Proof-Guided Underapproximation-Widening for Multi-Process Systems

Orna Grumberg
Computer Science Department
Technion Haifa, Israel
orna@cs.technion.ac.il

Flavio Lerda
Computer Science Department
Carnegie Mellon
Pittsburgh, PA, USA
lerda@cmu.edu

Ofer Strichman
Information Systems Engineering
Faculty of Industrial Engineering
Technion Haifa, Israel
ofers@ie.technion.ac.il

Michael Theobald
Computer Science Department
Carnegie Mellon
Pittsburgh, PA, USA
theobald@cs.cmu.edu

ABSTRACT

This paper presents a procedure for the verification of multi-process systems based on considering a series of underapproximated models. The procedure checks models with an increasing set of allowed interleavings of the given set of processes, starting from a single interleaving. The procedure relies on SAT solvers' ability to produce proofs of unsatisfiability: from these proofs it derives information that guides the process of adding interleavings on the one hand, and determines termination on the other. The presented approach is integrated in a SAT-based Bounded Model Checking (BMC) framework. Thus, a BMC formulation of a multi-process system is introduced, which allows controlling which interleavings are considered. Preliminary experimental results demonstrate the practical impact of the presented method.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification

General Terms

Algorithms, Reliability, Verification

Keywords

Abstraction, Bounded model checking, SAT proofs, Software Verification, Underapproximation-Widening

1. INTRODUCTION

We present a procedure for model checking multi-process systems based on considering a series of underapproximated models. The procedure checks models with an increasing set of allowed interleavings of the given set of processes, starting from a single interleaving. Only in the worst case, the procedure attempts to verify a model with the same amount of concurrency as the original one has.

The procedure considers underapproximations with increasing amount of concurrency until one of two possible termination conditions holds:

- The procedure finds a **counterexample**: in this case, since the considered model is an underapproximation of the original one, it concludes that the property is violated.

- The procedure proves that all traces in the underapproximated model satisfy the property, and the proof **does not rely on the underapproximation**. In this case, the proof can also serve as a proof for the original model. Hence, the procedure concludes that the property holds.

The *Underapproximation-Widening* (UW) of the model (widening is the term we use for 'adding behaviors') progresses by adding concrete behaviors, and this way removes 'false positive' results. Unlike Abstraction-Refinement (AR), which is based on iterations of overapproximation and narrowing (removing spurious behaviors) and can be fully automated [20, 10, 11, 3, 22, 4], we are not aware of a completely automated UW procedure for model checking of either a single or multiple processes. The reason for the difference is that refinement of overapproximated models in AR is easier to *guide*, because it can rely on spurious counterexamples generated by the model checker. Indeed, work on counterexample-guided AR [13, 11] showed how to gradually remove spurious counterexamples by investigating the so called 'failing state', i.e., the first state in the spurious trace that cannot be simulated on the concrete model. Such guidance does not naturally exist in the UW framework, because there is no counterexample to show what went wrong.

A recent investigation into deriving proofs of unsatisfiability from state-of-the-art Boolean satisfiability solvers (SAT solvers) [2, 9]

enables us to design a fully automated UW process. SAT solvers are used in model checking to verify the validity of a property. In particular, we can check whether a generated proof of unsatisfiability – which is a proof of the validity of the property – relies on the underapproximation, and if so, we can guide the widening process according to the variables that participate in the proof. Both of the above references [2, 9] exploit proofs of unsatisfiability in an AR process, in contrast to this paper.

To the best of our knowledge, the presented UW approach is the first underapproximation model checking algorithm that is fully automated. There are several approaches that use sequences of underapproximations in the context of Binary Decision Diagrams (BDDs) based model checking, in order to keep the size of the BDDs small [25, 26, 5], but they include no explicit notion of automatic widening: the sequence is given a-priori by the user. Our method, on the other hand, computes the initial underapproximation and successively widened underapproximations automatically.

Since we generate underapproximations by limiting the number of interleavings of concurrent components, our work is aimed at multi-process systems. Another technique, also addressing the exponential blow-up due to interleavings, is *partial-order reduction* [17, 28, 24, 1, 18, 16]. The way we compute underapproximations is similar to the way partial-order techniques compute a reduced system. However, partial-order reduction does not work iteratively: it simply generates a restricted model that is guaranteed to have the same temporal properties over visible behavior (i.e. properties expressed by the global variables of the system, disregarding variables that are local to the processes) as the original model. Our technique is more aggressive than partial-order reduction, since it considers models with fewer interleavings. In more fortunate cases it finds errors in the underapproximated model, which reflects a smaller state space and, hence, it is typically easier to analyze. Moreover, we use information derived from the proof of unsatisfiability to decide which additional behaviors to introduce at the next iteration. We benefit from the fact that a bug in a multi-process system typically corresponds to multiple counterexamples and every counterexample corresponds to a single interleaving. The proposed proof-guided approach for adding behaviors often results in underapproximated models that exhibit the error but include much fewer interleavings than the original model.

Since our approach relies on the ability of SAT solvers to generate proofs of unsatisfiability, it is incorporated in a Bounded Model Checking (BMC) [8, 6] framework. BMC is an iterative procedure that checks the validity of a property for traces of increasing length. At each iteration the property is checked for traces of fixed length by encoding them as a Boolean formula. A SAT solver is used to determine satisfying assignments of such a formula, which represent violations of the property. We use Underapproximation/Widening to improve this verification step. Since BMC is only able to prove a property for bounded executions, it is mainly aimed at property falsification, i.e., finding counterexamples. Our approach extends this technique for the case of multi-process systems.

The remainder of the paper is organized as follows. Section 2 presents preliminaries. Section 3 describes the Underapproximation-Widening algorithm for the BMC/SAT framework. Section 4 describes how to apply the algorithm to multi-process systems. Section 5 presents experimental results, and Section 6 gives conclusions.
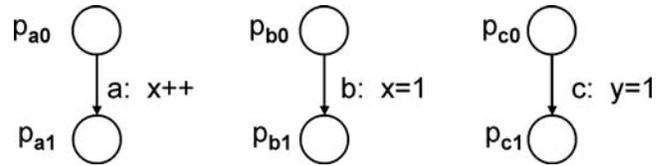
Figure 1: The processes of a multi-process system. Each process is given as a control-flow graph in which nodes are control points and edges are statements.

2. PRELIMINARIES

2.1 Multi-Process Systems

We consider asynchronous multi-process systems, which can be defined as the composition of N processes $P_1, \ldots, P_N$ according to the standard interleaving semantics. Processes communicate by reading and writing to shared variables that are accessible to all processes.

Processes are modeled in the *guarded command* framework [14]. Each process can be expressed as a control-flow graph: nodes represent control points, edges represent transitions (or statements). At every control point, there can be none, one, or multiple *enabled* transitions. Each transition has a guard associated to it: intuitively a transition is enabled if the condition that guards it is true in the current state and the process to which it belongs is at the control point corresponding to its source. Consider, for example, the following code fragment:

```
if
::  b > 1 -> a = 1;
::  b > 2 -> a = 2;
fi;
```

At the initial control point, if the current state is such that $b > 2$, then both transitions are enabled, because their guards (the conditions appearing to the left of '->') are true. In this case, one of the two transitions is selected non-deterministically. If, on the other hand, $b \leq 1$, no transition is enabled and the process stops until, possibly, b is updated by another process. We say that a process is *enabled* in a state s if at least one of its transitions is enabled in s.

EXAMPLE 1. *Figure 1 shows the control-flow graphs of three simple processes: A, B, and C. Each has a single transition. The guards of all transitions are* TRUE*, thus only the process' control point determines whether a transition is enabled or not. For instance, the transition labeled **a** is enabled if and only if process A is at control point P_{a0}. Figure 2 (top) presents the control-flow graph of the multi-process system consisting of processes A, B, and C. The top part of the figure includes all possible interleavings. Note that, in the initial control point (upper left node) all transitions are enabled since all processes are in their initial control points. Once a transition **a** is taken, **a** becomes disabled and only transitions **b** and **c** remain enabled. The middle and bottom control-flow graphs in Figure 2 present two possible underapproximations of the original control-flow graph, where only the marked transitions can be taken.*

2.2 Partial and Full Expansions

Although, in general, it is necessary to consider all possible interleavings to verify the correctness of a multi-process system, there has been a lot of work on reducing the number of interleavings that
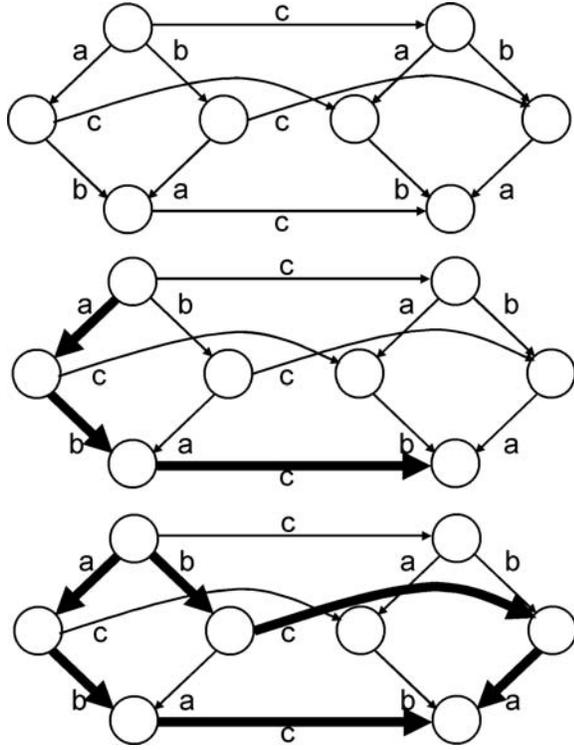
Figure 2: All interleavings of a multi-process system and two underapproximations, given as control-flow graphs.

are actually considered. This section introduces some basic terminology.

For a given state of a multi-process system, we distinguish between **full expansion** and **partial expansion** steps:

- A full expansion generates the next states for all enabled transitions.[1]

- A partial expansion generates the next states for a subset of all enabled transitions. We will consider only subsets that correspond to all enabled transitions of *one* process.

In this paper, therefore, full expansions correspond to allowing all enabled processes to execute next and partial expansions correspond to fixing one process that is to execute next — each of the enabled transitions of this process may be taken.

The technique presented in this paper uses partial expansion steps to define underapproximations. Restricting the set of transitions that can be executed at each state defines a reduced model that disallows some of the interleavings that are allowed in the original model. Another technique based on partial and full expansions is partial-order reduction (POR) [15]. Unlike our approach, however, partial-order reductions build a reduced model which agrees with the original model on both property verification and refutation. As mentioned earlier, the proposed UW approach is much more aggressive than partial-order reduction in using partial expansions. In particular, each underapproximation is guaranteed to agree with the original model only on property refutation.

---

[1]Note that each process may have (multiple) enabled transitions – each of the transitions in the union of the enabled transitions of all processes may be executed next.

2.3 SAT and Proofs of Unsatisfiability

We assume that the reader is familiar with Boolean satisfiability (SAT) and formulas in conjunctive normal form (CNF formulas). Here we mention several well-known terms and observations that we will use to justify the correctness of our method.

2.3.1 Underapproximation of CNF Formulas

Let φ be a CNF formula and let φ' be a CNF formula obtained from φ by adding to it additional clauses, i.e. $\varphi' = \varphi \wedge \bigwedge_{i=1}^{n}(c_i)$ where $c_1, \ldots, c_n$ are arbitrary clauses referring to φ's variables. Then it is easy to see that

$$\varphi \text{ is unsatisfiable} \rightarrow \varphi' \text{ is unsatisfiable} \qquad (1)$$

Notice that the other direction does not hold. That is, it is possible that φ is satisfiable while φ' is not due to the additional constraints $c_1, \ldots, c_n$. φ' can be thought of as an *underapproximation* of φ (or φ as a *widening* of φ') since the set of *satisfying interpretations* to φ' is a subset of the satisfying interpretations to φ. If φ' underapproximates φ, every satisfying interpretation to the former, is a satisfying interpretation to the latter, or, more formally, for every interpretation α it holds that:

$$\alpha \models \varphi' \rightarrow \alpha \models \varphi \qquad (2)$$

assuming the added clauses only refer to φ's variables.

2.3.2 Resolution Proofs and Unsatisfiable Cores

A key notion in our discussion is the concept of *resolution proofs*. When a complete SAT solver[2] concludes that there is no satisfying assignment to a given instance, its internal steps for reaching this conclusion can be used to construct a resolution proof, i.e. a sequence of deduction steps based on the rule

$$\frac{a_1 \vee \ldots \vee a_n \vee \beta \qquad b_1 \vee \ldots \vee b_m \vee (\neg\beta)}{a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m}$$

where $a_1, \ldots a_n, b_1, \ldots b_m, \beta$ are literals. Modern SAT solvers such as zChaff can output such a sequence that serves as an independently checkable proof of unsatisfiability.

The sequence of deduction steps can be represented as a directed acyclic graph (DAG), where the nodes are clauses and there is a directed edge from node c to c' iff the clause c participates in the deduction of c'. Hence, every node is either a root or has two incoming edges. The root nodes are clauses from the original formula. The internal nodes represent clauses that are deduced by the SAT solver at run-time (typically these are deduced when the SAT solver detects a conflict due to a bad assignment, and are therefore known as *conflict clauses*).

An *unsatisfiable core* of an unsatisfiable CNF formula φ is a subset of its clauses that is unsatisfiable. Let S be the set of clauses in a given unsatisfiable CNF formula φ and let $\overline{S}$ be the set of root nodes in the DAG corresponding to a resolution proof for the unsatisfiability of φ. Since the conjunction of the clauses in the set $\overline{S}$ is unsatisfiable by itself, $\overline{S}$ is an unsatisfiable core of φ. There are three observations regarding $\overline{S}$ that are important for our discussion:

1. $\overline{S}$ is equal to or a subset of S ($\overline{S} \subseteq S$). Typically the SAT solver concludes that an instance is unsatisfiable without using all of its constraints. This is possible because some of the constraints may be redundant.

---

[2]A complete SAT solver is a SAT solver that is guaranteed to find, in a finite amount of time, a satisfying assignment if one exists.

2. By Equation 1, every CNF formula φ', with a set of clauses S' such that $\overline{S} \subseteq S'$, is unsatisfiable, and $\overline{S}$ is an unsatisfiable core of φ' as well. More formally:

THEOREM 1. *Let φ and φ' be two unsatisfiable CNF formulas such that the set of clauses of φ is contained in the set of clauses of φ'. Then if $\overline{S}$ is an unsatisfiable core of φ, it is also an unsatisfiable core of φ'.*

PROOF. $\overline{S}$ is contained in the set of φ's clauses and hence also in set of clauses of φ'. Therefore, since $\overline{S}$ is unsatisfiable, it is an unsatisfiable core of φ'. $\square$

In the general case, the other direction of Theorem 1 does not hold. However:

THEOREM 2. *Let $\overline{S}'$ be an unsatisfiable core of a formula $\varphi' = \varphi \wedge \bigwedge_{i=1}^{n}(c_i)$, and assume that for all $1 \leq i \leq n$, $c_i \notin \overline{S}'$. Then φ is unsatisfiable and $\overline{S}'$ is an unsatisfiable core of φ.*

PROOF. $\overline{S}'$ is contained in the set of clauses of φ and is unsatisfiable. Therefore, by Equation 1, φ is unsatisfiable and $\overline{S}'$ is an unsatisfiable core of φ. $\square$

3. The set $\overline{S}$ is not unique. In particular, there can be more than one proof of unsatisfiability, each of which can result in a different set of root nodes. It is an NP-hard problem[3] to find the minimum unsatisfiable core, and indeed SAT solvers do not attempt to find such minimum cores. Rather, they produce a proof that reflects the internal steps that led them to this conclusion. These steps are determined by various heuristics that typically do not find the shortest proof or the minimum unsatisfiable core.

2.4 SAT-Based Bounded Model Checking

SAT-Based Bounded Model Checking (BMC) is a procedure that given a model M, a Linear-time Temporal Logic (LTL) formula φ, and a bound k, checks whether $M \models_k \varphi$, i.e, if all possible traces of M of length k satisfy φ (cf. Figure 3). The procedure *check* constructs a propositional formula that is satisfiable if and only if $M \not\models_k \varphi$. The formula can be expressed as the conjunction of two sub-formulas:

- $bmc_{M,k}$ whose satisfying assignments correspond to the possible execution traces of length k of model M;

- $violates_\varphi$ whose satisfying assignments correspond to the set of traces of length k that violate the property φ.

Their conjunction represents the bounded traces of length k of the model which violate the property. This formula is then given to a SAT solver:

- if the solver returns 'Satisfiable', it implies that the property does not hold (in this case the satisfying assignment serves as a counterexample to the proposition $M \models \varphi$);

- if, on the other hand, it returns 'Unsatisfiable', we can conclude that $M \models_k \varphi$, although we cannot conclude that $M \models \varphi$ because there might be a violation of length greater than k.

Thus, in the latter case, k is increased and the procedure is repeated. BMC terminates either when an error is found, the Completeness Threshold CT [19, 12] is reached or, what is more common, the problem becomes intractable. Therefore BMC is mainly used for falsification rather than for verification.

[3]To be more accurate, it is Σ_2^p-complete problem, which is the same complexity as one alternation QBF.

```
main(M,φ) {
  k = 0
  repeat forever:
    if check(M, φ, k) = 'counterexample':
      return 'property fails (M ⊭ φ)'
    k = k + 1
}


check(M, φ, k) {
  if SAT(bmc_{M,k} ∧ violates_φ) = 'Satisfiable':
    return 'counterexample'
  else
    return 'valid'
}
```

Figure 3: *Bounded Model Checking* algorithm. At each iteration of the loop of the *main* procedure, *check* is invoked to test if there exists a counterexample of length k.

3. BOUNDED MODEL CHECKING USING UNDERAPPROXIMATION-WIDENING

3.1 Underapproximation-Widening

The Underapproximation-Widening (UW) procedure proves or disproves a property by considering a series of underapproximations of a given model. The procedure *uw* (see Figure 4) consists of a loop: at each step, an underapproximation of the original model is considered and the validity of the property on the underapproximated model is checked (*verify* procedure) with two possible outcomes:

- a counterexample is found. Since the counterexample is a valid execution of the underapproximated model, it must also be a valid execution of the original model and therefore can be used to disprove the property under consideration, and the procedure terminates.

- no counterexample exists. It is then necessary to generate a proof that the underapproximated model satisfies the property (*proof_of* procedure). Given such a proof, two cases are possible:

  - the proof depends on the underapproximation itself, i.e., it does not apply to the original model. In this case, information from the proof can be used to determine a new underapproximated model (*widen* procedure), and the procedure is repeated.

  - the proof **does not** depend on the underapproximation itself, i.e., it applies to the original model. In this case, the proof can be used to show that the property holds on the original model, and the procedure terminates.

The presented Underapproximation-Widening procedure can be considered a dual approach to the Abstraction-Refinement (AR) technique. However, while all model checkers are able to produce a counterexample if the property is violated, most approaches do not produce a formal proof that the model does not violate the property. This limits the applicability of the Underapproximation-Widening algorithm to approaches that are able to produce such proofs. Hence, in this paper, we apply this algorithm to SAT-based Bounded Model Checking.

```
uw(M, M₀) {
  i = 0
  while true:
    if verify(Mᵢ, φ):
      proof = proof_of(Mᵢ, φ);
      if is_valid_for(proof, M):
        return VALID;
      else
        M_{i+1} = widen(Mᵢ, proof);
        i++;
    else
        return INVALID;
}
```

Figure 4: *Underapproximation-Widening* **algorithm. The** *uw* **procedure receives as arguments a model** M **and an initial underapproximation** M_0. **At each iteration of the loop, an underapproximated model is analyzed, which either violates the property, or can be proven to be correct. The proof is used to either terminate or** *widen* **the model.**

3.2 A Framework for BMC using Underapproximation-Widening

The Underapproximation-Widening (UW) procedure described above can be used in the framework of Bounded Model Checking to improve the verification for a given bound.

This section presents a general framework that can be instantiated with various methods to perform underapproximation. This framework is the basis for the procedure introduced in the next section that can be used to improve Bounded Model Checking of multi-process systems, by checking underapproximated models with an increasing set of allowed interleavings of the given processes, starting from a single interleaving.

Bounded Model Checking consists of a main loop which performs the verification for an increasing bound k (procedure *main* in Figure 3), and a procedure that performs the verification for a given bound k (procedure *check* in Figure 3). The main loop is repeated until an error is detected or the problem becomes intractable.

The Underapproximation-Widening procedure is used to perform each of the verification steps for a given bound k more efficiently. In particular, for a given bound k, procedure *main_uw* calls the procedure *check_uw* (in Figure 5), which tries to prove or disprove the property $M \models_k \varphi$ by means of a finite sequence of underapproximations of the original model M.

The set P in procedure *check_uw* denotes the set of additional clauses that are used to define an underapproximation. The set P is initialized with a finite set of clauses that may depend on the model M and the bound k. The formula $bmc_{M,k}$ is determined as usual, however, the SAT solver is given the conjunction of $bmc_{M,k} \wedge \bigwedge_{c_i \in P}(c_i)$, which encodes a subset of the bounded traces of the original model M, and $violates_\varphi$, which encodes the bounded traces that violate the property.

If the SAT solver returns 'Satisfiable', the procedure returns 'counterexample'. Otherwise, based on previous work [2, 9], we derive an *unsatisfiable core* S of the SAT instance, i.e., a subset of its clauses that are by themselves unsatisfiable. The set of clauses in the unsatisfiable core corresponds to the roots of the proof of unsatisfiability generated by the SAT solver. From this core, we compute ψ, the set of clauses of P that belong to the unsatisfiable core of the formula. If the set ψ is empty, the procedure returns 'valid'; otherwise one of the clauses belonging to ψ is removed from P: we call this process *widening*, because at the next

```
main_uw(M,φ) {
  k = 0
  repeat forever:
    if (check_uw(M, φ, k) = 'counterexample':
      return 'property fails (M ⊭ φ)'
    k = k + 1
}


check_uw(M, φ, k) {
  P = initial_clauses(M, k)
  repeat forever:
    if SAT(bmc_{M,k} ∧ ⋀_{c_i∈P}(c_i) ∧ violates_φ) =
                                'Satisfiable':
      return 'counterexample'
    S = unsatisfiable core
    ψ = S ∩ P
    if ψ = ∅:
      return 'valid'
    Remove from P a clause in ψ
}
```

Figure 5: *Underapproximation-Widening* **algorithm. The procedure** *check_uw* **considers a set of underapproximations of model** M **until it finds a counterexample or it is able to prove the property without relying on the underapproximation.**
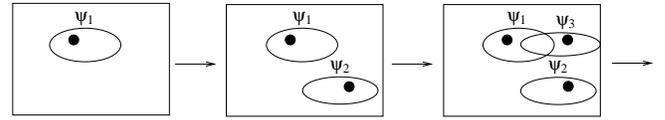
Figure 6: $\psi_1 \ldots \psi_3$ **are sets of clauses of** P **that belong to the unsatisfiable core at each step. The boxes denote the initial set of clauses in** P. **The widening step selects a single clause from each unsatisfiable core (denoted by a black dot) and removes it from the set** P. **A proof in step** i **cannot rely on clauses removed from** P **in previous steps.**

iteration a formula that represents additional behaviors is going to be considered. The procedure is repeated until one of the previous termination conditions becomes true.

The progress of *check_uw* is depicted schematically in Figure 6. This figure shows three successive iterations of widening and the clauses that are removed at each step (see caption).

3.3 Correctness and Termination

PROPOSITION 3. *If the procedure check_uw(M, φ, k) returns, it returns* 'valid' *if and only if* $M \models_k \varphi$.

PROOF. ($\rightarrow$) *check_uw* returns 'valid' only if for the current set P, the formula $bmc_{M,k} \wedge \bigwedge_{c_i \in P}(c_i) \wedge violates_\varphi$ is unsatisfiable and none of the clauses in P belong to the unsatisfiable core of the formula. By Theorem 2, the formula $bmc_{M,k} \wedge violates_\varphi$ must be unsatisfiable, and, therefore, $M \models_k \varphi$.

($\leftarrow$) If $M \models_k \varphi$ holds, the formula $bmc_{M,k} \wedge violates_\varphi$ must be unsatisfiable. By Equation 1, the formula $bmc_{M,k} \wedge \bigwedge_{c_i \in P}(c_i) \wedge violates_\varphi$ is also unsatisfiable for any set of clauses P. Therefore, the procedure cannot return 'counterexample', because this only occurs if, for some set P, the previous formula is satisfiable. Since the procedure can only return 'counterexample' or 'valid', the procedure must have returned 'valid'. □

PROPOSITION 4. *The procedure check_uw(M, φ, k) always terminates.*

PROOF. In each iteration of its main loop, the procedure *check_uw* either returns 'counterexample' if the formula is satisfiable, returns 'valid' if none of the clauses in P belongs to the unsatisfiable core of the formula, or removes one clause from P and repeats the loop.

Since P is initialized with a finite set of clauses and at each step the procedure either returns or removes one clause from P, the procedure must terminate after a finite number of iterations. Specifically, if the set P is empty, the body of the loop corresponds to checking the property φ on the original model M: if the property does not hold, the corresponding formula is satisfiable and the procedure returns 'counterexample'; otherwise the formula is unsatisfiable and, since the set P is empty, the set ψ must be empty as well and the procedure returns 'valid'. $\square$

4. UNDERAPPROXIMATION-WIDENING FOR MULTI-PROCESS SYSTEMS

This section describes how the Underapproximation-Widening loop can be used to improve Bounded Model Checking of a multi-process system. In particular, we will define a family of underapproximations obtained by limiting the allowed interleavings of the different processes. The work presented here represents one possible way of defining underapproximations: it is however of particular interest since Bounded Model Checking is often unable to cope well with concurrent systems. Other families of underapproximations can be defined, for both a single or multiple processes, but they are beyond the scope of this paper.

4.1 Modeling a Multi-Process System

We formalize the model M of a multi-process system as follows:

$$M = \langle S, I, P, V \rangle \qquad (3)$$

where

- S is a set of states;
- $I \subseteq S$ is a set of initial states;
- $P = \{P_1, \ldots, P_N\}$ is a set of processes.
- $V = \{v_1, \ldots, v_m\}$ is a set of variables.

The set S is determined as follows. Given the set of variables V, let $D_1, \ldots, D_m$ be their domains. Then $S = D_1 \times \ldots \times D_m$, i.e., S is the cross-product of the domains of the variables, and represents all the possible states. Note that not every state in S is necessarily reachable from one of the initial states in I.

Each process itself is a tuple:

$$P_i = \langle T_i, pc_i, n_i, \langle at_{i,1}, \ldots, at_{i,n_i} \rangle \rangle \qquad (4)$$

where:

- $T_i : S \times S \to \{0, 1\}$ is the transition relation for process i, i.e., $T_i(s, s')$ holds if and only if s' is a successor of s obtained by executing a transition of process P_i;
- $pc_i \in V$ is a special variable called the *program counter* whose domain is the set of control points in process P_i;
- $n_i \in N$ is the number of control points in process P_i;

- $at_{i,j} : S \to \{0, 1\}$ is a predicate over the set of states that holds for a state s if and only if process P_i is at control point j in state s, i.e., pc_i has the value j in s.

In this model, we do not discriminate between local and global variables; local variables are those variables that are syntactically accessible only by one process. Such distinction is not necessary for our approach, and a model that makes such a distinction can always be translated into a model as we defined it by variable renaming. Moreover, we do not require for the transition relations T_i to be total, i.e., a state s does not need to have a successor. However, we assume that processes have control points: in the case of models of concurrent software, control points can correspond to the nodes of a control-flow graph.

4.2 Bounded Model Checking for Multi-Process Systems

In order to perform Bounded Model Checking, we need to define the formula $bmc_{M,k} \wedge violates_\varphi$ that encodes all possible counterexamples of length k for model M. For simple invariant properties, the formula has the form:

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge F(s_k) \qquad (5)$$

where

- $I(s_0)$ enforces the first state to be one of the initial states;
- $T(s_h, s_{h+1})$ constraints consecutive states to be related by the transition relation T;
- $F(s_k)$ is derived from the property being checked and asserts that the final state violates the property.

The satisfying assignments to Formula (5) are those valid bounded traces of model M – i.e., the first state of the trace is an initial state and each pair of successive states is related by the transition relation – that violate the property.

A similar formula can be constructed in order to check a system with respect to an LTL property: $F(s_k)$ is then replaced by a more complex formula [7], which may also involve the intermediate states. The construction introduced in this section is concerned with limiting the traces of the model, and is applicable to the verification of LTL formulas as well.

The model as described in Section 4.1 does not directly define the transition relation T for the whole model, but only the transition relations T_i for each of the processes. It is possible to compute the transition relation T from the transition relations T_i in the following way:

$$\begin{aligned} T(s, s') &= \exists 1 \le i \le N : T_i(s, s') \\ &= \bigvee_{i=1}^{N} (ps = i \wedge T_i(s, s')) \qquad (6) \end{aligned}$$

where ps is a free variable that can assume any value between 1 and N.

The meaning of this formula is that s' is a successor of s, if there exists a value for ps such that s' is a successor of s when process P_{ps} is executed. The variable ps, called the *process selector*, indicates which process is going to execute next. In Equation (6), since there are no constraints on the values ps can assume, every interleaving is possible. In order to build the transition relation for a model that only contains a subset of the interleavings, we place restrictions on the values that the variable ps can assume. This will be explained in the next two subsections.

4.3 Introducing the Predicates

We want to define underapproximations by limiting the amount of interleavings in a multi-process system. To do so, we will use the concepts of *partial and full expansion* introduced in Section 2.2.

At each state, we have two possibilities:

- perform a full expansion step;

- perform a partial expansion step that corresponds to the transitions of a single process[4].

Therefore, for each state, we need to determine what kind of step we want to perform. This information will define an underapproximation of the original model: for instance, if for every state we perform a full expansion, we obtain the original model.

However, a simple mapping from each state to the particular type of step that we need to perform would be too expensive to manage since there is a large number of potentially reachable states.

Instead, we will use only part of the information from the current state to make the decision: in particular, we will only consider the current control points of the different processes. However, if the system is made of n processes with m control points each, there are still m^n possible combinations: this is clearly not going to scale to large systems. Therefore, instead of considering all combinations, we will use the following strategy:

- each control point is associated with a predicate: the predicate being true represents the fact that the process it belongs to cannot be used for a partial expansion;

- for any given state, if none of the control points can be used for a partial expansion, i.e., all predicates corresponding to the current control points of each process are true, then a full expansion is performed;

- if at least one of the processes can be used for a partial expansion, i.e., the predicate associated with its current control point is false, then a partial expansion is performed that expands the transitions of the first process according to some predefined fixed order that can perform a partial expansion.

Intuitively, the predicate associated with a given control point being true means that the statements of the corresponding process at that specific location should be interleaved with other processes. This allows encoding an underapproximation using only $m \times n$ boolean predicates.

Formally, given a model M, we introduce a set of predicates $p_{i,j}$ for every i and j such that $1 \leq i \leq N$ and $1 \leq j \leq n_i$. Predicate $p_{i,j}$ is associated with control point j of process P_i. Let's assume that in state s each process P_i is at some control location j. Let $\check{P}(s)$ be the set of processes that at state s are enabled[5] and their corresponding predicate $p_{i,j}$ is FALSE. The successors of state s are limited as follows:

- if $\check{P}(s)$ is not empty, then only *one* of the processes from $\check{P}(s)$ is expanded, i.e. a partial expansion is performed. Let the transition relation of the underapproximated model be T': then $\forall s' \, T'(s,s') = T_i(s,s')$ must hold for some i such that $P_i \in \check{P}(s)$.

---

[4]A partial expansion can be defined to expand any subset of the enabled transitions, however, for simplicity, we chose to consider only partial expansions that expand all enabled transitions of a single process.
[5]A process is enabled in s, if it has at least one successor state for s.

- otherwise, $\check{P}(s)$ is empty and $\forall s' \, T'(s,s') = T(s,s')$ must hold, i.e., all enabled transitions at s may execute.

As described in Section 3, the underapproximation is determined by a set of clauses P. In our formulation of UW for multi-process systems, the initial value for the set P will contain a negative unit clause $(\neg p_{i,j})$ for each of the predicates corresponding to control points in model M. When a clause is removed from the set P, the corresponding predicate is left unconstrained, i.e., it can be either TRUE or FALSE.

It can be shown that the formula we are constructing is equivalent to the formula corresponding to the original model M if all predicates are left unconstrained. Intuitively, if all predicates are unconstrained, it is possible to perform both a partial and a full expansion: since the original model always performs a full expansion and the set of next states corresponding to a partial expansion is a subset of the ones corresponding to a full expansion, they allow the same behaviors.

4.4 The Transition Relation of the Underapproximated Model

We now formalize the transition relation T'. We construct T' in a way that enables us to control the amount of interleavings it represents, by limiting the values the process selector variable ps can assume.

The strategy described in the previous section can be formulated by adding a predicate $valid(s, ps)$ to the formula for the transition relation. This predicate must be true for all values of ps if, at state s, all enabled processes are at control points whose predicates are true. Otherwise, it must be true only for a single value of ps, such that P_{ps} is at a control point whose predicate is false and at least one transition of P_{ps} is enabled. Therefore, the formula for T', presented below, depends on whether the predicates $p_{i,j}$ are true or false.

We first define a predicate $enabled_i : S \rightarrow \{0,1\}$ over the states such that $enabled_i(s)$ is true if and only if there exists a transition in T_i from state s to an arbitrary state s':

$$enabled_i(s) = \exists s' \in S : T_i(s,s') \tag{7}$$

We then define $partial_i(s)$ as:

$$
\begin{aligned}
partial_i(s) &= enabled_i(s) \wedge (\forall 1 \leq j \leq n_i : (at_{i,j} \rightarrow \neg p_{i,j})) \\
&= enabled_i(s) \wedge \left(\bigwedge_{j=1}^{n_i} (at_{i,j} \rightarrow \neg p_{i,j}) \right) \tag{8}
\end{aligned}
$$

which is true if process P_i is enabled and at a control point whose predicate is false, i.e. $partial_i(s) \Rightarrow P_i \in \check{P}(s)$.

If there is only one process in $\check{P}(s)$, then it is sufficient to assert that $partial_{ps}(s)$ holds, i.e., that process P_{ps}, whose transitions are going to be taken, can be partially expanded (the corresponding $p_{ps,j}$ is false). However, this is not sufficient for a state s where multiple enabled processes are at control points whose predicates are false. Therefore, we introduce an additional predicate $first\_partial_i(s)$ defined as follows:

$$
\begin{aligned}
first\_partial_i(s) &= partial_i(s) \wedge (\forall 1 \leq i' < i : \neg partial_{i'}(s)) \\
&= partial_i(s) \wedge \left(\bigwedge_{i'=1}^{i-1} \neg partial_{i'}(s) \right) \tag{9}
\end{aligned}
$$

which is guaranteed to be true for at most one of the processes.

Finally, we need to discriminate between the case where, at state s, there exists at least one enabled process that can be partially expanded, and the case where all enabled processes must be expanded. Therefore, we define:

$$
\begin{aligned}
exists\_partial(s) &= \exists 1 \leq i \leq N : partial_i(s) \\
&= \bigvee_{i=1}^{N} partial_i(s)
\end{aligned}
\qquad (10)
$$

We can now write the predicate that constrains ps as:

$$
valid(s, ps) = (exists\_partial(s) \rightarrow first\_partial_{ps}(s)) \qquad (11)
$$

Note that, if $exists\_partial(s)$ holds then $valid(s, ps)$ holds for exactly one value of ps. On the other hand, if $exists\_partial(s)$ does not hold, then $valid(s, ps)$ is true for every value of ps. The latter is exactly the case where full expansion is performed.

The formula for the transition relation $T'(s, s')$ can be expressed as:

$$
T'(s, s') = \exists ps \left(valid(s, ps) \wedge \bigvee_{i=1}^{N} (ps = i \wedge T_i(s, s')) \right) \qquad (12)
$$

It is now possible to use the expression obtained for the transition relation to write the formula that expresses all bounded traces of the reduced model:

$$
I(s_0) \wedge T'(s_0, s_1) \wedge \ldots \wedge T'(s_{k-1}, s_k) \qquad (13)
$$

Notice that this formula, if the predicates are left unconstrained, corresponds to all the bounded traces of model M. This formula will be conjoined during the execution of procedure *check_uw* (cf. Section 3) with the clauses in P. Initially, the set P will contain a negative unit clause for each of the predicates corresponding to the control points of M. The clauses in P constrain these predicates to false, therefore limiting the amount of allowed interleavings.

EXAMPLE 2. *Let us consider again the system from Figure 1 where we want to check that the invariant $\neg(x = 2)$ holds. We will describe how the Underapproximation-Widening algorithm of Figure 5 proceeds for the case $k = 3$.*

Initially P contains negative unit clauses for each predicate $p_{i,j}$ for $i \in \{A, B, C\}$ and $j \in \{P_{a0}, P_{a1}, P_{b0}, P_{b1}, P_{c0}, P_{c1}\}$, therefore constraining all predicates to be false.

As a result, at every state, only the transitions of the enabled process with the smallest index are expanded (where we assume $A < B < C$). For example, only transition $\boldsymbol{a}$ is expanded from the initial state. The control-flow graph in the middle of Figure 2 shows the transitions taken at each of the reachable states.

This iteration did not reveal any counterexamples. The procedure will then analyze the unsatisfiable core for the generated formula and determine if any of the negative unit clauses we introduced was used by the proof of unsatisfiability. Let us assume that the clause $\neg p_{A,P_{a0}}$ was used by the proof, and therefore remove it from P and continue to the next iteration. This means that $p_{A,P_{a0}}$ may now be either TRUE or FALSE, while all other predicates must be FALSE.

The control-flow graph at the bottom of Figure 2 shows which transitions are taken in the next iteration. This is a superset of the ones taken previously. For example, in the initial state, transition $\boldsymbol{a}$ is expanded when $p_{A,P_{a0}}$ is FALSE, while transition $\boldsymbol{b}$ is expanded

when $p_{A,P_{a0}}$ is TRUE, since, in the latter case, B is the process with the smallest index satisfying the conditions for partial expansion.

In the second iteration a counterexample is found: if transition $\boldsymbol{b}$ is executed before transition $\boldsymbol{a}$, the final value of x will be 2, which is a violation of the invariant we were checking. At this point a counterexample is generated and the procedure terminates without having to ever consider all interleavings in the original model.

5. EXPERIMENTAL RESULTS

In this section, we compare the Underapproximation-Widening loop against Bounded Model Checking with full interleaving. Given a model and a property, Bounded Model Checking produces a CNF formula which is satisfiable if and only if there is a violation of the property in the given model. The proposed Underapproximation-Widening loop is also based on Bounded Model Checking, but instead of verifying the model directly as classical Bounded Model Checking would do, it verifies a series of underapproximated models, increasingly including more behavior.

We have implemented UW using NuSMV and the SAT solver zChaff. We use NuSMV to generate a CNF for the model M, which is parameterized on the values of the predicates introduced in Section 4.3. We modify the generated CNF by adding the set of negative unit clauses in P. We use zChaff ability to produce an unsatisfiable core to decide either termination or which clause to remove from the set P.

This approach, even if it might require an additional number of iterations, can be advantageous in the following two cases:

- A counter-example is found that it is present in an heavily underapproximated model. In this case, the SAT solver is given a much simpler system and it is possible to find a satisfying assignment more quickly.

- No counter-example is found, i.e., the formula is unsatisfiable, and, even if we analyzed a restricted model, the proof of unsatisfiability does not depend on the underapproximation. Again, even if this may occur after a few iterations, the reduced model verified at each iteration can be much simpler than the original model, and therefore require less time and resources.

Both of these cases appear in the examples presented below.

Table 1 contains the results obtained in the verification of a reachability property for a model of a leader election protocol. We can identify three different behaviors:

- **BMC is faster for small values of k**

  For small values of k (5 or less), the UW loop additional overhead is too high to compete with BMC. The formulas generated for small values of k are simple enough for the SAT solver to prove their unsatisfiability directly.

- **UW can prove the property using a much simpler model**

  For intermediate values of k (between 6 and 8), the UW loop is able to prove the property removing the unit clauses corresponding to only a few control points (at most 12 out of 30) and therefore it performs much better than BMC. Even if the number of iteration increases, the number of iteration is equal to the number of clauses that have been removed from P, the models with a limited set of interleavings are much simpler problems to solve.

- **UW can produce a counterexample very quickly**

Table 1: Comparison of the running time of BMC and UW on a reachability property of a leader election protocol. The first column represents the limit k; the second column reports the result of the verification; the last column reports the number of iterations needed to prove or disprove the property.

k	Result	BMC	UW	iter
0	valid	1.63s	1.99s	0
1	valid	1.65s	3.53s	1
2	valid	1.84s	4.63s	1
3	valid	2.28s	7.50s	2
4	valid	5.26s	13.75s	3
5	valid	18.87s	23.58s	4
6	valid	103.27s	55.10s	6
7	valid	191.69s	120.54s	12
8	valid	433.84s	144.83s	10
9	invalid	57.79s	11.10s	0
10	invalid	151.05s	11.07s	0
11	invalid	40.48s	21.27s	0
12	invalid	41.84s	22.18s	0

For k greater or equal to 9, the property is violated: UW is able to find a counterexample without removing any of the clauses from P. The CNF generated by UW is much simpler than the one generated by BMC, and the SAT solver is able to find a counterexample much faster.

In conclusion, one of the main advantages of this technique is the ability to detect property violations without having to look at the full model, but using a series of underapproximations. It is also possible to find a restricted model that is sufficient to prove the property. However, if the number of iterations necessary to reach that result is too large or the restricted model is not simpler than the original one, the overhead involved in additional iterations might become too large to make the procedure advantageous.

6. CONCLUSIONS AND FUTURE WORK

We presented a new efficient procedure for the verification of multi-process systems based on an Underapproximation-Widening loop. While a lot of successful work on automating Abstraction-Refinement loops for model checking has recently been done, the presented approach is, to the best our knowledge, the first fully automated approach based on underapproximations and widening. The procedure is mainly effective in discovering bugs due to the nature of BMC.

The UW approach is not limited to the verification of multi-process systems. The context of multi-process system verification has been chosen as it is straightforward to introduce 'clauses' that limit the state space.

Since our algorithm relies on the ability of SAT solvers to generate proofs of unsatisfiability, it is incorporated in a Bounded Model Checking framework, which, in its classical form, is incomplete. As future work, we plan to extend this framework to produce a complete BMC procedure, possibly based on existing work such as McMillan's work on interpolation and SAT-based model checking [23], the work of Sheeran et al. on k-induction [27], and the work of Kroening et al. on finding the *Completeness Threshold* [19, 12].

The procedure, as presented in the context of Bounded Model Checking, tries to prove or disprove the property for traces of a given length k, and then proceeds to analyze longer traces. Bounded Model Checking is therefore very effective in detecting shallow

counterexamples. However, since the UW procedure analyzes different models of increasing complexity for the same length k, it is possible to explore the generated models in a different order. We plan to investigate this idea.

Another topic for future research are heuristics to update the set of clauses in P to be used at the next iteration: in the presented work, when multiple clauses from the set P belong to the unsatisfiable core generated by the SAT solver, no indication is given on how to choose the clause to be removed. Moreover it is not strictly necessary to remove a single clause at each step, as long as the set of clauses removed from the initial value of P are sufficient to eliminate all the previous proofs. We intend to investigate such heuristics as well as alternative ways of updating the set P at each iteration.

There are similarities between the way underapproximations are built by the presented algorithm and the way partial-order reduction approaches construct a reduced model. Partial-order reduction approaches exploit static analysis as well as syntactic information to determine a set of control points at which it is sufficient to perform partial expansions instead of full expansions. These techniques will perform a partial expansion *only* when it is guaranteed that the validity of the property is preserved. While our approach is based on partial expansions as well, a main difference is that the reduced models that the proposed approach considers are not known a priori to preserve a property. However, partial order algorithms that determine partially and fully expanded control points statically [21], may be exploited to improve our procedure, e.g. its termination conditions. Exploiting partial order algorithms in the UW framework will be studied in future work.

7. REFERENCES

[1] R. Alur, R. Brayton, T. A. Henzinger, S. Quadeer, and S. Rajamani. Partial-order reduction in symbolic state space exploration. In O. Grumberg, editor, *Proc. of the 9th conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 340–351, Haifa, June 1997.

[2] N. Amla and K. McMillan. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *9th Intl. Conf. on Tools And Algorithms For The Construction And Analysis Of Systems (TACAS'03)*, volume 2619 of *Lect. Notes in Comp. Sci.*, 2003.

[3] F. Balarin and A. Sangiovanni-Vinventelli. An iterative approah to language containment. In C. Courcoubetis, editor, *Proc. 5th Intl. Conference on Computer Aided Verification (CAV'94)*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 29–40. Springer-Verlag, 1993.

[4] S. Barner, D. Geist, and A. Gringauze. Symbolic localization reduction with reconstruction layering and backtr acking. In *Proc. of Conference on Computer-Aided Verification (CAV)*, Copenhagen, Denmark, July 2002.

[5] S. Barner and O. Grumberg. Combining symmetry reduction and upper-approximation for symbolic model checking. In *14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002.

[6] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zue. *Bounded Model Checking*, volume 58 of *Advances in computers*. Academic Press, 2003.

[7] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, , and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC'99)*, 1999.

[8] A. Biere, A. Cimatti, E. M. Clarke, and Y. Yhu. Symbolic

model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[9] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In A. O'Leary, editor, *Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, lncs, Portland, Oregon, Nov 2002.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *J. ACM*, 50(5):752–794, 2003.

[11] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction - refinement using ILP and machine learning techniques. In E. Brinksma and K. Larsen, editors, *Proc. 14th Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 265–279, Copenhagen, Denmark, July 2002. Springer-Verlag.

[12] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *Proc. 5th Intl. Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lect. Notes in Comp. Sci.*, pages 85 – 96, Venice, Italy, Jan 2004.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proc. 12th Intl. Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2000.

[14] E. W. Dijkstra. *A discipline of programming*. Prentice Hall, 1976.

[15] E.M.Clarke, O.Grumberg, and D.Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[16] M. Glusman and S. Katz. A mechanized proof environment for the convenient computations proof method. *Formal Methods in System Design*, 23(2):115 – 142, 2003.

[17] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Workshop on Computer-Aided Verification (CAV'91)*, volume 575 of *LNCS*, pages 332–342, 1991.

[18] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107 – 120, September 1992.

[19] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *Proc. 4th Intl. Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309, NYU, New-York, January 2003. Springer Verlag.

[20] R. Kurshan. *Computer aided verification of coordinating processes*. Princeton University Press, 1994.

[21] R. Kurshan, V. Levin, M. Minea, and D. P. H. Yenigün. Combining software and hardware verification techniques. *Formal Methos in System Design*, 21(3):251–280, 2002.

[22] J. Lind-Nielsen and H. Andersan. Stepwise CTL model checking of state/event systems. In N. Halbwachs and D. Peled, editors, *Proc. 11th Intl. Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 316–327. Springer-Verlag, 1999.

[23] K. L. McMillan. Interpolation and sat-based model checking. In J. Warren A. Hunt and F. Somenzi, editors, *cav03*, Lect. Notes in Comp. Sci., Jul 2003.

[24] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design*, 8 (1):39–64, 1996. also appeared in 6th International Conference on Computer Aided Verification 1994, Stanford CA, USA, LNCS 818, Springer-Verlag, 377-390.

[25] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 154–158, November 1995.

[26] K. Ravi and F. Somenzi. Hints to accelerate symbolic traversal. In *CHARME'99*, LNCS 1703, pages 250–264. Springer-Verlag, 1999.

[27] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a sat-solver. In Hunt and Johnson, editors, *Proc. Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000.

[28] A. Valmari. A stubborn attack on state explosion. In *Workshop on Computer-Aided Verification (CAV'90)*, volume 531 of *LNCS*, New Brunswick, 1990.