SAT-based Verification without State Space Traversal

Per Bjesse and Koen Claessen

Department of Computing Science
Chalmers University of Technology
412 96 Göteborg
{bjesse, koen}@cs.chalmers.se

Abstract. Binary Decision Diagrams (BDDs) have dominated the area of symbolic model checking for the past decade. Recently, the use of satisfiability (SAT) solvers has emerged as an interesting complement to BDDs. SAT-based methods are capable of coping with some of the systems that BDDs are unable to handle.

The most challenging problem that has to be solved in order to adapt standard symbolic model checking to SAT-solvers is the boolean quantification necessary for traversing the state space. A possible approach to extending the applicability of SAT-based model checkers is therefore to reduce the amount of traversal.

In this paper, we investigate a BDD-based verification algorithm due to van Eijk. Van Eijk's algorithm tries to compute information that is sufficient to prove a given safety property directly. When this is not possible, the computed information can be used to reduce the amount of traversal needed by standard model checking algorithms. We convert van Eijk's algorithm to use a SAT-solver instead of BDDs. We also make a number of improvements to the original algorithm, such as combining it with recently developed variants of induction. The result is a collection of substantially strengthened and complete verification methods that do not require state space traversal.

1 Introduction

Symbolic model checking based on satisfiability (SAT) solvers [2, 1, 15, 12] has recently emerged as an interesting complement to model checking with Binary Decision Diagrams (BDDs) [3]. There are a number of systems which are not suited to be effectively verified using BDD-based model checkers, but can be verified using SAT-based methods. The use of SAT-solvers rather than BDDs also has advantages such as freeing the user from providing good variable orderings, and making the number of variables in the system less of a bottleneck. However, the boolean quantification that is necessary for computing characterisations for sets of predecessors (and successors) of states can sometimes lead to excessively large formulas in SAT adaptions of standard model checking algorithms.

In the hope of alleviating these problems, we investigate a BDD-based algorithm due to van Eijk [6] that attempts to verify safety properties of circuits without performing state-space traversal. The main idea behind the algorithm is to use induction to cheaply compute points in the circuit that always have the same value (or always have opposite values) in the reachable state space. This information sometimes directly implies the safety properties. If such a direct proof is not possible, the computed information can be used to decrease the number of necessary fixpoint iterations in backwards reachability algorithms. Van Eijk [6] has used the algorithm to directly prove equivalence between the original circuits and synthesised and optimised versions of 24 of the 26 circuits in the ISCAS'89 benchmark suite.

We are specifically interested in using van Eijk's algorithm to prove safety properties of circuits that are hard to represent using BDDs. Also, when a direct proof is not possible, we want to use the computed information to reduce the amount of state space traversal in exact SAT-based model checking methods as this could decrease the amount of necessary quantification drastically. As a consequence, we want to find alternatives to the use of BDDs in the original analysis. Van Eijk's algorithm also has the drawback of always computing the *largest* possible set of equivalences, even when this is not needed for the verification of the particular safety property at hand. In some cases this can become too costly; we would therefore like to be able to control how much work we put into finding equivalences.

We solve the two problems by converting the algorithm to use propositional formulas to represent points in the circuit, and by applying Stålmarck's saturation algorithm [14, 13] rather than BDDs for discovering equivalences between points.

The resulting algorithm is generalised in three ways. First, we make the algorithm complete by changing the induction scheme that is used in the method to some recently developed stronger variants of induction [12]. Second, we modify the algorithm to also discover implications between points in the circuit. Third, we demonstrate that van Eijk's algorithm can be viewed as an approximate forwards reachability analysis, and use this insight to construct the dual approximate backwards reachability algorithm and a mutual improvement algorithm.

The information that is computed by the resulting algorithms can in principle be used together with any BDD- or SAT-based model checking method. We show some benchmarks that demonstrate that the methods on their own can be very powerful tools for checking safety properties. For example, we use the algorithms to verify a non-trivial industrial example that previously has been out of reach for the SAT-based model checker FixIt [1].

2 Van Eijk's method: finding equivalence classes

In this section, we describe van Eijk's method [6]. In the original paper it is presented as a method for equivalence checking of sequential synchronous circuits.

However, while using the method we have observed that it can work well also for general safety property verification.

Basic idea. The idea behind van Eijk's algorithm is to find the points in the circuit which have the same value (or have opposite values) in all reachable states. This information can then be used to either directly prove the safety property or to strengthen other verification methods.

The information is represented as an equivalence relation over the points of the circuit and their negations. The algorithm computes such an equivalence relation by means of a fixed point iteration. It starts with the equivalence relation that necessarily holds between the points in the initial state. Then it improves the relation by assuming that the equivalences hold at one time instance and deriving the subset of these equivalences that must hold in the next time instance. After a number of consecutive improvements, a fixed point is reached. The resulting equivalence relation is satisfied by the initial states, and is moreover preserved by any circuit transition. Therefore, it must hold in all reachable states.

Before we give a more precise description of van Eijk's algorithm, we first introduce some definitions.

Formulas. We describe the systems we are dealing with using propositional logic formulas. These are syntactic objects, built from variables like x and y, boolean values 1 and 0, and connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$. We say that a formula is *valid* if and only if it evaluates to 1 for all variable assignments under the usual interpretation of the connectives.

State machines. We represent sequential synchronous circuits as state machines in the standard way [4], where the set of states is the set of boolean valuations of a vector s of variables; one variable for each input and internal latch. As we do not restrict the input part of the states, these state machines are non-deterministic. The standard representation also guarantees that every state has at least one outgoing transition.

We characterise the set of initial states and the transition relation of the state machine by the propositional logic formulas $\mathsf{Init}(s)$ and $\mathsf{Trans}(s, s')$, respectively. In other words, $\mathsf{Init}(s)$ is satisfied exactly by the initial states, and $\mathsf{Trans}(s, s')$ is satisfied precisely when there is a transition between the states s and s'. The safety property of the system we want to verify is represented by the formula $\mathsf{Prop}(s)$.

Example 1. Assume that we want to decide whether the two subcircuits in Figure 1 are equivalent. This amounts to checking whether the signal p is always true. Let us construct the necessary formulas. There are four state variables—one for every input and one for every delay component—so $s = (x, d_1, d_2, d_3)$. Since the delay components have an initial value of 0, the formula for the initial states becomes:

$$\mathsf{Init}(x, d_1, d_2, d_3) \quad = \quad \neg d_1 \wedge \neg d_2 \wedge \neg d_3$$

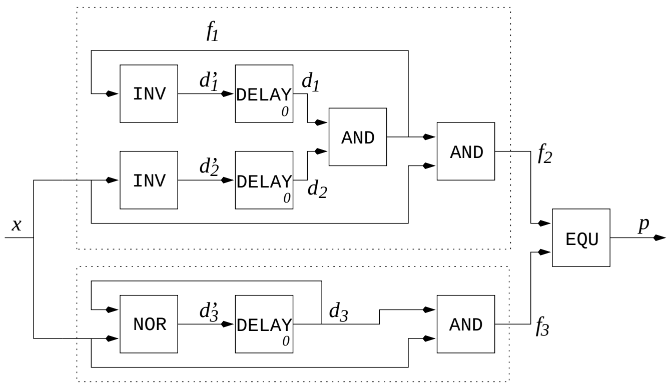

Fig. 1. An example circuit

Looking at the logic contained in the circuit, we can write down the formula for the transition relation:

$$\begin{aligned}\mathsf{Trans}(x, d_1, d_2, d_3, x', d_1', d_2', d_3') &= \\ (d_1' \Leftrightarrow \neg(d_1 \wedge d_2)) \;\wedge\; (d_2' &\Leftrightarrow \neg x) \;\wedge\; (d_3' \Leftrightarrow \neg(x \vee d_3))\end{aligned}$$

Lastly, we define the formula for the property p:

$$\mathsf{Prop}(x, d_1, d_2, d_3) \;=\; (d_1 \wedge d_2 \wedge x) \;\Leftrightarrow\; (d_3 \wedge x)$$

Signals. Given the formulas that characterise a state machine, we define the set $\mathsf{Signals}$ that models the points in the circuit. The elements of $\mathsf{Signals}$ are functions taking state variable vectors to formulas instantiated with these variables. Specifically, for every subformula $f(s)$ of the system formulas $\mathsf{Trans}(s, s')$ and $\mathsf{Prop}(s)$, such that $f(s)$ is not dependent on any of the variables of s', we add the corresponding functions f and $\neg f$ to the set. Moreover, we also add the constant signals tt and ff, for which $\mathsf{tt}(s) = 1$ and $\mathsf{ff}(s) = 0$.

Example 2. f_1 is a signal in Figure 1 with the definition $f_1(x, d_1, d_2, d_3) = d_1 \wedge d_2$. The negated signal $\neg f$ has the definition $\neg f_1(x, d_1, d_2, d_3) = \neg(d_1 \wedge d_2)$.

Signal correspondence. For a given equivalence relation $\equiv$ over the set $\mathsf{Signals}$, we define the *signal correspondence condition*, denoted by $\mathsf{Holds}(\equiv, s)$, as follows:

$$\mathsf{Holds}(\equiv, s) \;=\; \bigwedge_{f \equiv g} f(s) \Leftrightarrow g(s).$$

This means that the correspondence condition for an equivalence relation is satisfied by a state when all the signals that are equivalent have the same value in that state. We define a *signal correspondence relation* as an equivalence relation whose correspondence condition holds in all reachable states.[1]

Algorithm. In order to find a signal correspondence relation, van Eijk's algorithm computes a sequence of equivalence relations $\equiv_i$, each being a better overapproximation of the desired relation. The sequence stops when an n is found such that $\equiv_n$ is equal to $\equiv_{n+1}$.

The first approximation $\equiv_0$ is the equivalence relation that holds in the initial states. We can define it as follows; for all f and g, $f \equiv_0 g$ if and only if the following formula is valid:

$$\mathsf{Init}(s_1) \quad \Rightarrow \quad f(s_1) \Leftrightarrow g(s_1).$$

This means that two signals are equivalent precisely if they must have the same value in all initial states. The original algorithm computes $\equiv_0$ by constructing a BDD for every signal, and pairwise comparing these BDDs under the assumption that the BDD for $\mathsf{Init}(s)$ holds.

The other approximations $\equiv_{n+1}$ for $n \geq 0$ are subsets of $\equiv_n$. We can define them as follows; for all f and g, $f \equiv_{n+1} g$ if and only if $f \equiv_n g$ and the following formula is valid:

$$\mathsf{Holds}(\equiv_n, s_1) \wedge \mathsf{Trans}(s_1, s_2) \quad \Rightarrow \quad f(s_2) \Leftrightarrow g(s_2).$$

This means that two signals are equivalent in the new relation, when (1) they were equivalent in the old relation and (2) they have the same value in the next state if the old relation holds in the current state. The original algorithm computes $\equiv_{n+1}$ by pairwise comparison of the BDDs for the signals related by $\equiv_n$ under the assumption that the BDD for $\equiv_{n+1}$ holds.

The construction of approximations $\equiv_i$ has the shape of an *inductive* argument; it has a base case and a step that is iterated until it is provable. The final signal correspondence relation therefore holds in all reachable states.

For a schematic overview of the algorithm, see Figure 2. At lines 5 and 12, we use the function VALIDBDD that checks if a formula is valid by building its BDD. At lines 6 and 13, we use **set** to modify an equivalence relation by merging the equivalence classes for f and g.

Example 1 (ctd.). The signal correspondence relation found by the algorithm for Example 1 looks as follows:

$$\{\ldots, (f_1, d_3), (f_2, f_3), (p, \mathsf{tt}), \ldots\}$$

From this information it follows immediately that the property p is always true.

[1] Note that this is a slight generalisation of van Eijk's original definition [6].

```
$\equiv_1, \equiv_2$ := $\emptyset, \emptyset$ ;
-- compute first approximation
for every $f, g$ in Signals do
    form := $\mathsf{Init}(s_1) \Rightarrow (f(s_1) \Leftrightarrow g(s_1))$ ;
    if (validBdd(form)) then
        set $f \equiv_2 g$ ;
-- iterate until a fixed point is reached
while ($\equiv_1 \neq \equiv_2$) do
    $\equiv_1, \equiv_2$ := $\equiv_2, \emptyset$ ;
    for every $f, g$ in Signals such that $f \equiv_1 g$ do
        form := $\mathsf{Holds}(\equiv_1, s_1) \wedge \mathsf{Trans}(s_1, s_2) \Rightarrow (f(s_2) \Leftrightarrow g(s_2))$ ;
        if (validBdd(form)) then
            set $f \equiv_2 g$ ;
return $\equiv_1$ ;
```

Fig. 2. Van Eijk's algorithm

Remarks. The signal correspondence relation found by the algorithm sometimes implies the safety property directly. If this is not the case, then we can strengthen the transition formula $\mathsf{Trans}(s, s')$ to a new formula $\mathsf{Trans}(s, s') \wedge \mathsf{Holds}(\equiv, s) \wedge \mathsf{Holds}(\equiv, s')$. This is legal as we only are interested in transitions in the reachable state space. The new transition formula relates fewer states, and can consequently reduce the number of fixpoint iterations in conventional model checking methods.

Van Eijk's original paper presents a number of improvements of the basic method, such as *retiming* techniques that enlarge the set Signals so that the equivalence relation can contain more information, and *random simulation* that aims to reduce the number of pairwise comparisons by computing a better initial approximation $\equiv_0$. We will not discuss these techniques here, but refer to the original paper [6].

3 Stålmarck's method instead of BDDs

Van Eijk's method has a number of disadvantages. First of all, sometimes it is impossible to complete the analysis as some signals in the circuit can not be represented succinctly as BDDs. Second, the algorithm always finds the largest equivalence relation, which can be unnecessarily costly for proving the property. Third, the equivalences are computed by pairwise comparisons of signals, which means we have to build a quadratic number of BDDs. We will now focus on trying to solve these problems by using a SAT method instead of BDDs.

Stålmarck's method. Stålmarck's *saturation method* [14, 13] is a patented algorithm that is used for satisfiability checking. The method has been successfully applied in an wide range of industrial formal verification applications. The algorithm takes a set of formulas $\{p_1, \ldots, p_n\}$ as input, and produces an equivalence relation over the negated and unnegated subformulas of all p_i. Two subformulas are equivalent according to the resulting relation only when this is a logical consequence of assuming that all formulas p_i are true. The algorithm computes the relation by carefully propagating information according to the structure of the formulas.

The saturation algorithm is parameterised by a natural number k, the *saturation level*, which controls the complexity of the propagation procedure. The worst-case time complexity of the algorithm is $O(n^{2k+1})$ in the size n of the formulas, so that for a given k, the algorithm runs in polynomial time and space. For any specific k, there are formulas for which not all possible equivalences are found, but for every formula there is a k such that the algorithm finds all equivalences. A fortunate property is that this k is surprisingly low (usually 1 or 2) for many practical applications, even for extremely large formulas.

The advantage of having control over the saturation level is that the user can make a trade-off between the running time and the amount of information that is found. A disadvantage is that it is not always clear what k to choose in order to find enough information. In contrast, finding equivalences using BDDs results in discovering either all information, or no information at all due to excessive time and space usage.

Modification of van Eijk's method. We now adapt van Eijk's algorithm to use Stålmarck's method.

To compute the initial approximation $\equiv_0$, we use the saturation procedure to compute equivalence information between positive and negative subformulas of $\mathsf{Init}(s_1)$ and $\mathsf{Holds}(\mathsf{Id}, s_1)$ under the assumption that both of the these formulas are true. Here, Id denotes the identity equivalence relation on signals, relating f to g if and only if $f = g$. Note that $\mathsf{Holds}(\mathsf{Id}, s_1)$ is a valid formula, so assuming that it is true adds no real information; we just add it to the system to ensure that all subformulas that correspond to signals are present in the resulting equivalence relation. We then use the resulting information to generate the equivalence relation $\equiv_0$ on signals.

To improve a relation $\equiv_n$, we run the saturation procedure on a set of formulas that contains $\mathsf{Holds}(\equiv_1, s_1)$, $\mathsf{Trans}(s_1, s_2)$, and $\mathsf{Holds}(\mathsf{Id}, s_2)$. Again, we need the formula $\mathsf{Holds}(\mathsf{Id}, s_2)$ to ensure that all subformulas that correspond to signals are present. From the result we extract $\equiv_{n+1}$ by looking at the equivalences between subformulas depending on s_2, and taking the intersection with the original equivalence relation $\equiv_n$. The intersection of two equivalence relations relates two signals if both original relations relate them.

For a schematic overview of our algorithm, see Figure 3. The notation $\equiv /s_1$, occuring at lines 4 and 10, turns an equivalence relation $\equiv$ on formulas into an equivalence relation on signals, by relating two signals f and g if and only if their instantiated formulas $f(s_1)$ and $g(s_1)$ are related by $\equiv$.

In our modified algorithm, we have explicit control over the running time complexity of each iteration step; each step is guaranteed to take polynomial time and space. As a consequence, we do not have to worry about a possible exponential space blowup, as in the case of building BDDs. However, having this

```
$\equiv_1$     := $\emptyset$ ;
-- *compute first approximation*
system  := $\{\mathsf{Init}(s_1), \mathsf{Holds}(\mathsf{Id}, s_1)\}$ ;
$\equiv_2$     := STÅLMARCK(system) / $s_1$ ;
-- *iterate until a fixed point is reached*
**while** ($\equiv_1 \neq \equiv_2$) **do**
    $\equiv_1$     := $\equiv_2$ ;
    system  := $\{\mathsf{Holds}(\equiv_1, s_1), \mathsf{Trans}(s_1, s_2), \mathsf{Holds}(\mathsf{Id}, s_2)\}$ ;
    $\equiv$      := STÅLMARCK(system) ;
    $\equiv_2$     := $\equiv_1 \cap (\equiv / s_2)$ ;
**return** $\equiv_1$ ;
```

Fig. 3. Van Eijk's algorithm using Stålmarck's method

explicit control also means that we do not always compute the *largest* relation, since the saturation algorithm is possibly incomplete depending on what k we have chosen. In many cases it turns out that even for small k, the equivalence relation we compute using Stålmarck's algorithm is still large enough to decide if the property is true or not, or to considerably reduce the number of subsequent model checking iterations.

Signal implications. Finding equivalences between signals is a rather arbitrary choice. We could just as well try to find other information about signals that is easy to compute. For example, we can compute *implications* between signals.

An implication $f \Rightarrow g$ occurs between two signals f and g, if g must be true whenever f is true. The implications over the set of signals are interesting as they can capture *all* binary relations. The reason for this is that any formula that contains two variables can be characterised by a finite number of implications between these variables, their negations, and constants.

The presented algorithm can easily be extended to find implications between the computed equivalence classes of signals. Note that implications *within* equivalence classes do not give any information, since we know that every point in an equivalence class implies every other point in the same class. Our approach for generating the implications is simple: To begin with, we generate all the equivalence classes that hold over the reachable state space using the algorithm in Figure 3. From each equivalence class we take a representative signal. Finally, we run a modified version of the algorithm in Figure 3, that uses induction to find implications rather than equivalences between the representatives.

4 Induction

In order to further improve our adaptions of van Eijk's method, we start by investigating another safety property verification method: *induction* [12].

Simple induction. The idea behind simple induction is to attempt to decide whether all reachable states of the described system make the formula $\mathsf{Prop}(s)$ true by proving that the property holds at the initial states, and proving that if it holds in a certain state, it also holds in the next state. The inductive proof is expressible in propositional logic using the following two formulas:

$$\mathsf{Init}(s_1) \Rightarrow \mathsf{Prop}(s_1)$$
$$\mathsf{Prop}(s_1) \wedge \mathsf{Trans}(s_1, s_2) \Rightarrow \mathsf{Prop}(s_2)$$

If the first formula is valid, we know that all the initial states of the system make the property true. If the second formula is valid, we know that any time a state makes the property true, all states reachable in one step from that state also make the property true. We can thus infer that all reachable states are safe.

Simple induction is not a complete proof technique for safety properties; it is easy to construct a system whose reachable states all make $\mathsf{Prop}(s)$ true, but for which the inductive proof fails. Just take a safe system and change it by adding two unreachable states s_1 and s_2 in such a way that there is a transition between s_1 and s_2, the formula $\mathsf{Prop}(s_1)$ is true, and $\mathsf{Prop}(s_2)$ is false. This system can not give a provable induction step even though all reachable states satisfy the property. A stronger proof scheme is needed for completeness.

Induction with depth. In the step case of simple induction we prove that the property holds in the current state, assuming that it holds in the previous state. One way to strengthen the induction step is to instead assume that the property holds in the previous n consecutive states. Correspondingly, the base case becomes more demanding.

Let $\mathsf{Trans}^*(s_1, \ldots, s_n)$ be the formula that expresses that $s_1, \ldots, s_n$ are states on a path $s_1, \ldots, s_n$, and let $\mathsf{Prop}^*(s_1, \ldots, s_n)$ be the formula that expresses that Prop is true in all of the states $s_1, \ldots, s_n$. *Induction with depth* n amounts to proving that the following formulas are valid:

$$\mathsf{Init}(s_1) \wedge \mathsf{Trans}^*(s_1, \ldots, s_n) \Rightarrow \mathsf{Prop}^*(s_1, \ldots, s_n)$$
$$\mathsf{Prop}^*(s_1, \ldots, s_n) \wedge \mathsf{Trans}^*(s_1, \ldots, s_{n+1}) \Rightarrow \mathsf{Prop}(s_{n+1})$$

The modified base case expresses that all states on a path with n states starting in the initial states make the property true. The step says that if $s_1 \ldots s_{n+1}$ is a path where $s_1 \ldots s_n$ all make Prop true, then s_{n+1} also makes Prop true. We henceforth refer to n as the *induction depth*. Note that induction with depth 1 is just simple induction.

Unique states induction. The induction scheme with depth discovers paths to any state s in the reachable state space that makes $\mathsf{Prop}(s)$ false: A path with n states starting from the initial states and ending in a bad state is a counterexample to base cases of depth n or larger. As we are verifying finite systems, some depth n is therefore sufficient to discover all bugs.

Unfortunately the scheme is still not complete; it is possible to construct a safe system where the induction step fails for any depth n. Just take any safe system and change it by adding two unreachable states s_1 and s_2, so that the property holds in s_1, s_1 can both reach itself and s_2, and the property fails in s_2. Then there exist a counterexample for every depth n that loops $n-1$ times in s_1, and then visits s_2.

However, a state that is reachable from the initial states must be reachable by at least one path that only contains *unique* states. Therefore, we can add a formula $\mathsf{Uniq}(s_1, \ldots, s_n)$ to the induction step that expresses that $s_1, \ldots, s_n$ are different from each other. This restriction on the shape of considered paths makes it impossible to generate counterexamples of arbitrary length from loops in the unreachable state space. The induction step now becomes:

$$\mathsf{Prop}^*(s_1, \ldots, s_n) \wedge \mathsf{Trans}^*(s_1, \ldots, s_{n+1}) \wedge \mathsf{Uniq}(s_1, \ldots, s_{n+1}) \Rightarrow \mathsf{Prop}(s_{n+1})$$

The result is a complete scheme for verifying safety properties of finite systems: If there are paths in the unreachable state space that falsely make the induction step unprovable, they are ruled out from consideration by some induction depth n. However, a major problem is that this n can be extremely large for some verification problems, and that it is often difficult to predict what n is needed.

5 Stronger induction in van Eijk's method

We will now make use of the insight into induction methods we gained in the previous section. The underlying proof method that van Eijk's algorithm uses to find equivalences that always hold in the reachable state space is simple induction. Recall that we have demonstrated that this proof technique is too weak to prove all properties that hold globally in the reachable states. Consequently there are circuits that contain useful equivalences that van Eijk's original algorithm misses due to the incompleteness of its underlying proof method.

Generalisation to completeness. We can make van Eijk's original algorithm complete by modifying our implementation to use unique states induction with depth n rather than simple induction. In the base case of the algorithm, we compute an equivalence relation on signals that hold in the first n states on paths from the initial states. In the algorithm step, we assume that our most recently computed signal equivalence relation holds in the first n of $n+1$ consecutive unique states, and derive the subset of the signal equivalences that necessarily holds in state $n+1$.

For a detailed description of the resulting algorithm, see Figure 4. We use the notation $\mathsf{Holds}^*(\equiv, s_1, \ldots, s_n)$, occuring at lines 3 and 9, as a shorthand for $\mathsf{Holds}(\equiv, s_1) \wedge \cdots \wedge \mathsf{Holds}(\equiv, s_n)$. The algorithm for finding implications between signals is modified in an analogous way.

We can now discover all equivalences that hold globally in the state space. In particular, if a safety property holds in all reachable states, there exists a saturation level and an induction depth that is sufficient to discover that the corresponding signal is equivalent to the true signal.

```
$\equiv_1$ := $\emptyset$ ;
-- *compute first approximation*
system := $\{\mathsf{Init}(s_1), \mathsf{Trans}^*(s_1, \ldots, s_n), \mathsf{Holds}^*(\mathsf{Id}, s_1, \ldots, s_n)\}$ ;
$\equiv$ := STÅLMARCK(system) ;
$\equiv_2$ := $(\equiv/s_1) \cap \ldots \cap (\equiv/s_n)$ ;
-- *iterate until a fixed point is reached*
**while** $(\equiv_1 \neq \equiv_2)$ **do**
    $\equiv_1$ := $\equiv_2$ ;
    system := $\{\mathsf{Holds}^*(\equiv_1, s_1, \ldots, s_n), \mathsf{Trans}^*(s_1, \ldots, s_{n+1}),$
                  $\mathsf{Holds}(\mathsf{Id}, s_{n+1}), \mathsf{Uniq}(s_1, \ldots, s_{n+1})\}$ ;
    $\equiv$ := STÅLMARCK(system) ;
    $(\equiv_2)$ := $\equiv_1 \cap (\equiv/s_{n+1})$ ;
**return** $\equiv_1$ ;
```

Fig. 4. The adaption of the algorithm for depth n unique states induction

As an additional benefit, the possibility to adjust both the saturation level and the induction depth allows a high degree of control over how much work is spent on discovering equivalences. We can now increase the number of equivalences that can be discovered for a fixed saturation level by increasing the induction depth; this can be useful as an increase in saturation level means a big change in the time complexity of the algorithm.

We note that the idea of using stronger induction not is restricted to our SAT-based version of van Eijk's algorithm; the original BDD-based algorithm can also be made complete by stronger induction.

6 Approximations

In this section we show that van Eijk's algorithm is an *approximative forwards reachability* analysis. We then use this insight to derive an analogous backwards approximative analysis, and combine the two algorithms into a mutual improvement algorithm.

The forwards reachability view. Figure 5 shows the shape of a standard forwards reachability analysis, where we use INIT to denote the set of initial states, and the operation POSTIMAGE to compute postimages (the postimage of a set of states S is the set of states that can be reached from S in one transition). We now demonstrate that van Eijk's algorithm performs such a forwards analysis approximatively, in the sense that it is a variant of the standard analysis where INIT has been replaced with an overapproximation, and the exact operations $\cup$ and POSTIMAGE have been replaced by overapproximative operators.

Van Eijk's algorithm computes a sequence of relations $\equiv_i$. Each of the corresponding formulas $\mathsf{Holds}(\equiv_i, s)$ can be seen as the characterisation of a set of states A_i.

```
1. $n, S_0 \ := 0, \textsc{init}$ ;
2. **loop**
3.     $S_{n+1} \ := \textsc{postImage}(S_n) \cup S_n$ ;
4.     $n \ := n + 1$ ;
5. **until** $(S_{n+1} \neq S_n)$ ;
6. **return** $S_{n+1}$ ;
```

Fig. 5. A standard forwards reachability algorithm

In the base case, the algorithm computes the binary relation $\equiv_0$ that holds between points in all the initial states. The formula $\mathsf{Holds}(\equiv_0, s)$ will therefore be valid for every state s that makes $\mathsf{Init}(s)$ valid, and possibly for some other states. A_0 is consequently a superset of the initial states.

In the step, the algorithm computes the subrelation $\equiv_{n+1}$ of $\equiv_n$ that provably holds after a transition under the assumption that $\equiv_n$ holds before the transition. Every state s that is reachable in one transition from a state in A_n therefore makes $\mathsf{Holds}(\equiv_{n+1}, s)$ valid. But $\equiv_{n+1}$ is a subrelation of $\equiv_n$, so every state s in A_n also satisfies $\mathsf{Holds}(\equiv_{n+1}, s)$. Consequently, the step operation corresponds to computing A_{n+1} as the overapproximative union of A_n and the overapproximative postimage of A_n.

Finally, the algorithm checks whether $\equiv_{n+1}$ is the same relation as $\equiv_n$. This corresponds to checking whether $A_{n+1} = A_n$. If this is the case, the algorithm terminates, otherwise the step is iterated.

Approximative backward analysis. It is well known that forwards reachability analysis has a dual analysis called *backwards* reachability analysis [4]. We can perform the backward analysis using the forwards algorithm by modifying the characterisation of the underlying system in the following way:

$$\begin{aligned} \mathsf{Init}'(s) &= \neg\mathsf{Prop}(s) \\ \mathsf{Trans}'(s, s') &= \mathsf{Trans}(s', s) \\ \mathsf{Prop}'(s) &= \neg\mathsf{Init}(s) \end{aligned}$$

The result of the computation is the set of states that are backwards reachable from the bad states—the states where the property does not hold.

We can use the system transformation together with any of our variants of van Eijk's algorithm. In particular, we can compute a relation $\equiv$ that characterises an overapproximation of the states that are backwards reachable from the states that make $\mathsf{Prop}(s)$ false. Analogously to the forwards algorithm, the system is safe if $\mathsf{Holds}(\equiv, s)$ implies the safety property, which in this case corresponds to that no initial state is in the overapproximation of the set that can be backwards reached from the bad states. Also, if we do intersect the initial states, we can still use the approximation to constrain the transition relation in order to reduce the number of necessary iterations of an exact forwards reachability algorithm.

Mutual improvement. The new approximate backward algorithms can be very useful on their own. However, there exists a general way of enhancing approximative reachability analyses that improves matters further [8].

The idea is to first generate the overapproximation of the reachable states. If the corresponding set has an empty intersection with the bad states, we are done. If it has an nonempty intersection, there are two possible reasons: Either the system is unsafe, or the approximation is too coarse. Regardless of which is the case, we know that the only possible bad states we can reach are those that are contained in the intersection. We can therefore take the intersection to be our new bad states.

But now we can apply approximate backwards reachability analysis from the new bad states. If we do not intersect the initial states, the system must be safe. If we do, we can take the intersection to be the new initial states and restart the whole process. The algorithm terminates if we generate the same overapproximations twice, as this implies that no further improvement is possible.

The resulting mutually improved overapproximations are always at least as good as the original overapproximations, and they can sometimes be substantially better as we demonstrate in the next section.

7 Experimental results

In this section, we present a number of experiments we have done using a prototype implementation of our variants of van Eijk's algorithm. We compare our results against the results of three other methods. The first two are reachability analysis and unique states induction, as implemented in the SAT-based model checking workbench FixIt [1]. The third method is BDD-based model checking, as implemented in the verification tool VIS version 1.3. In the experiments with VIS we have used dynamic variable reordering and experimented with different partitionings.[2] All running times are measured on a 296 MHz Ultrasparc-II with 512 MB memory. The results are displayed in Table 1.

The motivation for the choice of benchmarks is as follows. We have chosen one industrial example, one example that is difficult to represent with BDDs, and one example that belongs to the easy category for BDDs.

The Lalita example. The Lalita example is an industrial telecommunications example from Lucent Technologies that was one of the motivations for the research presented in this paper. We received the example as a challenge from Prover Technology, a Swedish formal verification company. It was given to us as a black-box problem; we had no information about the structure of the system. The design was already known to be within reach of BDD technology, but not all of the properties were possible to verify using unique states induction.

[2] We have also tried approximate model checking in VIS, but there appears to be a bug in the implementation which makes it unsound.

Property	FixIt Reach.	FixIt Induct.	VIS	Our Method
Lalita, nr. 2	0.3	0.2	219.9	2.3[a]
7	41.8	0.2	207.3	2.2[a]
10	[>15min]	[>15min]	86.6	9.7[b]
11	[>15min]	[>15min]	199.3	2.1[a]
Butterfly, size 2	0.1	1.0	0.3	0.1[a]
4	16.6	[>15min]	2.0	0.1[a]
16	[>15min]	—	[>15min]	1.4[a]
64	—	—	—	37.4[a]
Arbiter	2.5	[>15min]	2.1	76.9[c]

[a] with equivalences, [b] with mutual improvement, [c] with implications

Table 1. Experimental results (times are in seconds).

When we attempted to verify the design using SAT-based reachability analysis, the representations became excessively large due to the computation of pre- and postimages.

The design contains 178 latches. The problem comes with thirteen safety properties that should be verified; we present the four most interesting properties: the two properties that were most difficult for VIS (2 and 7), one of the two properties that are hard for induction (11), and the property that was hardest for our methods (10).

All of the properties except property 10 and 11 can be done using SAT-based induction. However, we can verify or refute every property except property 10 directly using our forwards equivalence algorithm. Property 10 is verified using one iteration of mutual improvement of the computed equivalences. As the table demonstrates our analyses are a factor 10 to 100 faster than BDD-based verification in VIS.

The butterfly circuits. This family of benchmarks arose when we were designing sorting circuits for implementation on an FPGA. The problem is to decide whether a butterfly network containing reconfigurable sequential components is equivalent to an optimised version where the components have been shifted around. When we attempted to verify the circuits we discovered that standard algorithms could not handle circuits of any reasonable size. In particular, BDD-based methods did not work because the BDDs representing the circuits became too large.

The model checking algorithms in VIS are unable to verify larger networks than size 4 in a reasonable amount of time and space. SAT-based reachability analysis and induction are also unable to cope with larger instances of the circuits. However, the forwards equivalence algorithm handles all the sizes we have tried in less than 40 seconds.

The arbiter. This example is a simple benchmark from the VIS distribution. The arbiter controls three clients that compete for bus access. We verify the property of mutual exclusion.

The problem is easy both for BDDs and SAT-based symbolic reachability analysis, but can not be done using unique states induction. The example clearly demonstrates that finding implications between equivalence classes can be stronger than only computing equivalences: Our equivalence based analysis alone is unable to verify the design in a reasonable amount of time, but we can verify the design in 77 seconds by computing implications between the equivalence classes.

8 Related work

The first approach in the literature to apply SAT-based techniques to model checking was Bounded Model Checking [2]. Bounded model checking of safety properties corresponds to searching for bugs by attempting to prove the base case only of induction with depth. Certain bugs that are hard to find using BDD-based model checking can be found very quickly in this way. In order to effectively also prove safety of systems, standard symbolic reachability analysis was adapted to use SAT-solvers [1] which resulted in the analysis implemented in FixIt. Currently, interesting work is being done on combinations of SAT-solvers and BDDs for model checking [15].

The idea to use approximate analyses to generate semantic information from systems originally comes from the field of program analysis. Many different such analyses can be seen as abstract interpretations [5]. In particular, Halbwachs et al. [8, 10] have used abstract interpretation techniques to generate linear constraints between arithmetic variables that always holds in the reachable state space of synchronous programs and timed automata. This information is used both for compilation purposes and for verification. The same techniques are used for generating strengthenings in the STeP system [11] that is targeted towards deductive verification of reactive programs.

The main differences between our work and the work on synchronous programs and STeP, is (1) that the analyses we present here are specially designed for generating information about *gate level circuits* rather than programs, (2) that we focus specifically on simple relations between boolean signals, and (3) that we use Stålmarck's saturation method as a possibly incomplete but fast method for generating the relations. Also, we generate information while keeping in mind that we can apply an exact analysis later, and we have consequently optimised the algorithms for quickly generating information. In the case of synchronous program verification and STeP, a precise analysis is not even possible in general as infinite state systems are addressed.

Dill and Govindaraju [7] have developed a method for performing BDD-based approximate symbolic model checking based on overlapping projections. Their idea is to alleviate BDD blow-up by representing an overapproximation of a set of states S as a vector of BDDs, where each individual BDD characterises the relation in S between some subset of the state variables. The conjunction of the BDDs represents an overapproximation of the underlying set. The main difference between our approach and theirs, is that we consider some particular relations between *all* pairs of signals, while they consider all relations between a number of subsets of state variables. Also, the user of Dill and Govindaraju's method must manually choose the subsets of state variables to build BDDs for, whereas our methods are fully automatic.

9 Conclusions and Future Work

We have taken an existing BDD-based verification method which finds equivalent points in a circuit, and adapted it to use Stålmarck's method instead of BDDs. Then, we strengthened the resulting algorithm by combining it with recently developed induction techniques. We also discussed how the algorithm can be improved by computing implications rather than simple equivalences between points. Lastly, we observed that the algorithm can be transformed into a mutual improvement approximative reachability analysis.

The resulting collection of new algorithms can be seen as SAT-based improvements of van Eijk's original algorithm, where we use stronger inductive methods. Viewed from this angle, we have made van Eijk's method complete and provided a more fine-grained tuning between the time used and the information found.

Viewing our work in a different way, we can say that we have improved an inductive method by using van Eijk's algorithm to find equivalences. In some cases, such as the butterfly examples (see Section 7), unique states induction needs an exponentially larger induction depth than our improved analyses.

We believe the proposed methods work well for several reasons. First of all, van Eijk's original idea of finding equivalences of points in the circuit makes it very hard for properties to "hide" deep down in the logic of a circuit. Comparing this with problems occurring with methods that only look at state variables (such as conventional model checking methods) or methods that only look at the outputs (such as inductive methods) clearly shows that this is a desirable thing to do.

Second, the use of Stålmarck's saturation algorithm forms a natural fit with van Eijk's original algorithm. The possibility of controlling the saturation level pays off especially in systems where it is hard to find *all* equivalences, but sufficient to find some. Stålmarck's algorithm is also rather robust in the number of variables used in formulas.

Third, inductive methods perform well because they do not need any iteration or complicated quantification. Unfortunately, when we prove partial properties of systems, or when we prove properties about systems with a lot of logic between the latches and the property, induction performs poorly because the induction hypothesis is not strong enough to establish the inductive step. In this case, finding equivalence or implication information is just the right thing to do, because it strengthens the induction hypothesis, and provides direct information not only about the latches, but about all points in the circuit.

For future work, we would like to investigate other signal relations than equivalences and implications. General relations over three variables is a candidate, although it is not clear how to represent the found information. Furthermore, we are interested in extending the proposed algorithms to work with other properties than just safety properties. Lastly, we would like to extend the presented ideas to the verification of safety properties of synchronous reactive systems; for example, systems implemented in the programming language Lustre [9]. In order to do this, we need to add support for integer arithmetic and to investigate how Halbwachs's ideas [8] can be combined with our analyses.

Acknowledgements. Many thanks to Niklas Sörensson who implemented a large part of the algorithms and benchmarks, as well as to Niklas Eén, who helped with running some of the benchmarks. We also thank Byron Cook, Koen van Eijk, Carl-Johan Lillieroth, Gordon Pace, Mary Sheeran, and Satnam Singh for their useful comments on earlier drafts of this paper.

References

1. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proc. TACAS '00, 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2000.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS '99, 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.
3. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.
4. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
6. C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. Conf. on Design, Automation and Test in Europe*, 1998.
7. S. G. Govindaraju and D. L. Dill. Approximate symbolic model checking using overlapping projections. In *Electronic Notes in Theoretical Computer Science*, July 1999. Trento, Italy.
8. N. Halbwachs. About synchronous programming and abstract interpretation. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, Namur, Belgium, September 1994. LNCS 864, Springer Verlag.
9. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
10. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

11. Z. Manna and the STeP group. STeP: The Stanford Temporal Prover. Technical report, Computer Science Department, Stanford University, July 1994.
12. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer Aided Design*, 2000.
13. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's method of propositional proof. *Formal Methods In System Design*, 16(1), 2000.
14. G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. *Swedish Patent No. 467076 (1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995)* , 1989.
15. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc.* 12th *Int. Conf. on Computer Aided Verification*, 2000.