs with pointers. They do not leverage on the separation between coordination and computation which is the core of our ESST_{BIP} approach. Moreover, because of the pointers, they rely on a dynamic dependence relation for applying POR.

6 Experimental Evaluation

We implemented ESST_{BIP} extending the Kratos [13] software model checker. We implemented the encoding from BIP to transition system in a tool based on the BIP framework [2]. Our tool generates models in the input language of nuXmv, allowing us to reuse its model checking algorithms.

In the experimental evaluation, we used several benchmarks taken and adapted from the literature, including the temperature control system model and ATM transaction model used in [7], the train gate control system model used in [25], and several other consensus and voting algorithm models. Every

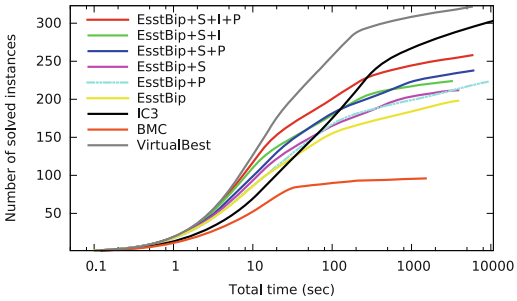

Fig. 1. Cumulative plot for all the benchmarks

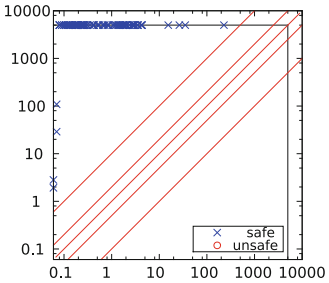

Fig. 2. Run time (sec.) DFinder (y axes) Ic3 (x axes)

benchmark is scalable with respect to the number of components. In total, we created 379 instances of both safe and unsafe models, and verified different invariant properties. All the benchmarks are infinite-state, due to integer variables, and some of them feature data transfer on interaction. Due to lack of space, we do not provide the details of each benchmark, but refer to our webpage[10] for more information.

We run several configurations of ESST_{BIP}: EsstBip, EsstBiP+P, EsstBip+S, EsstBip+S+P, EsstBip+S+I and EsstBip+S+I+P, where EsstBip is the base version without any optimization, P denotes the use of partial order reduction, S denotes the use of the simultaneous execution of the interaction edges and I denotes the implicit execution of the primitives functions. After the encoding into transition systems, we run two algorithms implemented in nuXmv: (Ic3) an implementation of the *IC3* algorithm integrated with predicate abstraction [14]; (Bmc) an implementation of *Bounded Model Checking* [9] via SMT [1] solving. For the benchmarks that do not exhibit data transfer, we also compared our approaches against DFinder (version 2) [7].

All the experiments have been performed on a cluster of 64-bit Linux machines with a 2.7 Ghz Intel Xeon X5650 CPU, with a memory limit set to 8 Gb and a time limit of 900 s. The tools and benchmarks used in the experiments are available in our webpage.

Comparison with DFINDER. We first compare on the subset of the benchmarks (100 instances) that DFinder can handle (these benchmarks do not have data transfer and are safe). We compare DFinder and Ic3 in the scatter plot of Fig. 2: DFinder is able to solve only 4 of our instances, while Ic3 solves all the 100 instances. The best configuration of ESST_{BIP} (EsstBip+S+I+P) shows a similar trend (solving 75 instances). For lack of space we do not show the respective plot. DFinder requires about 142 s to solve the four benchmarks, while both Ic3 and EsstBip+S+I+P solve all of them in a fraction of a second. The main explanations for these results are: (i) DFinder cannot prove 60

[10] https://es.fbk.eu/people/mover/atva15-kratos.tar.bz2.

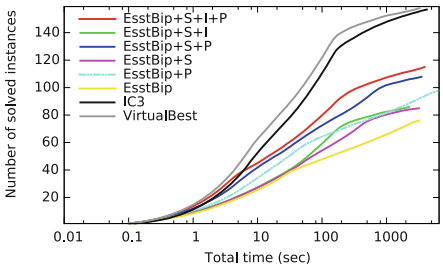

Fig. 3. Safe benchmarks

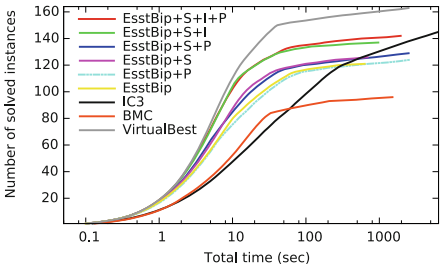

Fig. 4. Unsafe benchmarks

instances since it cannot find strong enough invariants to prove the property; (ii) it exceeds the memory limits for the remaining 36 instances.

Comparison of NUXMV and ESST$_{\text{BIP}}$. We show the results of the comparison among our approaches on the full set of instances in Fig. 1, where we plot the cumulative time to solve an increasing number of instances. IC3 clearly outperforms all the other approaches, while the version of ESST$_{\text{BIP}}$ with all the optimization outperforms all the other ESST$_{\text{BIP}}$ configurations. In Fig. 3 we focus only on the safe instances: the plot shows that IC3 is more efficient than ESST$_{\text{BIP}}$. IC3 is much more effective than ESST$_{\text{BIP}}$ on a subset of the instances, where IC3 can easily find an inductive invariant (for this subset, the number of frames needed by IC3 to prove the property does not increase when increasing the number of components in each benchmark). In these cases instead, ESST$_{\text{BIP}}$ still has to visit several nodes before succeeding in the coverage check. In Fig. 4, we focus on the unsafe properties. Both all the ESST$_{\text{BIP}}$ approaches that enable the implicit primitive function execution and IC3 outperform BMC. The main reason is that BMC is not effective on long counterexamples, while in our benchmarks the length of the counterexamples grows with the number of components. We also observe that, for the unsafe cases, the approach EsstBip+S+I+P is faster than IC3. Thus, the experiments show that IC3 and ESST$_{\text{BIP}}$ are complementary, with IC3 being more efficient in the safe case, and ESST$_{\text{BIP}}$ being more efficient for the unsafe ones. This can be also seen in Fig. 1, where we plot the virtual best configuration (VirtualBest) (i.e. the configuration obtained taking the lower run time for each benchmark), which shows the results that we would obtain running all our approaches in parallel (in a portfolio approach).

Evaluation of the ESST$_{\text{BIP}}$ Optimization. In Fig. 5a and b we show two scatter plots to compare the results obtained with and without partial order reduction. The plot 5a shows how POR improves the performance when applied to EsstBip (for EsstBip+S we get similar results), while the plot 5b shows the same for EsstBip+S+I. The plots show that POR is effective on almost all benchmarks, even if in some cases the POR bookkeeping introduces some overhead. In Fig. 5c and d we show the results of applying the simultaneous execution

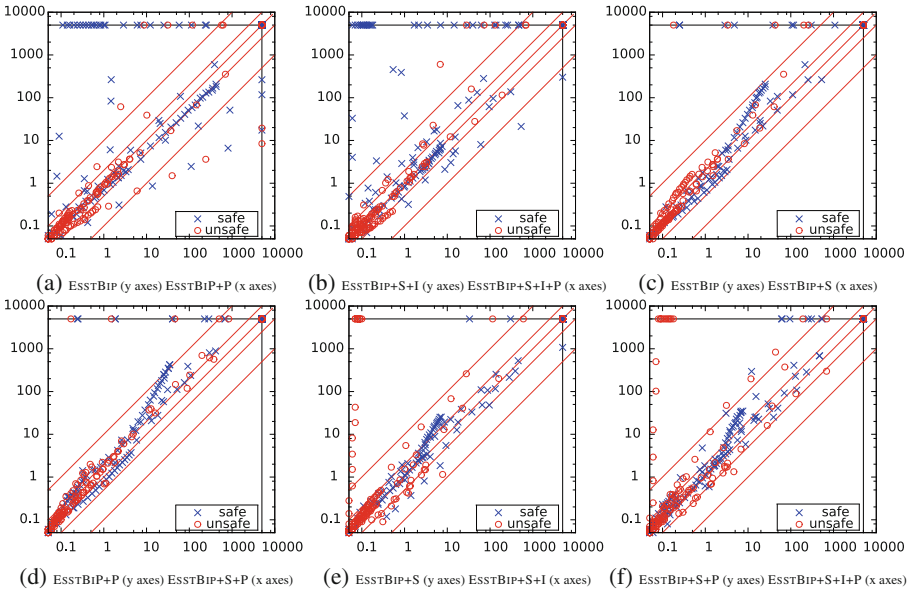

Fig. 5. Scatter plots of run times (sec.) for the ESST_{BIP} optimizations

of the edges participating in an interaction to the basic configuration with and without partial order reduction enabled (EsstBip and EsstBiP+P). In both cases, the improvements to the run times brought by the concurrent execution of edges is consistent, since the run times are always lower and the number of solved instances higher. Finally, in Fig. 5e and f we show the plots that compares EsstBip+S with EsstBip+S+I and EsstBip+S+P with EsstBip+S+I+P. In both cases the implicit execution of the primitives functions always brings a performance improvement.

7 Conclusions and Future Work

In this paper, we described two complementary approaches for the verification of infinite-state BIP models that, contrary to the existing techniques, consider all the features of BIP such as the global effects on the interactions and priorities. First, we instantiated for BIP the ESST framework and we integrated several optimization sound w.r.t. the BIP semantics. Second, we provided an encoding of BIP models into symbolic transition systems, enabling us to exploit the existing state of the art verification algorithms. Finally, we implemented the proposed techniques and performed an experimental evaluation on several benchmarks. The results show that our approaches are complementary, and that they outperform DFinder w.r.t performance and also w.r.t. the coverage of the BIP features. As future work we would like extend the proposed techniques to support timed BIP [4] (e.g. the symbolic encoding could be extended to HyDI [15])

and, in the case of ESST we would improve its performance in finding bugs using direct model checking [19]. Finally, we will investigate the possibility to exploit the invariants computed by DFinder in all our approaches.

References

1. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. Handb. Satisfiability **185**, 825–885 (2009)
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. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010, Part II. LNCS, vol. 6117, pp. 32–46. Springer, Heidelberg (2010)
4. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM (2006)
5. Basu, A., Gallien, M., Lesire, C., Nguyen, T.H., Bensalem, S., Ingrand, F., Sifakis, J.: Incremental component-based construction and verification of a robotic system. ECAI **178**, 631–635 (2008)
6. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: QEST (2006)
7. 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)
8. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD (2009)
9. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, p. 193. Springer, Heidelberg (1999)
10. Bliudze, S., Cimatti, A., Jaber, M., Mover, S., Roveri, M., Saab, W., Wang, Q.: Formal verification of infinite-state bip models. Technical report. https://es-static.fbk.eu/people/mover/paper/fvbip.pdf
11. Boussinot, F.: FairThreads: mixing cooperative and preemptive threads in C. Concurrency Comput. Pract. Experience **18**(5), 445–469 (2006)
12. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Heidelberg (2014)
13. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos – a software model checker for systemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 310–316. Springer, Heidelberg (2011)
14. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 46–61. Springer, Heidelberg (2014)
15. Cimatti, A., Mover, S., Tonetta, S.: HyDI: a language for symbolic hybrid systems with discrete interaction. In: SEAA (2011)
16. Cimatti, A., Narasamdya, I., Roveri, M.: Software model checking with explicit scheduler and symbolic threads. Log. Methods Comput. Sci. **8**(2), 1–42 (2012)

17. Cimatti, A., Narasamdya, I., Roveri, M.: Verification of parametric system designs. In: FMCAD (2012)
18. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
19. Edelkamp, S., Schuppan, V., Bošnački, D., Wijs, A., Fehnker, A., Aljazzar, H.: Survey on directed model checking. In: Peled, D.A., Wooldridge, M.J. (eds.) MoChArt 2008. LNCS, vol. 5348, pp. 65–89. Springer, Heidelberg (2009)
20. He, F., Yin, L., Wang, B.-Y., Zhang, L., Mu, G., Meng, W.: VCS: a verifier for component-based systems. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 478–481. Springer, Heidelberg (2013)
21. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. ACM SIGPLAN Not. **39**, 232–244 (2004). ACM
22. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL (2002)
23. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
24. Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 247–251. Springer, Heidelberg (2012)
25. Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn clauses for communicating timed systems. In: HCVS (2014)
26. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York (1992)
27. IEEE 1666: SystemC language Reference Manual (2005)
28. Sifakis, J.: Rigorous system design. Found. Trends Electron. Des. Autom. **6**(4), 293–362 (2013)
29. Su, C., Zhou, M., Yin, L., Wan, H., Gu, M.: Modeling and verification of component-based systems with data passing using BIP. In: ICECCS (2013)
30. Wachter, B., Kroening, D., Ouaknine, J.: Verifying multi-threaded software with Impact. In: FMCAD (2013)