Efficient Deadlock Detection for Concurrent Systems

Saddek Bensalem*, Andreas Griesmayer*†, Axel Legay‡, Thanh-Hung Nguyen* and Doron Peled§
*Verimag Laboratory, Université Joseph Fourier Grenoble, CNRS, France
†Imperial College London, UK
‡INRIA/IRISA, Rennes, France
§Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

***Abstract*—Concurrent systems are prone to deadlocks that arise from competing access to shared resources and synchronization between the components. At the same time, concurrency leads to a dramatic increase of the possible state space due to interleavings of computations, which makes standard verification techniques often infeasible. Previous work has shown that approximating the state space of component based systems by computing invariants allows to verify much larger systems then standard methods that compute the exact state space. The approach comes with the drawback, though, that not all of the reported specification violations may be reachable in the system. This paper deals with that problem by combining the information from the invariant with model checking techniques and strategies for reducing the memory footprint. The approach is implemented as post processing step for generating the exact set of reachable specification violations along with traces to demonstrate the error.**

I. Introduction

Deadlock detection is one of the most common tasks in model checking and can be used (with some overhead) to check any safety property [1]. The complexity of deadlock detection of concurrent systems is PSPACE-complete in the size of the system, same as model checking. There have been many approaches for deadlock detection, e.g., using explicit state space [2], [3], [4], symbolic model checking [5] and SAT solving [6]. Due to the inherent complexity of the problem, it is not surprising that each approach finds its own difficult instances.

We propose here a new algorithm for deadlock detection for concurrent systems. Our method works in two stages; first we use an over-approximation to limit the potential deadlocks using a recent technique [7], [8]; then we employ a backwards-forwards search to eliminate false positives and to provide the full counterexamples from initial states to deadlocks. The main principle of our approach is to provide a tight invariant to calculate an efficient over-approximation for the reachable deadlocks of the system. There is an obvious tradeoff between the tightness of the initial approximation (which is a key factor in the success of our algorithm) and the efficiency of its calculation; a forward search from the initial states, for example, would provide an exact set of reachable states and deadlock states, however, it may fail to do so due to time and space limitations.

The research leading to these results has received funding from the European Communitys Seventh Framework Programme [FP7] under grant agreements no 257414 (ASCENS).

Our approach starts by using structural properties of the system to provide an over-approximation of the set of reachable states. It combines local invariants of the processes with an interaction invariant, calculated based on *Boolean behavioral constraints* that capture the effects of synchronization of parallel components [7], [8]. The computation of the interaction invariants is efficient, as it does not involve a state space search. We find this combination of local invariants and interaction invariant quite satisfying in terms of the tightness of the approximation and the amount of work required, but it comes with the drawbacks that 1) not all of the reported deadlocks are really reachable and 2) it does not generate counterexamples for the deadlocks, which makes locating the actual error much harder.

To remedy these problems, we add a second phase that is the main contribution of this paper. In this phase, we use the invariant to compute a set of deadlock suspects and perform a backwards/forwards search to check their reachability: from the deadlock states towards the initial states, and reversely, from the initial states towards the deadlock states. In this way we find concrete traces for each reachable deadlock state (starting from initial states and ending at deadlock states). The approach exploits *binary decision diagrams (BDDs)* [9] to use the invariant for a reduction of the memory footprint of the procedure, and is therefore dependent on a Boolean representation of the model. For problems that can be directly represented in such a way, the presented approach returns the exact set of reachable deadlock states along with an error trace that demonstrates its reachability. General BIP models require abstraction, e.g., by reduction to the Boolean variables and the control structure of BIP (control states of the components and their interactions). Such an abstraction may lead to infeasible traces, which do, however, show the interactions among components and assists a manual feasibility check to find possible remaining false positives. Traces to the deadlocks are also a precondition for future extensions to compute better abstractions by performing automatic feasibility checks and predicate abstraction in form of an abstraction refinement loop (CEGAR). As usual for infinite systems, a CEGAR loop may not terminate, but nevertheless produce intermediate models that are valid for refuting false positives [10], [11]. For this paper and the implementation we consider Boolean systems and do not go into details on their construction.

Our approach has been implemented in the BIP tool-set [12]. In the BIP language, atomic components are represented by transition systems labeled with C/C++ functions – which allows the encoding of a wide range of applications including module coordination in robotic software [13]. Bigger components are obtained by combining smaller units through a series of interactions and communication primitives. The tool can use those communication mechanisms to automatically synthesize C code that coordinates the components. BIP uses DFinder [7] to verify programs written in the BIP language. In order to achieve this goal, DFinder computes an invariant of the system that is an (hopefully compact) over-approximation of the set of reachable states on which the tool can reason in an efficient manner. In addition, DFinder uses an incremental design process which exploits the hierarchy between components that is induced by the BIP language. More precisely, DFinder allows reuse of the results obtained for sub-components in order to ease the verification of the global system. We have implemented the counter example refinement algorithm on top of DFinder, hence providing a finer analysis of BIP models by removing false-positive introduced by invariant over-approximation.

A. Related Work.

There are numerous approaches and tools that propose techniques for checking deadlocks of concurrent systems (see for e.g., [3], [4], [14], [15], [16]). These approaches comprise techniques like abstraction and symmetry or partial order reduction and are highly effective in reducing the state space to be searched, and in principle can be combined with the approach shown in this paper. Other approaches do not reduce number of states to be searched, but impose additional conditions and search strategies to keep the representation small. [17], [18] systematically perform under-approximations to incrementally compute the full set or reachable states. In contrast, we use information that can be computed statically in form of invariants to give an over- approximation of the reachable state space and use it to reduce the size of the BDDs.

The work presented here is in the context of the *BIP* framework, which promotes a component based design and allows synthesizing the C code for coordination of the participating components. This structure helps in generating the invariants required for the presented approach and furthermore allows its application already in early design stages. In contrast to this, most existing approaches for general programs in languages such as C or Java [19], [20], [10] start from the code and need to extract the control structure by, e.g., abstraction to decide whether the underlying system behaves properly. While these approaches often work very well for single programs, it is hard to extract dependencies among distributed programs – information that is given explicitly in the work flow of *BIP*.

B. Structure of the paper.

Section II gives a brief introduction to the language BIP (used to describe our systems) and to the methods used to compute the deadlock suspects. Section III presents the general methodology on how to compute the set of all shortest counterexamples while exploiting the advantages given by the use of the invariant. In this section, we also propose a series of heuristics to improve efficiency. Section IV discusses experimental results and compares the approach to previous results without counter example generation. Finally, Section V concludes the paper and provides an overview over some related work.

II. Preliminaries

In this section, we present concepts and definitions that will be used throughout the rest of the paper. We start with the component-based design framework that is used to describe our systems. Then we give a brief description on the computation of component and interaction invariants, which are used to reduce the state space in further computation. The framework that we will present resembles a subset of the *BIP* modeling language [21]. The language of the tool allows for component-based design of complex systems in a hierarchical manner. Although not all *BIP* models are directly covered, a large subset can be transformed to this framework [22], [13], [23]. For more technical details regarding this section, please refer to current work on *BIP* and invariant generation presented in [24], [8].

A. A framework for Concurrent Systems

In our framework, a concurrent system shall be viewed as a set of atomic components that synchronize on interactions. To define the semantics of an atomic component, we start with a transition system with a set of locations $L = \{l_1, l_2, \ldots, l_k\}$, and transitions $\mathcal{T} \subseteq L \times P \times L$, where P denotes the set of *ports* that are used for synchronization between components. This basic transition system is extended by adding variables X, guards $\{g_\tau\}_{\tau\in\mathcal{T}}$ and actions $\{f_\tau\}_{\tau\in\mathcal{T}}$ on transitions as follows:

Definition 1 (Semantics of Atomic Components): The semantics of an Atomic Component B=(L,P, $\mathcal{T}$, X, $\{g_\tau\}_{\tau\in\mathcal{T}}$, $\{f_\tau\}_{\tau\in\mathcal{T}}$), is a transition system $(Q, P, \mathcal{T}_0)$ such that

- $Q = L \times \mathbf{X}$ where $\mathbf{X}$ denotes the set of valuations of variables X.
- $\mathcal{T}_0$ is the set of transitions $((l, \mathbf{x}), p, (l', \mathbf{x}'))$ such that $g_\tau(\mathbf{x}) \wedge f_\tau(\mathbf{x}, \mathbf{x}')$ for some $\tau = (l, p, l') \in \mathcal{T}$. As usual, if $((l, \mathbf{x}), p, (l', \mathbf{x}')) \in \mathcal{T}_0$ we write $(l, \mathbf{x}) \xrightarrow{p} (l', \mathbf{x}')$.

Intuitively, L defines the set of control locations. We call a valuation of the variables at a control location ($L \times \mathbf{X}$) a *configuration* c and write $l(c)$ for its location and $\mathbf{x}(c)$ for the variable valuation respectively. A transition τ is *enabled* for a valuation $\mathbf{x}$ if $g(\mathbf{x})_\tau$ gives *true*, and $f_\tau(\mathbf{x}, \mathbf{x}')$ is a variable transformation with $\mathbf{x}'$ being the new values. We use location l also as predicate which is $true$ iff the system is at location l. A state predicate Φ is a Boolean expression involving location predicates and predicates on X. Any state predicate can be put in the form $\bigvee_{l\in L} l \wedge \varphi_l$. Notice that the transition system is in exactly one control location, which means that only one location predicate is true at a time and their disjunction is always true.

The system is composed by synchronizing the atomic components by interactions, which describe the simultaneous execution of a transition at different components as follows:

Definition 2 (Interactions): Given a set of components $B_1, B_2, \ldots, B_n$, where $B_i = (L_i, P_i, \mathcal{T}_i, X_i, \{g_\tau\}_{\tau\in\mathcal{T}_i}, \{f_\tau\}_{\tau\in\mathcal{T}_i})$, an interaction a is a nonempty set of ports, subset of $\bigcup_{i=1}^n P_i$, such that $\forall i = 1, \ldots, n\ |a \cap P_i| \leq 1$.

The BIP framework allows defining rich interaction models by using hierarchical interactions extended with data transfer as presented in [22], [13]. In this work, we restrict to pure synchronizations. The absence of hierarchy is not a real limitation, as long as hierarchical interaction models can be statically transformed into equivalent flat interaction models with a potentially increased number of interactions [22], [13], [23].

Definition 3 (Parallel Composition): Given n components B_i=$(L_i, P_i, \mathcal{T}_i, X_i, \{g_\tau\}_{\tau\in\mathcal{T}_i}, \{f_\tau\}_{\tau\in\mathcal{T}_i})$ that do not share common variables and a set of interactions γ, we define $B = \gamma(B_1, \ldots, B_n)$ as the component $(L, \gamma, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$, where:

- $(L, \gamma, \mathcal{T})$ is the transition system such that
  - $L = L_1 \times L_2 \times \ldots \times L_n$ is the set of control locations,
  - $\mathcal{T} \subseteq (L, X) \times \gamma \times (L, X)$ consists of transitions $\tau = ((l_1, \mathbf{x}_1), \ldots, (l_n, \mathbf{x}_n), a, ((l'_1, \mathbf{x}'_1), \ldots, (l'_n, \mathbf{x}'_n)))$ obtained by synchronization of sets of transitions $\{\tau_i = ((l_i, \mathbf{x}_i), p_i, (l'_i, \mathbf{x}'_i) \in \mathcal{T}_i\}_{i\in I}$ such that $\{p_i\}_{i\in I} = a \in \gamma$ and $l'_j = l_j$ if $j \notin I$, for arbitrary $I \subseteq \{1, ..., n\}$
- $X = \bigcup_{i=1}^n X_i$ and for a transition τ resulting from the synchronization of a set of transitions $\{\tau_i\}_{i\in I}$, the associated guard and function are respectively $g_\tau = \bigwedge_{i\in I} g_{\tau_i}$ and $f_\tau = \bigwedge_{i\in I} f_{\tau_i} \wedge \bigwedge_{i\notin I}(X'_i = X_i)$.

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

Definition 4 (System): A system $\mathcal{S}$ is a pair $\langle B, Init\rangle$ where B is a component and $Init$ is a state predicate characterizing the initial states of B.

Example 1: Figure 1 shows the model of the well known Dining Philosopher example with two philosophers *P1* and *P2* using two forks *F1* and *F2*. The diamond states are initial. Each philosopher picks first the left fork (*pick_l*), then the right fork (*pick_r*), then eats (*eat*) and eventually puts back both forks simultaneously. Picking up a fork is done by synchronization with the *pick* port of the respective fork component, while *eat* is an internal port that is not synchronized with any other component. Note that the *put* interaction comprises three component and is only enabled when all participants have the respective port enabled. It is easy to see that the given configuration leads to a deadlock if *P1*, *F1* perform the interaction $\{P1.pick\_l, F1.pick\}$, followed by the interaction $\{P2.pick\_l, F2.pick\}$ from the other two components. The reason is that this sequence of interactions leads to the global state $P1.HAS\_ONE\_FORK$, $P2.HAS\_ONE\_FORK$, $F1.USED$, $F2.USED$, in which none of the interactions is enabled. As usual, the deadlock can be removed by changing one of the philosophers to first take the right fork (or changing the connectors respectively). Note that in addition to the components shown in this example, a component also can contain local variables which can be changed when a transition is executed. Furthermore, the enabledness of a transition (and therefore a port) can be dependent on expressions over the local variables.

Definition 5 (Invariants): Given a component as $B = (L, P, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$, a predicate Inv on (L, X) is an invariant of B, denoted by $inv(B, Inv)$, if for any location $l \in L$, valuation $\mathbf{x}$, and port $p \in P$, $Inv(l, \mathbf{x})$, $\tau = (l, \mathbf{x}) \xrightarrow{p} (l', \mathbf{x}') \in \mathcal{T}$, and $f_\tau(\mathbf{x}, \mathbf{x}')$ imply $Inv(l', \mathbf{x}')$, where $Inv(l, \mathbf{x})$ means that l and $\mathbf{x}$ satisfy Inv. For a system $S = \langle B, Init\rangle$, Inv is an invariant of $\mathcal{S}$, denoted by $inv(\mathcal{S}, Inv)$, if it is an invariant of B and if $Init \Rightarrow Inv$.

Clearly, if Inv_1, Inv_2 are invariants of B (respectively $\mathcal{S}$) then $Inv_1 \wedge Inv_2$ and $Inv_1 \vee Inv_2$ are also invariants of B (respectively $\mathcal{S}$).

In the next section, we recap an efficient technique that can be used to compute invariants that was presented in [25], [26], [7], [24], [8]. We also show how to check for deadlocks using this methodology.

B. Invariant Computation

A tight approximation of the reachable state-space of the concurrent system is key to our approach. This approximation is computed by conjunction of two different invariants: The *component invariant* Φ, and *interaction invariant* Ψ. The component invariant is computed as conjunction of Φ_i, local invariants that over approximate the reachable states on the atomic components, while the interaction invariant Ψ captures restrictions on the reachable state space imposed by the communication between the components. In a series of work [26], [25], [7], we have presented new and efficient methodologies to compute these invariants and verify properties of concurrent systems. In the center of these methods is the approximation of the reachable states by compositional invariant computation based on the following rule:

$$\frac{\{B_i < \Phi_i >\}_i,\ \Psi \in II(\|_\gamma\{B_i\}_i, \{\Phi_i\}_i),\ (\bigwedge_i \Phi_i) \wedge \Psi \Rightarrow \Phi}{\|_\gamma\{B_i\}_i < \Phi >}$$

The rule states that if all components B_i fulfill their respective *component invariants* Φ_i, the composition of all components $II(\|_\gamma\{B_i\}_i, \{\Phi_i\}_i)$ with the interactions γ fulfills an *interaction invariant* (which expresses constraints on the global state space induced by interactions) Ψ, and if the conjunction of the invariants $(\bigwedge_i \Phi_i) \wedge \Psi$ implies a predicate on the global system Φ, then also the global system $\|_\gamma\{B_i\}_i$ itself fulfills Φ. Although the approach shown in this paper is applicable to general safety properties, we concentrate on

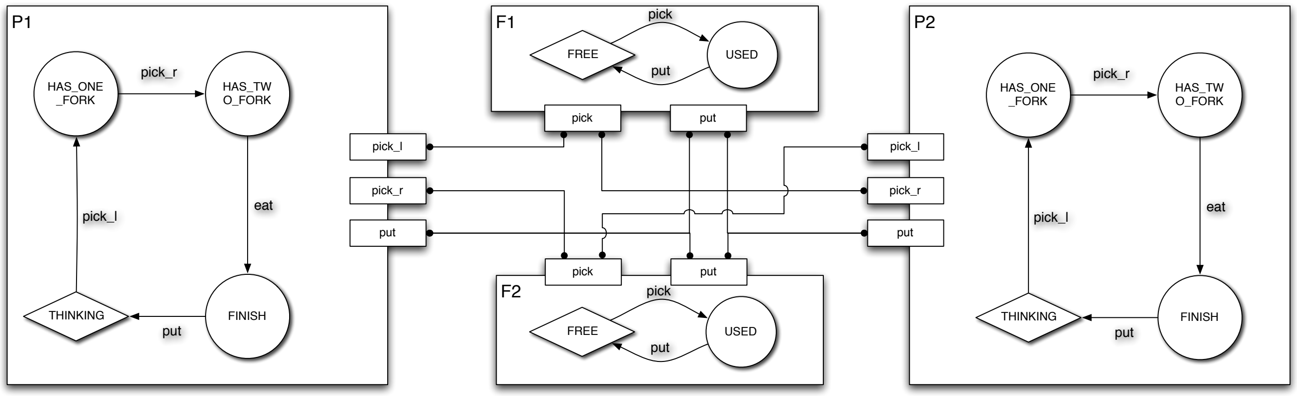

Fig. 1. Dining Philosophers

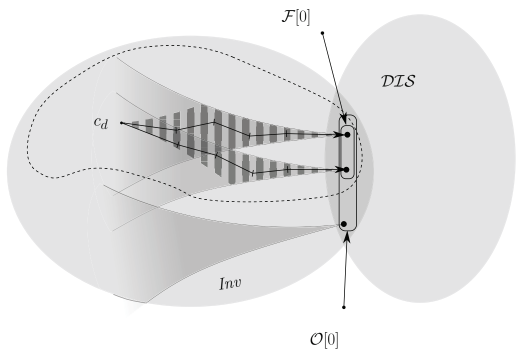

Fig. 2. Counter example generation using pre- and postimage computation

deadlock detection. To this end, we first compute the set of deadlocks as $\mathcal{DIS} = \bigwedge_{\tau \in \mathcal{T}} : \neg enabled(\tau)$. The above rule can now be used to check global deadlock-freedom by proving that predicate $\neg\mathcal{DIS}$ is implied by the invariants.

- *Computing Component Invariants:* The component invariants Φ_i are local invariants that over-approximate the reachable states on the atomic components. Each Φ_i is computed as fixed point from alternating computation of the possible valuations $\mathbf{x}$ on one of its locations l, and the post-condition computation of the transitions. Intuitively, the possible evaluations in a location l is the disjunction of the valuations that are computed by the functions f on incoming transitions, which in turn can be computed as the post-condition of the invariant at the start state, strengthened by the guard. This computation continuously leads to smaller sets of possible valuation of the variables until eventually a fixed-point (or timeout) is reached. For finite state systems, the local set of reachable states can also be computed exactly to improve the overall invariant. See [27] for more details.
- *Computing Interaction Invariant:* The *interaction invariant* Ψ captures constraints on the behavior of the system that are induced by the synchronization of the components. Static analysis of the atomic components and their interactions gives so called *Boolean behavioral constraints* [8], which are structural properties of concurrent systems that capture the communication between different components and hence allow to model a unified transition relation. Solutions of BBCs can be used to symbolically compute a strong interaction invariant. A series of previous work is concerned with various methodologies [28], [24], [8], [25] to effectively compute Ψ using the above principle. In our experiments, we use our most recent methodology to incrementally construct Ψ from smaller subsystems [8]. The resources spent to compute the invariants are negligible compared to the efforts of generating the error traces.

III. Methodology

Using the results from the previous section, we compute the set of *deadlock suspects* as $\mathcal{D} = (\bigwedge_i \Phi_i) \wedge \Psi \wedge \mathcal{DIS}$. In practice, this gives a good approximation of the deadlock state. In fact, this approach has shown to be very effective for verification of component based systems [13]. This section shows how to extend this approach of invariant computation by static analysis with state space search. This combined approach removes false positives from $\mathcal{D}$ and results in the exact set of deadlocks along with the corresponding error traces from the initial states. Intuitively, we use the global invariant of the system, $Inv = (\bigwedge_i \Phi_i) \wedge \Psi$, to guide the state space search from $\mathcal{D}$ towards the initial states.

Although the invariant computation and many of the following concepts are applicable to general systems, we restrict ourselves in the following to *Boolean systems* and discuss the implementation and optimizations based on *Boolean decision diagrams* (BDD) [9]. In practice, such a system can be obtained by removing all operations and conditions on data from the components, and only considering the explicit control and interaction structure of BIP models. In such a case, not all moved, but the users are assisted by the computed trace in identifying potentially remaining false positives. (More accuracy can be obtained by predicate abstraction.)

To describe the details of the algorithm, we extend the notion of configuration from components to systems with parallel composition in the natural way as $c \in ((L_1 \times L_2 \times \cdots \times L_n), \mathbf{X})$ and define a trace in $\mathcal{S}$ as follows:

Definition 6 (Trace): Given a System composed of parallel components $\mathcal{S} = (\mathcal{L}, \gamma, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$, we define a trace π of length m as $(c^1, a^1, c^2, \ldots, a^{m-1}, c^m)$ with $(c^j, a^j, c^{j+1}) = \tau_j \in \mathcal{T}$, $g_{\tau_j}(\mathbf{x}(c^j)) = true$ and $f_{\tau_j}(\mathbf{x}(c^j)), \mathbf{x}(c^{j+1}))$ for $0 < j < m$

A *trace* is *initial* if $c^1 \in init$, and is called an error trace (or interchangeably counterexample) if it is initial and has a final configuration $c^m \in \mathcal{D}$. To identify and remove false positives, we search for an error trace for each of the states in $\mathcal{D}$. The approach is visualized in Fig. 2 with $\mathcal{DIS}$ and *Inv* depicted as ellipses, and the (unknown) reachable states within a dotted line. Computation starts at $\mathcal{D}$ and performs pre-state computation to find initial configurations c_d that reach a deadlock. In the second step, forward computation is performed to compute the states that are on an error trace. If the approach cannot find such an error trace for a state d, then d is not *reachable* and therefore a false positive that is excluded from the result. The algorithm is given as pseudo code in Fig. 3.

We denote sets of configurations as capital letters Y, Z, and define pre- and postimage computation as follows:

Definition 7 (Preimage): Given a System of parallel components $\mathcal{S} = (\mathcal{L},\gamma, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$ and an invariant *Inv*, we define the preimage $pre()$ of a set of configurations Z as

$$\begin{aligned} pre(Z) = \{c \quad &| \quad \exists c' \in Z, a \in \gamma, \tau \in \mathcal{T} \\ &. \quad (c, a, c') = \tau \wedge c \in Inv \\ &\wedge \quad g_\tau(\mathbf{x}(c)) = true \wedge f_\tau(\mathbf{x}(c)), \mathbf{x}(c')\} \end{aligned}$$

and similarly

Definition 8 (Postimage): Given a System of parallel components $\mathcal{S} = (\mathcal{L},\gamma, \mathcal{T}, X, \{g_\tau\}_{\tau\in\mathcal{T}}, \{f_\tau\}_{\tau\in\mathcal{T}})$ and an invariant *Inv*, we define the postimage $post()$ of a set of configurations Z as

$$\begin{aligned} post(Z) = \{c' \quad &| \quad \exists c \in Z, a \in \gamma, \tau \in \mathcal{T} \\ &. \quad (c, a, c') = \tau \wedge c' \in Inv \\ &\wedge \quad g_\tau(\mathbf{x}(c)) = true \wedge f_\tau(\mathbf{x}(c)), \mathbf{x}(c')\} \end{aligned}$$

Note that the pre- and postimages return only states within the invariant. This guides the search in the direction of the reachable states as traces that include states outside the invariant are omitted. It is easy to see that this guidance does not influence the computation of *initial* traces, as by definition any state on an initial trace is reachable, and by construction of *Inv* also within the invariant.

In the first phase of the counter example computation, we incrementally construct the states that reach $\mathcal{D}$ (depicted by gray cones in Fig 2) by computing

```
Let $\mathcal{F}[0] := \mathcal{D}$
while $\mathcal{D} \neq false$ and $\mathcal{F}[0] \neq false$
begin
  Let $i := 0$
  Let $\mathcal{O}[0] := \mathcal{D}$;
  Let $S := false$;
  while $\mathcal{O}[i] \wedge init = false$ and $\mathcal{O}[i] \neq false$ do
  begin
    Let $S := S \vee \mathcal{O}[i]$;
    Let $\mathcal{O}[i+1] := pre(\mathcal{O}[i]) \wedge \neg S$;
    Let $i := i + 1$
  end;
  If $\mathcal{O}[i] \wedge init \neq false$ then do
  begin
    Let $\mathcal{F}[i] := \mathcal{O}[i] \wedge init$;
    for $j := i - 1$ backto 0
       do $\mathcal{F}[j] := \mathcal{O}[j] \wedge post(\mathcal{F}[j+1])$;
    Let $\mathcal{D} := \mathcal{D} \wedge \neg\mathcal{F}[0]$;
    extract error traces from $\mathcal{F}$
  end
  else Let $\mathcal{F}[0] := false$;
end
```

Fig. 3. Counter example generation for $\mathcal{D}$

$$\begin{aligned} \mathcal{O}[0] &= \mathcal{D} \\ \mathcal{O}[i]_{i>0} &= pre(\mathcal{O}[i-1]) \setminus \bigcup_{0\leq j<i} \mathcal{O}[j] \end{aligned}$$

Thus, if a (reachable) configuration c has a path to $\mathcal{D}$, then the *shortest* path from c to $\mathcal{D}$ has length i iff $c \in \mathcal{O}[i]$. If there exists a $c_d \in \mathcal{O}[d] \cap init$, then there exists a configuration in $\mathcal{D}$ that is reachable in d steps and a counterexample can be generated by computing a series of configurations $c_i \in post(\{c_{i+1}\}) \cap \mathcal{O}[i]$ for $d > i \geq 0$. To compute an error trace for each of the reachable configurations in $\mathcal{D}$, a small overhead is added to the usual counter example generation. Instead of directly computing a single trace as described above, we compute the states that reach $\mathcal{D}$ and are reachable by c_d (depicted by the shaded area in Fig 2) as:

$$\begin{aligned} \mathcal{F}[d] &= \mathcal{O}[d] \cap init \\ \mathcal{F}[i]_{d>i\geq 0} &= post(\mathcal{F}[i+1]) \cap \mathcal{O}[i] \end{aligned}$$

Thus, $\mathcal{F}[0] \subseteq \mathcal{O}[0]$ and all error traces of length d to $\mathcal{F}[0]$ consist of transitions between adjacent sets in $\mathcal{F}$ and can be computed for each $c_0 \in \mathcal{F}[0]$ by selecting a $c_i \in pre(\{u_{i-1}\}) \cap \mathcal{F}[i]$ for $0 < i \leq d$. As we are only interested in one counterexample per configuration $c \in \mathcal{D}$, c_0 is reported to be a reachable deadlock and removed from $\mathcal{D}$ after computation. This computation is repeated until either $\mathcal{D}$ is empty, or there is a i such that $\mathcal{O}[i]$ is empty (the remaining elements in $\mathcal{D}$ are unreachable).

Theorem 1: The algorithm in Figure 3 terminates for finite state systems.

Proof: Termination is given because the inner loop terminates due to the finite state space and $\mathcal{D}$ is reduced in each step by the found reachable configurations, or the approach reached a fixed point ($\mathcal{O}[i] = false$ and thus $\mathcal{F}[0] = false$). ∎

A. Implementation and Optimization

The approach was implemented as an add-on to the *D-Finder* tool-set [7], which uses *BIP* [21] as input language and implements the various techniques for computing interaction invariants introduced in [24], [8], [25].

DFinder uses JavaBDD [29] for manipulating BDDs, which in turn allows using various BDD managers as back-ends by accessing them via the Java Native Interface (JNI). BDDs allow a succinct representation for sets of configurations and effective implementations for $pre()$ and $post()$ computation. As usual for symbolic representations [5], the transition relation is encoded using separate *current-* and *next* state variables. The order of those state variables, and the corresponding variables for the *ports*, have a great influence on the number of nodes a BDD requires to represent a relation and therefore strongly influence the size of the system we are able to verify.

To optimize for large systems, the implementation consequently requires careful handling of variable ordering and management of the memory. Two aspects have to be considered: reducing the size of BDDs by grouping variables that are strongly dependent on each other in terms of functionality, and optimization for frequently used operations like conversion between *current-* and *next* state variables [30], [31]. These considerations were especially relevant for deciding if *port* variables should use a consecutive block of the BDD variables to facilitate computations on the interactions, or be interleaved with the state variables of their respective component for a better representation of the local transitions. Experimental results showed that an interleaved ordering of *ports* and *current-* and *next* state variables performed by magnitudes better in both the time needed to compute the transition relation, and the size of its representation.

In general, a smaller set of states does not necessarily also mean a smaller BDD[1]. It can therefore be advantageous to add or remove valuations of the BDDs in order to reduce its size. This is especially important for big sets like the intermediate results from the computation of $\mathcal{O}$. To find save valuations to add or remove from the BDDs, we use our knowledge about Inv, which is an over-approximation of the reachable states. Clearly, a configuration can only be in an error trace, if it is reachable. Configurations *outside* the invariant can therefore be used to simplify the BDDs. In other words, for a BDD Z describing a set of configurations, we are looking for a smaller BDD Z', such that $Inv \implies (Z' \leftrightarrow Z)$. A simplification function that exactly provides us with this behavior is the $restrict(Z, Y)$ function from Coudert, Madre, and Berthet [32]. The function computes a BDD such that $Y \implies (restrict(Z, Y) \leftrightarrow Z)$ is a tautology.

[1]As an example, the set of all states is represented by a single BDD node for *true*.

Applying this function to our setting, $restrict(\mathcal{O}[i], Inv)$ gives us a BDD $\mathcal{O}'[i]$ which coincides with the original onion ring $\mathcal{O}[i]$ on the states that are within the invariant Inv. Because we know that states outside this invariant are not reachable, this simplification does not induce false error traces.

Theorem 2 (Preservation of error traces): Using the *restrict* function to reducing the BDD size of the intermediate values $\mathcal{O}[i]$ does not introduce new error traces.

Proof: Assume that by using $\mathcal{O}'[i] = restrict(Inv, \mathcal{O}[i])$ a new (false) error trace π was introduced. Then, because restrict only adds states in $\neg Inv$, there is a configuration c_i in π that is outside Inv and there is a prefix of π that starts at an initial state, and ends in c_i. This is in contradiction to the fact that the set of reachable states is a subset of Inv. ∎

We use the $restrict$ function to reduce the size of the intermediate sets in the computation of $\mathcal{O}$, resulting in typical reductions of 50% to 75% of the BDD size.

As a side note: Although the *pre-image* computation only returns states that lie within Inv, the restrict function in general could lead to extra states x if the pre-image of a state that is generated by the restrict function has a predecessor within Inv. However, since x can reach a state outside the invariant, but states outside the invariant cannot be reached from an initial state, we know that x itself can not be reachable. Nevertheless, to avoid the extra computation that can arise by the extra states, we restrict the transition relation to not contain any transitions between Inv and $\neg Inv$, thus avoiding x.

In the case that the computation of $\mathcal{O}$ still fails due to size restrictions, we fall back to compute $\mathcal{O}$ for single configurations of $\mathcal{D}$. In case an initial state is reached, a counterexample is created as described above. Otherwise, the result is used to strengthen the Inv as follows: The result of $pre(Z)$ contains all configurations that can reach Z in one step. Consequently, repeated pre computation on the result will give all states that can reach Z. If this set does not contain an initial state, then it is unreachable. We use this observation to iteratively strengthen the invariant (and in consequence $\mathcal{D}$) until it is strong enough to continue computations of $\mathcal{O}$ for the remaining set $\mathcal{D}$.

IV. Experimental results

To show the applicability of the approach, we use the results of a number of experiments with a prototype implementation on top of the DFinder tool-set [7], [33]. We use different versions of the Dining Philosopher example, and other benchmarks which easily can be instantiated with different domain sizes to show the scalability of the approach. Note that we do not use techniques to exploit the symmetry of the examples but use the instantiation of many components purely for generation of large models. We use the same benchmarks as in previous papers without error trace generations and added deadlocks in those examples that were deadlock-free. The experiments were performed on a MacBook Pro OS X with 2.6 MHz and

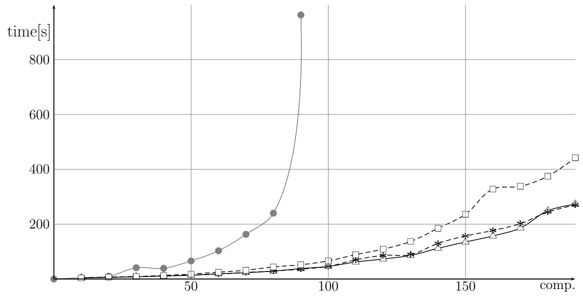

Fig. 4. Philosopher1

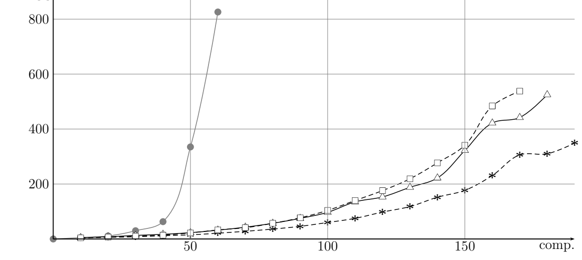

Fig. 5. Philosopher2

8 GB of memory. To analyze the strengths and weaknesses of the approach for different kinds of models, we implemented a mode for counter example generation that omits computation of invariants and reports the very first error it can find, thus basically performing standard backwards model checking (*mc*). Note that the shown results only concern examples that do have deadlocks. In many cases without deadlock, the invariant computation suffices to show the correctness of the model immediately. Furthermore, by first reporting possible deadlock suspects which are later on refined, our approach is able to give quick response to a user, who therefore might be able to quickly identify real problems.

A. Dining Philosopher

We use different versions of the well known philosopher example to examine and clarify the impact of the properties of a model on the performance of the counter example generation. The models differ in the way the philosophers are implemented, which leads to a different number of false positives and counterexamples. The first model, *philosopher1*, models the philosopher with four states and allows the exact identification of the deadlock just by computing the intersection of Inv and $\mathcal{DIS}$. This leads to a set $\mathcal{D}$ with only one entry, which is reachable from the initial state. A second model, *philosopher2*, uses only three states and gives a less exact invariant that leads to three deadlock suspects, one of which is actually reachable. To study the impact of increasing number of false positives, we add a “confused” philosopher as shown in Fig. 8. It uses a Boolean variable “confused”, which is initialized non-deterministically. If true, it enables additional transitions that allow the philosopher to pick the second fork but forget to put it down again, and to pick the forks in a different order. This behavior adds two additional reachable deadlocks and gives more false positives, whose actual number depends on the total number of philosophers (several hundred for the examined range of components).

The results are given in Fig. 4 to 6, where we use the time for standard model checking (*mc*) without invariant as benchmark (●). The approach from Section III is referred to as *rmfp* and marked by $\triangle$. For comparison, we also give the time that is needed to only compute the first counterexample without removing all false positives from $\mathcal{D}$ (*1-shot*) with ($*$) and without (□) BDD optimizations.

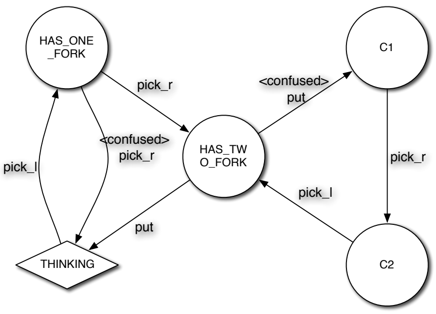

Fig. 8. Model of the confused philosopher

The first example shows a very strong performance of the approach where the use of the invariant allows us to check double the number of components then without. Because of the good invariant, *rmfp* and *1-shot* both perform only one backward search to generate a counter example because *rmfp* detects that there cannot be any more deadlocks in the model without requiring a fixed point computation. This advantage is not present anymore for philosopher2, where we have false positives in the computation of deadlock suspects. Note however, that the computation of the exact set of deadlocks with BDD optimizations is still faster then the *1-shot* computation without.

For models with larger sets of false positives, as present in the *confused philosopher* example in Fig. 6, backward computation of *rmfp* has, in spite of the invariant, to compute the pre-images of numerous states at once, which reduces the advantage it has over the pure *mc* computation. Note, though, that the gain is still considerable and that *rmfp* produces error traces for all three reachable deadlocks, while the time given for *mc* is for one error trace only. This is significantly harder, as it requires to compute a fixed point to show that the infeasible deadlocks can indeed not reach the initial state. This is also demonstrated by comparing with the results for the *1-shot* methods, where the version without BDD optimizations is not the first time faster then the full computation. We still think that giving a complete set where possible is favorable, which is the reason for using the more expensive computation of a complete set of reachable deadlocks as default setting.

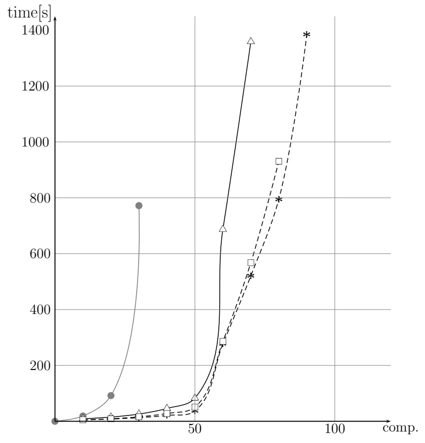

Fig. 6. Confused Philosopher

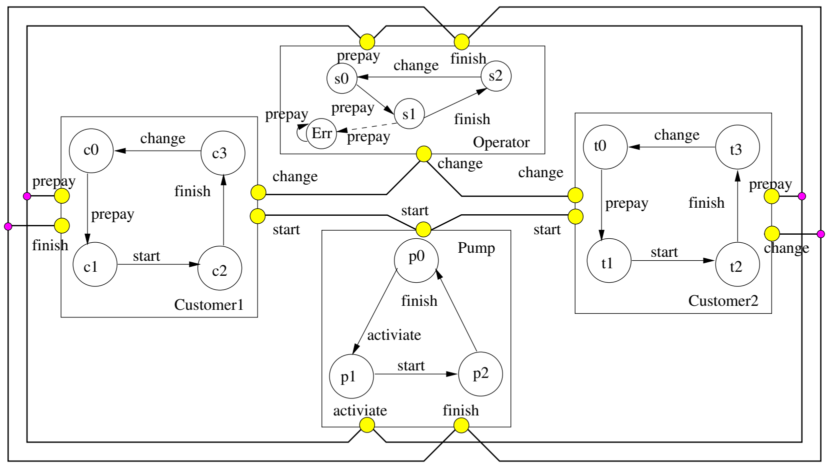

Fig. 7. Sketch of Gas Station

B. Gas Station

The Gas Station model [34] consists of an Operator with a computer, a set of pumps, and a set of customers. Each pump can be used by a fixed number of customers. The set of the atomic components involved in a system with n pumps and m customers for each pump is denoted by $B[n, m] = \{Operator, \{pump_i\}_{1 \leq i \leq n}, \{customer_{ij}\}_{1 \leq i \leq n, 1 \leq j \leq m}\}$.

Before using a pump, each customer has to prepay for the transaction. Then the customer uses the pump, collects his change and goes to a state from which he may start a new transaction. Before being used by a customer, the pumps have to be activated by the Operator. When a pump is shut off, it can be re-activated for the next operation.

Fig. 7 gives the model for Gas Station system for one pump and two customers. For that number of pumps, the Operator has two control locations and three ports. The transition labeled with $prepay$ accepts a customer's prepay and activates the pump for the customer. When a customer is served, the transition labeled with $finish$ will synchronize the pump and the customer. A pump has three control locations and three ports. Besides the synchronization between the Operator and customer through $activate$ and $finish$ ports, a pump and a customer are synchronized through $start$ ports.

In contrast to previous verification of the model, e.g., in [26], [7], [34], we introduce a deadlock by limiting the number of concurrent customers a operator can handle to be smaller than the actual number of pumps. This is done by replicating the control states to form a counter that keeps track of the number of customers that prepaid ; a sequence of *finish* and *change* reduces that counter by one. The operator moves to a deadlocking state if the number of supported customers is exceeded. For example, if Operator can handler a maximum of n customers, the prepay transition for the the $(n + 1)$th concurrent customer (in state s_n) will lead Operator to an error state from where the entire system is blocked.

Note, however, that this does not lead to an immediate deadlock of the complete system as the *customers* that already did pay still can interact with their assigned *Pumps*. Together with the fact that the deadlock only depends on the number of customers that interact with the operator, but not their particular IDs, this quickly leads to an immense number of reachable deadlocks that only differ by the combinations of customers that are interacting with a pump. This number of deadlocks exceeded several thousand for 30 customers already, which lead to the effect that generating and output of the traces actually exceeded the time required for excluding the false positives. For the results in Table I we therefore report the time taken for computing the first 10 counter examples in columns *full* and *opt-full* and compare with the *1-shot* option with and without BDD optimization as before. Columns *comps*, *locs*, and *intrs* correspond to the number of components, locations and interactions of the models. Note that in this particular case with a single initial state and shortest counterexamples of equal length, this still means that the approach computes the exact set of reachable deadlock states — only the last step of extracting the trace for all states is omitted.

The results are promising and even closer to the results from [7] then the philosopher example. In [7], without false positive elimination, we were able to compute a system with 300 pumps with the global approach in about 30 minutes. The differences are to be explained 1) by the higher complexity of the problem of exact computation and deadlock generation, but also by the more complex model that we constructed to introduce a deadlock.

C. Automatic Teller Machine

An *Automatic Teller Machine* (ATM) is a computerized telecommunication device that provides services to access financial transactions in a public space without the need for a cashier, human clerk or bank teller. The system is composed of the following components: User, ATM (modeling a cash dispenser) and Bank (modeling some aspects of bank operation). User and Bank interact only with ATM, but not with each other. Fig. 9 presents the modeling of ATM in BIP:

- Initially at location l_0, user can insert the card by *insert* transition and enter the confidential code (*enter*

TABLE I
RESULTS FOR REMOVING FALSE POSITIVES AND COMPUTING THE FIRST ERROR TRACE IN THE GAS STATION EXAMPLE

scale	comps	locs	intrs	full	opt-full	1shot	opt-1shot
5 ps×10 ctms	16	65	40	0:06	0:06	0:06	0:06
5 ps×20 ctms	26	105	80	0:11	0:10	0:10	0:11
5 ps×30 ctms	36	145	120	0:20	0:20	0:20	0:21
5 ps×40 ctms	46	185	160	0:32	0:32	0:31	0:33
5 ps×50 ctms	56	225	200	-	0:46	-	0:46
5 ps×70 ctms	76	305	280	-	1:15	-	1:15
5 ps×90 ctms	96	385	360	-	1:44	-	1:41

TABLE II
RESULTS FOR REMOVING FALSE POSITIVES AND COMPUTING THE ERROR TRACE IN THE ATM EXAMPLE

scale	comps	locs	intrs	full	opt-full	1shot	opt-1shot
5 atms	12	114	85	0:14	0:13	0:11	0:10
10 atms	22	224	170	0:32	0:27	0:18	0:28
15 atms	32	334	255	5:36	4:47	0:45	0:40
20 atms	42	444	340	4:19	2:50	2:11	1:43
25 atms	52	554	425	-	9:47	4:30	2:37
30 atms	62	664	510	-	-	-	3:19
35 atms	72	774	595	-	-	-	5:24

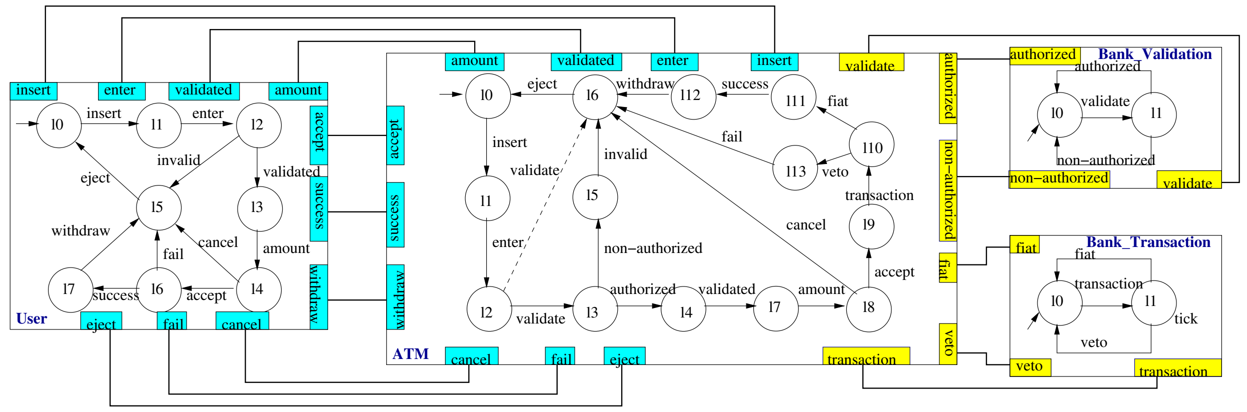

Fig. 9. Model of an ATM system in BIP

transition). Then there are two cases: if the code is invalid, user gets back the card by *eject* transition and returns to the initial state l_0; otherwise, user continues by entering the amount of cash he/she wants to withdraw. If the amount is not accepted, the transaction is canceled (*cancel* transition); else there are two cases: transition fails (*fail* transition) or is ready (*success* transition) for user to withdraw the money. Finally user gets back their card.

- Initially at location l_0, ATM is waiting for user to insert the card (*insert* transition) and then to enter the confidential code (*enter* transition). If it receives non-authorized for the code by *non_authorized* transition, *invalid* transition takes place and then it ejects the card. If it receives authorized signal by *authorized* transition, the transition *validated* takes place and it moves to a location where user can enter the amount of cash. If the user cancels the transaction or the amount is not allowed, it returns to l_6 to eject the card; else it accepts the amount and starts the transaction. If the transaction is forbidden (*veto* transition), it will announce to user by *fail* transition; else it will wait for user to withdraw the cash (*withdraw* transaction) and eject the card to finish the transaction.
- For Bank, there are two components: *BankValidation* component checks the validity of PIN code and *Bank-Transaction* component checks whether the transaction is forbidden (*veto*) or allowed (*fiat*). The use of these parallel components allows us to support multi Users and multi ATMs.

Again, we introduce a deadlock to the previously deadlock-free benchmark example to present our approach. We do this by modeling a deadlock situation where one ATM decides to cancel the transaction while waiting for the authorization from Bank Validation. This error behavior is represented by the transition *validate* from $l2$ to $l6$ in the component ATM in Fig. 9. This transition causes a deadlock because the ATM sends a request to Bank Validation to check the authorization without waiting for the response while Bank Validation is waiting for sending back the response to the ATM.

The model contains exactly one reachable deadlock, but this particular models proofs to be particularly hard for the interaction invariant computation, which quickly generates thousands of false positives even for small numbers of ATMs. Due to the large number of false positives to start the initial backward computation from, it therefore also represents a particularly hard case for the counter example generation. Especially the *one-by-one* fall-back becomes tooth-less. The results are given in Table II. We use the example to demonstrate the usefulness of the optimizations represented in Section III. The first result column gives computation times without simplification of the BDDs. Column *opt* gives the times for backward computation using the *restrict* function to reduce the size of the BDDs. As explained in Section III, this method reduces the size the onion rings in the backward computation by allowing us to add states from outside the invariant. We see that this method allows us to identify deadlocks for much greater systems than without considering the invariant. This becomes especially apparent for the two columns *1shot*, which give the time until the first counterexample is generated. Even though this instance is hard for removing all false positives of the system, comparison with previous work in [7] shows that extracting a single error trace is close to the global approach of showing the deadlock free model correct with the invariant computation and even outperforms the initial implementation of DFinder.

V. Conclusion

In a series of recent works [24], [8], [25], we have proposed efficient techniques for checking absence of deadlocks in concurrent systems. While very efficient in showing that systems are deadlock free, the approach has the drawback that it can report false positives, i.e., deadlocks might be reported for correct systems. Furthermore, no error traces are provided to assist the engineer in correcting the fault. This paper proposes a solution to this problem by combining the previous approach of invariant computation with model checking techniques. In particular, we use the computation of deadlock suspects as a pre-processing step and using the invariant to reduce the state space to search and guide in the search for error traces. Experimental results show that the approach can be very effective in practice and performs comparable to standard model checking tools in cases that are hard for the invariant approach.

Our technique is fully automatic and complete for the case of Boolean programs and was implemented as an extension of the *DFinder* tool-set. In current work we experiment with further techniques and heuristics to improve the performance of the approach. More specifically, we will exploit the component structure of the system to split the BDDs into a set of implicitly conjoined BDDs as motivated by [35], first steps of splitting the transition relation showed promising results. We also work on further strengthening the invariants, which directly will benefit to the shown approach, and started implementing an CEGAR approach to further reduce false positives in the case of non-Boolean systems. Finally, we shall extended our methodology beyond deadlock detection.

References

[1] P. Godefroid and P. Wolper, “Using partial orders for the efficient verification of deadlock freedom and safety properties,” *FMSD*, vol. 2, no. 2, pp. 149–164, 1993.

[2] G. J. Holzmann, *SPIN Model Checker, The: Primer and Reference Manual*. Addison-Wesley, 2003.

[3] D. Peled, “All from one, one for all: on model checking using representatives,” in *CAV*. Springer-Verlag, 1993, pp. 409–423.

[4] P. Wolper and P. Godefroid, “Partial-order methods for temporal verification,” in *CONCUR*, ser. LNCS, vol. 715. Springer, 1993, pp. 233–246.

[5] K. McMillan, *Symbolic model checking*. Kluwer Academic Publishers Norwell, MA, USA, 1993.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” *TACAS*, pp. 193–207, 1999.

[7] S. Bensalem, A. Griesmayer, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, “D-Finder 2: Towards efficient correctness of incremental design,” in *Nasa Formal Methods Symposium*, 2011, to be published.

[8] S. Bensalem, M. Bogza, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, “Incremental component-based construction and verification using invariants,” in *FMCAD*, 2010.

[9] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[10] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, “SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft,” in *IFM*, 2004.

[11] P. A. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine, “Constrained monotonic abstraction: A CEGAR for parameterized verification,” in *CONCUR*, 2010, pp. 86–101.

[12] A. Basu, M. Bozga, and J. Sifakis, “Modeling heterogeneous real-time components in BIP,” in *SEFM*. IEEE Computer Society, 2006, pp. 3–12.

[13] A. Basu, S. Bensalem, M. Gallien, F. Ingrand, C. Lesire, T.-H. Nguyen, and J. Sifakis, “Incremental component-based construction and verification of a robotic system,” in *ECAI*, 2008.

[14] S. Chaki, E. M. Clarke, J. Ouaknine, and N. Sharygina, “Automated, compositional and iterative deadlock detection,” in *MEMOCODE*. IEEE, 2004, pp. 201–210.

[15] C. Demartini, R. Iosif, and R. Sisto, “A deadlock detection tool for concurrent java programs,” *Softw., Pract. Exper.*, vol. 29, no. 7, pp. 577–603, 1999.

[16] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “NuSMV: a new symbolic model checker,” *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 410–425, 2000.

[17] K. Ravi and F. Somenzi, “Hints to accelerate symbolic traversal,” *Correct Hardware Design and Verification Methods*, pp. 639–673, 1999.

[18] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald, “Proof-guided underapproximation-widening for multi-process systems,” in *ACM SIGPLAN Notices*, vol. 40, no. 1. ACM, 2005, pp. 122–131.

[19] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker BLAST: Applications to software engineering,” *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy abstraction,” in *POPL*. New York, NY, USA: ACM, 2002, pp. 58–70. [Online]. Available: http://doi.acm.org/10.1145/503272.503279

[21] “BIP,” http://www-verimag.imag.fr/BIP,196.html?

[22] S. Bliudze and J. Sifakis, “The algebra of connectors — structuring interaction in BIP,” VERIMAG, Tech. Rep. TR-2007-3, 2007.

[23] M. Bozga, M. Jaber, and J. Sifakis, “Source-to-source architecture transformation for performance optimization in bip,” in *SIES*. IEEE, 2009, pp. 152–160.

[24] S. Bensalem, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, “Incremental invariant generation for compositional design,” in *TASE*, 2010, pp. 157–167.

[25] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen, “Compositional verification for component-based systems and application,” in *ATVA*, 2008, pp. 64–79.

[26] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, “D-Finder: A tool for compositional deadlock detection and verification,” in *CAV*, 2009, pp. 614–619.

[27] S. Bensalem and Y. Lakhnech, “Automatic generation of invariants,” *FMSD*, vol. 15, no. 1, pp. 75–92, 1999.

[28] J. Sifakis, “Structural properties of Petri nets,” *Mathematical Foundations of Computer Science 1978*, pp. 474–483, 1978.

[29] J. Whaley, “JavaBDD java binary decision diagram library,” http://javabdd.sourceforge.net/. [Online]. Available: http://javabdd.sourceforge.net/

[30] A. Hu and D. Dill, “Reducing BDD size by exploiting functional dependencies,” in *DAC*. ACM, 1993, pp. 266–271.

[31] S. Tani, K. Hamaguchi, and S. Yajima, “The complexity of the optimal variable ordering problems of shared binary decision diagrams,” *Algorithms and Computation*, pp. 389–398, 1993.

[32] O. Coudert, J. Madre, and C. Berthet, “Verifying temporal properties of sequential machines without building their state diagrams,” in *CAV*, 1991, pp. 23–32.

[33] “D-Finder tool,” available at http://www-verimag.imag.fr/DFinder.

[34] D. Heimbold and D. Luckham, “Debugging Ada tasking programs,” *IEEE Softw.*, vol. 2, no. 2, pp. 47–57, 1985.

[35] A. Hu, G. York, and D. Dill, “New techniques for efficient verification with implicitly conjoined BDDs,” in *Annual ACM IEEE Design Automation Conference: Proceedings of the 31 st annual conference on Design automation*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 1994.