Experimenting with Predicate Abstraction

Victor Cacciari Miraldo, Maria João Frade, Cláudio Lourenço, and Jorge Sousa Pinto

HASLab/INESC TEC & Universidade do Minho, Portugal

Abstract. *Predicate abstraction* is a technique employed in software model checking to produce abstract models that can be conservatively checked for property violations in reasonable time. The precision degree of different abstractions of the same program may differ based on (i) the set of predicates used; or (ii) the algorithmic technique employed to generate the model. In this paper we explain how we have implemented and optimized one such technique, that produces the most precise existential abstraction of a program, and give the first steps towards establishing a common framework for both this direct technique and a second one, based on cartesian abstraction by weakest precondition calculations.

1 Introduction

Model Checking [7] has been successful in validating hardware system designs, to a point where its use has become not only common, but essential. It has long been hoped that this success will carry over to the realm of software, but this has proved exceptionally difficult, due to an aggravated state-space explosion problem. This is of course a typical problem of model checking, but the particular characteristics of software systems (like the presence of datatypes) make it particularly hard to handle.

Two main families of techniques have been employed in the last 10 to 15 years to make software model checking useful in practice. The first is *bounded model checking* [5], which in fact is not a specific technique for software applications. In a nutshell, it is employed as an alternative to BDD-based symbolic model checking that limits the exploration of executions of a system to a given bound on their length: paths longer than this bound are simply not explored. When applied to software, this amounts to limiting the number of loop iterations and recursive calls considered. Bounded model-checking problems can be encoded as logical *satisfiability* problems, solved with the help of a satisfiability solver. The drawback of this technique is that in general when a safety property is valid in the bounded model (e.g. a given error state is unreachable), it cannot be guaranteed to be valid in the original program.

The alternative is to consider an *abstract model* of the program, which may dramatically reduce the state-space. The goal of abstraction is to compute an abstract model $\bar{M}$ from the concrete model M of a program, such that the size of the state-space is reduced, but in a way that it is still sound to check for safety properties. Abstraction [10] is of course the fundamental technique that makes static program analyses feasible; the kind of abstraction used for software model checking safety properties is known as *existential abstraction* [8]. Informally, existential abstraction produces an abstract model $\bar{M}$ from a contrete model M of a program so that any reachability problem that is solvable in M is also solvable in $\bar{M}$, which means that if the abstract model satisfies a given safety property, then so does the original program (i.e. concretization preserves safety).

This paper is about a specific existential abstraction technique known as *Predicate Abstraction* [13], which has the advantage over other abstraction methods that it can be computed algorithmically. Predicate abstraction keeps track of a given set E of predicates over the data, and registers how the truth value of these predicates changes with the concrete program steps. Computing a predicate abstraction of a program inherently takes exponential time on its length; but predicate abstractions are not unique, and there exist different approaches to computing them, with the usual trade-off between precision and efficiency.

In any working software model checker predicate abstraction is implemented as part of a refinement loop. In *Counter Example-Guided Abstract Refinement* (CEGAR), the refinement that is performed at each iteration consists of adding more predicates to E. The loop successively calculates predicate abstractions, starting from a rougher model, and refines them by generating new predicates, based on the false positive counter-examples returned by the model checker at each stage, until no counter-example is returned (in which case the program is safe) or a true counter-example is found (the program is unsafe, there exists a real property violation). Abstract refinement is not covered in the present paper: we focus on predicate abstraction using a fixed set of predicates, which is the fundamental (and costly!) building block of any software model checker.

Although the literature on software model checking accumulated over these last 15 years is vast, we found that no clear and uniform presentation of the different approaches to predicate abstraction can be found. In this paper we take two such approaches, each of which has been proposed as part of a major software model checker. We remark that the precision degree of two predicate abstractions of the same program may differ based on (i) the set E of predicates provided (using more predicates results in a more precise model); or (ii) the algorithmic technique used to generate the model from E. Given a set E, the two methods considered here differ in the degree of precision of the generated abstract models. The first method is the one found in the MAGIC software model checker developed at CMU [6]; it is a straightforward technique that focuses on directly computing minimal existential abstractions by solving SAT problems. The second method (more efficient but less precise) is based on *cartesian abstraction*; it can be found at the core of the Microsoft SLAM tool [2].

The two methods are unfortunately described in the literature in terms that make them hard to compare or to implement based on the same backbone: whereas the direct method works at the level of transition systems, the second method produces a *Boolean program* (so different model checking tools have to be used on the abstract models produced). In addition, cartesian abstraction is implemented using *weakest precondition* computations, whereas the direct method is based on satisfiability checks. The main goal of the paper is to take steps towards a formulation of both methods in the same framework, for the double purpose of reasoning and implementation. Section 2 introduces the basic concepts of predicate abstraction; Sections 3 and 4 then present the specific details of each of the two methods we consider. Section 5 explains how we have implemented and optimized the direct satisfiability method to produce Boolean programs. Our first steps towards a formulation of the cartesian abstraction method by using SAT checks are also described. Section 6 concludes the paper.

2 Predicate Abstraction

We start by defining formally the notions of model and existential abstraction. A *model* M is defined by a triple (S, S_0, T), where S is the set of *states*, $S_0 \subseteq S$ is the set of *initial states*, and $T \subseteq S \times S$ is the *transition relation.* In what follows we will require *abstraction* functions mapping states of a model into states of another model, which we will extend to sets of states as expected. A *concretization function* $\gamma : \bar{S} \to S$ mapping abstract states into some concrete states is associated to each abstraction function.

Definition 1. *A model* $\bar{M} = (\bar{S}, \bar{S_0}, \bar{T})$ *is an* existential abstraction *of another model* $M = (S, S_0, T)$ *w.r.t. an* abstraction function $\alpha : S \to \bar{S}$ *if*

1. $\exists s \in S_0.\ \alpha(s) = \bar{s}\ \to\ \bar{s} \in \bar{S}_0$
2. $\exists (s, s') \in T.\ \alpha(s) = \bar{s} \wedge \alpha(s') = \bar{s}'\ \to\ (\bar{s}, \bar{s}') \in \bar{T}$

The minimal *existential abstraction also satisfies the converse implications.*

We will use one form of existential abstraction called *predicate abstraction*, where we abstract data by keeping track of predicates on it; every operation on the concrete model M will be translated to a Boolean operation on the abstract model $\bar{M}$. Predicate abstraction can be applied in the model checking of transition systems in general; when applied to software model checking, it produces a *Boolean program*, i.e. a program whose only data consists of a set of Boolean variables, with the same control-flow structure as the original program.

The idea is simple: given a predicate p_i on the variables of the original program, there will be a corresponding Boolean variable b_i in the abstract Boolean program; instructions in the original program will be abstracted into instructions on the Boolean variables, that reflect the effects of the original instructions on the truth values of the predicate. In particular, sequences of assignment instructions are mapped into parallel assignments. As a very simple example, the instruction `x := -x` would be abstracted with the predicates $p_1 \doteq x \leq 0$, and $p_2 \doteq x > 0$ as the following parallel assignment: `b1,b2 := b2,b1`.

Throughout the paper we will write E for the set of predicates used to construct the abstraction, and V for the set of Boolean variables in the Boolean program, with $\#V = \#E$. We will denote by $E(b)$ the predicate that is represented by the variable $b \in V$, and extend this notion to Boolean expressions in the natural way, for instance $E(b_1 \wedge b_2) = E(b_1) \wedge E(b_2)$.

To see how this matches our general discussion of abstraction, let us denote the set of Boolean values by $\mathbb{B} = \{T, F\}$. A concrete state consists of the program location $l \in \mathcal{L}$ and an assignment to its variables (for the sake of simplicity we consider that concrete programs manipulate only integer variables). Given a concrete state s, we will denote by $b_i(s)$ the logical value of the predicate $E(b_i)$ in s. The corresponding abstract state $\bar{s}$ consists of the program location $l \in \mathcal{L}$ and a valuation of the propositional variables in V, ie, $\bar{S} = \mathcal{L} \times \mathbb{B}^n$, where $n = \#E$. The abstraction function α will map a concrete state into an abstract one, and is defined by: $\alpha(s) = (loc(s), b_1(s), \cdots, b_n(s))$.

Suppose that the original program contains a command `assert A`, and one wants to model check the (safety) property that whenever the command is reached the Boolean expression A is true. If $A = p_i \in E$ for some i, then the command will be translated into `assert` b_i in the Boolean program, which can now be model-checked (if not, then a suitable expression constructed from the b_i must be used instead). The advantage of doing this is that the reachability problem for Boolean programs is decidable [1]. Dedicated model checkers for Boolean programs include BEPOP [1] and BOOM [4]. Note that predicate abstraction and Boolean model checking do not absolutely require working with Boolean programs as we do here. We could calculate the abstraction at the level of transition systems, and then model-check the abstract transition system using a general-purpose model checker. This will be further explained in Section 3.

As stated before, how to choose and refine a suitable set of predicates for a given program is outside the scope of this paper: we assume a fixed set E is provided, and consider two methods to construct an abstraction based on E. The methods differ only in the way that basic blocks of code (sequences of assignment instructions) are treated: control-flow is basically preserved from the concrete to the Boolean program. Also, the treatment of data structures like arrays, structures, and pointers, is orthogonal to the choice of abstraction method. As such, in what follows we will essentially consider basic blocks as concrete programs, consisting of sequences of integer assignment instructions.

3 The Direct Method

The most straightforward way to compute a predicate abstraction is to interpret Definition 1 at the level of programs and apply it directly with the help of a satisfiability solver. The method is described at length in [9]; it is used in practice in the MAGIC tool [6].

As an abstract state is given by the values of the propositional variables in V induced by the logical values of the predicates in E (in a given concrete state), one needs to test which combinations of logical values of the predicates before and after execution of the block are feasible. For this we need first of all to have a logical encoding of the block, which can be obtained by converting it to *static single assignment* (SSA) form [11]. Take for instance the basic block:

$$P \equiv \texttt{x := x + 10; y := y + 1}$$

It is converted to the following: $P \equiv$ `x1 := x0 + 10; y1 := y0 + 1`. The logical encoding L_P of P can now be written simply as a conjunction of equations, $L_P \equiv x_1 = x_0 + 10 \wedge y_1 = y_0 + 1$. This encoding can be applied to a much richer language, including arrays, structures, and pointers [9].

Note that each variable in the concrete program is now represented by a family (both of size 2, in the above example) of variables in its logical representation. Of these we are only interested in the initial and final versions of each variable; when considering the execution of a basic block we will in general denote by s and s' the program state expressed in terms respectively of the initial and final versions of the variables, therefore $b_i(s)$ and $b_i(s')$ will denote the initial and final values of the predicate $E(b_i)$. Consider again our example program and take for instance $V = \{b_1, b_2\}$ with $E(b_1) = x \geq 0$ and $E(b_2) = \mathit{even}(y)$. Then $b_1(s)$ is $x_0 \geq 0$, $b_2(s)$ is $\mathit{even}(y_0)$, $b_1(s')$ is $x_1 \geq 0$, and $b_2(s')$ is $\mathit{even}(y_1)$.

Let us now introduce some basic definitions and notation. A *literal* is a Boolean variable or its negation. Let $V = \{b_1, \ldots, b_n\}$ be a set of Boolean variables. A *cube* over V is a conjuntion of literals in which each of the variables of V appears exactly once. A *cover* is a disjuntion of cubes. We let $l, l_i, \ldots$ range over literals, and $c, c_i, c', \ldots$ range over cubes. We will denote by C_V the set $\{c_1, \cdots, c_{2^n}\}$ of all possible cubes over V. Note that each such cube uniquely corresponds to a valuation of the propositional variables in V, and thus to an abstract state. For instance the cube $b_1 \wedge \neg b_2$ corresponds to the abstract state in which b_1 is true and b_2 is false.

Following existential abstraction, to decide whether to include in the abstract model a transition from the state characterized by the cube c_i to the state characterized by the cube c_j, it suffices to check the satisfiability of the formula

$$E(c_i)(s) \wedge L_P \wedge E(c_j)(s')$$

Say, for the program and predicates given above, we wish to check the existence of a transition from the state in which both predicates are false to the state in which both are true. We check the satisfiability of

$$\neg(x_0 \geq 0) \wedge \neg(\mathit{even}(y_0)) \wedge x_1 = x_0 + 10 \wedge y_1 = y_0 + 1 \wedge x_1 \geq 0 \wedge \mathit{even}(y_1)$$

Indeed the formula is satisfiable, for instance with $x_0 = -2$ and $y_0 = 0$, and the transition will thus be included in the abstract model. The abstract transition system can be constructed by exhaustively testing 2^{2n} formulas:

$$\begin{gathered} E(c_1)(s) \wedge L_P \wedge E(c_1)(s') \\ \vdots \\ E(c_{2^n})(s) \wedge L_P \wedge E(c_{2^n})(s') \end{gathered}$$

The formulas can be checked by an SMT solver using a theory of (unbounded) integers, or, if one wishes to employ a fixed-size bitvector encoding of numbers (that stands closer to the machine representation), by a SAT solver after bit-blasting. Every satisfiable formula (corresponding to an abstract transition in the model) from the above family is recorded, allowing us to calculate an *assignment table*. In our example the solver would return the following table:

b_1	b_2	b_1'	b_2'
F	F	F	T
F	F	T	T
F	T	F	F
F	T	T	F
T	F	T	T
T	T	T	F

As described in [9], the method is used to produce an abstract transition system, which the authors then export to a general-purpose symbolic model checker to find property violations. But our interest is not in exporting the abstract model in the form of a transition relation; instead, we would like to produce a *Boolean program*. The reasons for this are twofold: first, specific model checkers for Boolean programs are of course fine-tuned for this problem, and thus handle it more efficiently. Second, other methods for generating predicate abstractions produce Boolean programs natively, and so do most existing software model checking tools; for the sake of uniformity (and to facilitate comparison) we also choose to follow the latter approach.

4 Cartesian Abstraction by WP Computations

SLAM [2], the tool that might be called the most successful software model checker (it has become a comercial product, currently shipped by Microsoft as part of the Windows Driver Development Kit), uses a different method for constructing predicate abstractions. It constructs less precise abstractions, and naturally does so more efficiently than the direct method.

Let again P be a basic block and L_P its logical encoding, E be the set of predicates used to construct the predicate abstraction, and V the set of Boolean variables. An alternative to using satisfiability tests is to employ weakest precondition (WP) calculations. Recall that the weakest precondition of a basic block with respect to a given assertion ψ is given by the following two rules:

$$wp(x := e, \psi) \doteq \psi[e/x] \qquad wp(C_1; C_2, \psi) \doteq wp(C_1, (wp(C_2, \psi)))$$

Recall the example program and predicates of the previous section. Then

$$wp(P, E(b_1)) \equiv x \geq 0[y + 1/y][x + 10/x] \equiv x + 10 \geq 0$$

One way to construct an abstraction is to determine individually, for each Boolean variable $b \in V$, the sets of states in which the weakest preconditions $wp(P, E(b))$ and $wp(P, E(\neg b))$, respectively, are satisfied. In the first set of states execution of the block will make $E(b)$ hold in the final state, thus the assignment $b := T$ should be executed by the Boolean program. In the second set of states $b := F$ should be executed, and in states in which neither $wp(P, E(b))$ nor $wp(P, E(\neg b))$ are satisfied, the assignment $b := *$, signaling a non-deterministic assignment, should be executed. It is useful to employ the following function:

$$choose(pos, neg) = pos\ ?\ T : (neg\ ?\ F : *)$$

The general idea is that the basic block can be abstracted by a parallel assignment of the form $\dots, b, \dots := \dots, \mathit{choose}(\mathit{wp}(P, E(b)), \mathit{wp}(P, E(\neg b))), \dots$. But this is of course not a valid Boolean program, since $\mathit{wp}(P, E(b))$ cannot be expressed in terms of the Boolean variables. What we can do in the Boolean program is to determine the combinations of values of the Boolean variables that force each of the above WPs to hold. This can be formalized as follows. Given an assertion ψ, let S_V^ψ denote the following disjunction of cubes over V:

$$S_V^\psi = \bigvee \{c \in C_V \mid \quad \models E(c) \to \psi\}$$

(V will be dropped when clear from context) Note that constructing this set requires 2^n validity tests, where $n = \#V$. Then P is abstracted by the following Boolean program, where ϕ_i denotes the assertion $\mathit{wp}(P, E(b_i))$:

$$b_1, \cdots, b_n := \mathit{choose}\left(S^{\phi_i}, S^{\neg\phi_i}\right), \cdots, \mathit{choose}\left(S^{\phi_n}, S^{\neg\phi_n}\right)$$

Observe that computing the abstraction in this way requires testing the validity of $2n \times 2^n$ formulas. This is still exponential, but also exponentially better than the direct method. This method introduces more false positives than the direct method because the different predicates are considered independently of each other, thus contradictory states are present in the models. This is in fact what is known as *cartesian abstraction.*

To understand this, consider that E consists of the two predicates $E(b_1) \equiv x \geq 0$ and $E(b_2) \equiv x \leq 100$. This produces an *unsatisfiable cube*: $E(\neg b_1 \wedge \neg b_2) \equiv x < 0 \wedge x > 100$, which is a contradiction, and would be included in S_V^ψ for any condition ψ: there exists a transition from the state corresponding to the unsatisfiable cube to any other state. Moreover, transitions *into* this state could also be present, since the WPs are computed independently for $E(b_1)$ and $E(b_2)$. Compare this to what would happen with the direct method: any satisfiability formula involving a contradictory state, of the form

$$E(\neg b_1 \wedge \neg b_2)(s) \wedge L_P \wedge E(c_j)(s') \quad \text{or} \quad E(c_i)(s) \wedge L_P \wedge E(\neg b_1 \wedge \neg b_2)(s')$$

is UNSAT, and rejected from the assignment table. Thus the corresponding transitions will not be inserted in the construction of the abstract model.

5 Implementation of Predicate Abstraction Algorithms

Implementing the Direct Method. The description of the direct method in Section 3 is just the first half of the story: we have indeed identified the valid transitions in the abstract model, but our goal is to produce a Boolean program. In this section we explain how we have implemented the direct algorithm so that it outputs a Boolean program.

Our goal is to abstract a basic block as a parallel assignment of Boolean variables of the form $b_1, \dots, b_n := e_1, \dots, e_n$. The task is then to find the right-hand side expressions $e_1, \dots, e_n$, given an assignment table. To this end the table is first divided into n tables, one for each variable in the final state. Each resulting table is then divided into its ON-set and OFF-set (that is, the assignments that turn each output variable to T and F, respectively). In our example this yields the two tables shown on the left below.

b_1	b_2	b'_1
F	F	T
F	T	T
T	F	T
T	T	T
F	F	F
F	T	F

b_1	b_2	b'_2
F	F	T
F	F	T
T	F	T
F	T	F
F	T	F
T	T	F

b_1	b_2	b'_1
T	F	T
T	T	T
F	F	*
F	T	*

b_1	b_2	b'_2
F	F	T
T	F	T
F	T	F
T	T	F

Note that the first table contains non-determinism: the same combination of values of b_1 and b_2 may result in different values for b'_1. The second table on the other hand contains redundancy (repeated entries than can be removed). We rewrite and simplify the tables as shown on the right. Note that the first table now has what one might call an UNDET-set rather than an OFF-set. The ON, OFF and UNDET-sets constitute a partition of the set of assigments according to the possible results of the output variable. Each of these sets is captured by a Boolean formula which is the disjunction of the cubes that characterize each assigment in the set. We call these formulas respectively ON, OFF and UNDET-covers. We let ON_i (resp. OFF_i) denote the ON-cover (resp. OFF-cover) for b'_i.

A parallel assignment can be directly extracted from these tables by using these covers: $b_1, b_2 := ((b_1 \wedge \neg b_2) \vee (b_1 \wedge b_2)?\ T : *), ((\neg b_1 \wedge \neg b_2) \vee (b_1 \wedge \neg b_2)?\ T : F)$, which can in turn be simplified to $b_1, b_2 := (b_1?\ T : *), (\neg b_2?\ T : F)$. If the UNDET-set is not empty a nested conditional expression will have to be used. In fact, although this has to our knowledge never been made explicit, the parallel assignment can be written as follows using the *choose* function of Section 4:

$$b_1, \cdots, b_n \ := \ \mathit{choose}(\mathsf{ON}_1, \mathsf{OFF}_1), \ \cdots \ , \ \mathit{choose}(\mathsf{ON}_n, \mathsf{OFF}_n)$$

It is clear from this small example that it would be infeasible to export a Boolean program without first attempting to simplify the assigned expressions; let us now describe how we have implemented this simplification.

Boolean simplification. The minimization of a Boolean function is a well-known problem in the area of logic circuit design: a circuit with a large number of logic gates (equivalent to a complex Boolean function) takes up a lot of physical space in its implementation. This problem is believed to be intractable [14], but there exist effective heuristics for it, such as Karnaugh Maps and the Quine-McCluskey algorithm. Our testbed is implemented using a functional programming language; for this reason we have opted for a recursive algorithm based on the *prime consensus theorem*, described in R. Rudell's thesis [15] (Sect. 2.5.1).

First, let us introduce some definitions and notation. In what follows a *cube* is simply a conjuntion of literals. Associativity, commutativity and idempotence of conjuntions and disjuntion allow us to treat each cube as a set of literals and each cover as a set of sets of literals.

Given two cubes, c, c', we say they *differ in a variable* x if $x \in c$ and $\neg x \in c'$ (or vice-versa). The *distance* between c and c', written $\mathsf{dist}(c, c')$ is the number of variables where they differ. When $\mathsf{dist}(c, c') = 0$ we say that c and c' *intersect* and the *intersecting cube* is $c \cup c'$.

The *consensus* of two non-intersecting cubes, c and c', $\mathsf{consensus}(c, c')$, is defined as follows: if $\mathsf{dist}(c, c') \geq 2$, their consensus is empty; if $\mathsf{dist}(c, c') = 1$, their consensus is $(c \cup c') - \{x, \neg x\}$, assuming c, c' differ in x. The notion of consensus is lifted to sets of cubes, as the pairwise consensus of the two sets.

Given two cubes, c, c', we say that c' is *single-cube contained* in c if $c \subseteq c'$. Given a set of cubes C, the *single-cube containment of* C is the set $\mathsf{SCC}(C) = \{c \mid \exists c, c' \in C.\ c \neq c' \wedge c \subseteq c'\}$. Let f be a Boolean function. A cube c is an *implicant* of f whenever $c \to f$. Moreover, we say that c is a *prime implicant* of f if c is minimal, i.e., there is no other implicant of c except itself. The set of prime implicants of f is denoted by $\mathsf{primes}(f)$.

We can now state the fundamental theorem that stands at the heart of the simplification algorithm we have implemented.

Theorem 1 (Prime consensus theorem). *Let f be a Boolean function and let x be any input variable. The set of prime implicants of f can be partitioned into three sets: $P_x = \{c \in \mathsf{primes}(f) \mid x \in c\}$, $P_{\neg x} = \{c \in \mathsf{primes}(f) \mid \neg x \in c\}$ and $P_* = \{c \in \mathsf{primes}(f) \mid x \notin c \wedge \neg x \notin c\}$. Then,*

$$\forall c \in P_*. \exists c \in P_x. \exists c' \in P_{\neg x}.\ c = \mathsf{consensus}(c, c')$$

This theorem states that $P_* \subseteq \mathsf{consensus}(P_x, P_{\neg x})$, because the consensus of $P_x, P_{\neg x}$ may contain non-prime implicants. We can get rid of such non-primes by constructing the single-cube containment of that set. We have

$$P_* = \mathsf{SCC}(\mathsf{consensus}(P_x, P_{\neg x}))$$

Now that we know how to generate P_* from P_x and $P_{\neg x}$, let us focus on the construction of P_x and $P_{\neg x}$ given a cover F of a Boolean function, and an input variable x.

A *cofactor* of F with respect to a literal l, written F_l, is defined as follows $F_l = \{c - \{l\} \mid c \in F \wedge l \in c\}$. In fact, $P_l \subseteq \{l\} \cup \mathsf{primes}(F_l)$. So, as before, we have to get rid of the non-primes.

The following theorem summarizes how the prime implicants of a Boolean function f can be generated recursively, and is effectively an algorithm outline.

Theorem 2 (Recursive prime generation theorem). *Let f be a Boolean function with (ON+UNDET)-cover F and let x be any input variable. Then, the prime implicants of f can be generated as follows:*

$$\mathsf{primes}(f) = \mathsf{SCC}(A_x \cup A_{\neg x} \cup \mathsf{consensus}(A_x, A_{\neg x}))\ ,\ \textit{where}\ A_l = \{l\} \cup \mathsf{primes}(F_l)$$

Note that in this divide and conquer approach, the choice of division point (the splitting variable x) will have major impact on the algorithm's efficiency. Clever rules for termination have been proposed that can speed up the process [15].

Handling variable initialization and optimizations. The predicate abstraction constructed by this method naturally eliminates transitions *from* and *to* states corresponding to *unsatisfiable cubes*, as shown at the end of Section 4.

We have introduced two modifications in the original algorithm, which we now describe. The first has to do with the fact that this method does not deal well with variable initialization in the presence of unsatisfiable cubes. To see this, let P be the basic block `x := 10`, with the two previous predicates. Clearly the block should be abstracted to the Boolean program `b1, b2 := T, T`. For this, the expected assignment table would be:

b_1	b_2	b'_1	b'_2
T	T	T	T
T	F	T	T
F	T	T	T
F	F	T	T

Observe that the last row will *not* be in the table, since the following is not satisfiable (x_0 cannot be smaller than 0 and greater than 100 at the same time):

$$(x_1 = 10) \wedge \neg(x_0 \geq 0) \wedge \neg(x_0 \leq 100) \wedge (x_1 \geq 0) \wedge (x_1 \leq 100)$$

Our guess is that tools based on the direct method calculate predicate abstractions after running a *constant propagation* transformation. We propose a modification of the algorithm that does not require this transformation.

The second modification is an optimization: we initially run a battery of satisfiability checks of formulas combining the program and the post-state cubes. Admittedly this takes time 2^n, but observe that for each unsatisfiable cube found we save 2^n checks, one for each pre-state cube. Moreover, this initial round of checks also eliminates 2^n checks for every post-state corresponding to a cube that, although satisfiable, can never be attained by the program (such as $b_1 \wedge \neg b_2 \equiv x > 100$ in the example). This is trivially correct, since we are only eliminating from the assignment table (by factoring) unsatisfiable rows.

Abstraction algorithm. We have introduced simple modifications on the algorithm described in [9], which are able to prevent the erroneous abstractions produced by inconsistent states as described previously. Moreover, the resulting algorithm seems to dramatically reduce the number of solver calls.

Definition 2 (Dependent variable). *Let P be a basic block, we say that a given variable $x \in Vars(P)$ is* dependent *if the initial value of x in the pre-state is used in P (i.e. x is read before it is written). If P is a basic block in SSA form, x is dependent if x_0 occurs in P.*

Let P be a basic block, E a set of predicates, C_V the set of all possible cubes of E and L_P the logic encoding of P. Our algorithm computes the assignment table of P, by calculating for each possible satisfiable and attainable post-state, which pre-states can lead to it (note that for the pre-state we instantiate only the predicates where dependent variables occur). The pseudo-code is presented as Algorithm 1, where some auxiliary functions are used. *addAllCombinationsFor*(*pos*)

Algorithm 1 Abstraction of Basic Blocks

```
for $pos_c \in cubesOf(E)$ do
    $pos_f \leftarrow L_p \wedge instantiate(poststate, pos_c)$
    if $solve(pos_f) = SAT$ then
        $preds \leftarrow \{p \in E : varsOf(p) \cap dependentVariables(L_p) \neq \emptyset\}$
        if $preds = \emptyset$ then $addAllCombinationsFor(pos_c)$
        else
            for $pre_c \in cubesOf(preds)$ do
                $full_f \leftarrow pos_f \wedge instantiate(prestate, pre_c)$
                if $solve(full_f) = SAT$ then
                    $addLines(pos_c, interpolate(independentVariables(L_p), pre_c))$
                end if
            end for
        end if
    end if
end for
```

appends every possible cube with *pos* and appends the result to the assignment table. $interpolate(vars, cube)$ completes the cube by combining it with every possible combination of $vars$, in the correct positions. *addLines* simply adds rows to the assignment table. An example run is presented in the appendix.

Implementing Cartesian Abstraction. At the time of writing we have implemented cartesian abstraction by weakest precondition calculations in a straightforward way, following Section 4. Our implementation still produces unsimplified parallel Boolean assignments – it is not straightforward to apply here the simplification techniques implemented for the direct method, since the expressions to be simplified are not assignment table covers. An example of the unsimplified output can be found in the appendix.

We are presently investigating how cartesian abstraction can be implemented based on satisfiability checks, rather than WP computations and validity checks. Recall that the basis of the algorithm is the computation:

$$S^{wp(P,E(b_i))} = \bigvee \{c \in C_V \mid \quad \models E(c) \rightarrow wp(P, E(b_i))\}$$

A first observation to make is that in the SSA setting weakest preconditions can be computed without substitution, based on the same logical encoding of a program used in the direct method [12]. The above can be written instead as

$$\begin{aligned} S^{wp(P,E(b_i))} &= \bigvee \{c \in C_V \mid \quad \models E(c)(s) \rightarrow L_P \rightarrow E(b_i)(s')\} \\ &= \bigvee \{c \in C_V \mid \quad \text{UNSAT}\,(E(c)(s) \wedge L_P \wedge \neg E(b_i)(s'))\} \end{aligned}$$

which provides a basis for the relation we are seeking to establish. Indeed, the above satisfiability problems are very close to the formulas $E(c_i)(s) \wedge L_P \wedge E(c_j)(s')$ checked for satisfiability in the direct method. We are presently investigating this relation with the aim of producing a common backbone for both implementations, using the same Boolean simplification functionality.

6 Conclusion

In this paper we have discussed the calculation of predicate abstractions for basic blocks of code, and how, based on the prime consensus theorem, we have adapted the direct abstraction technique to produce Boolean programs, introducing modifications for correctly handling variable initialization, as well as optimizations that substantially reduce the number of necessary calls to the solver. Note that we are not claiming to have produced an algorithm that performs better than working tools based on the direct method, because these tools also incorporate many other optimizations (not fully documented). Furthermore, our algorithm needs benchmarking since it has only been tested with small programs.

With respect to cartesian abstraction, the use of simplified rules for calculating WPs of SSA code, together with our implementation of the direct method based on assignment tables, clearly point to the existence of a common framework in which both methods can be expressed, which we believe has never been identified. This will allow us to apply our simplification module to cartesian abstraction, as well as to establish a formal relation between both methods.

Although not described in detail here, our implementation also handles control flow constructs (branching and looping) in addition to basic blocks. Consider for instance the code shown below at the left:

```
P;                          AP;
while (c1) loop             while (b1) loop
  Q;                          AQ;
  if (c2) then                if (b2) then
    R;                          AR;
  end if;                     end if;
end loop;                   end loop;
S;                          AS;
```

Under the assumption that the Boolean conditions c_1 and c_2 belong to the set E of predicates used to construct the abstraction, it is straightforward to produce the abstraction of this fragment: it suffices to preserve the control flow structure in the resulting Boolean program, replacing the conditions by the corresponding Boolean variables. Let $E(b_1) = c_1$ and $E(b_2) = c_2$, and `AP`, `AQ`, `AR`, `AS`, be the abstractions of the basic blocks `P`, `Q`, `R`, and `S` respectively. We obtain the Boolean program shown on the right. In the more general case, when the conditions contain subformulas that are not part of E, some heuristic must be used to decide how to simulate the behavior of the concrete program. Different software model checkers propose different solutions, that we leave outside our discussion. In practice this is rarely required, since the controw-flow conditions are often chosen as predicates of E, or added by a refinement algorithm when a spurious counterexample passing through this location is found.

Our current prototype handles a subset of the SPARK Ada programming language [3], widely used in the development of safety-critical software, and is part of a larger effort, which also includes the development of a bounded model checker for SPARK programs. It handles simple blocks by the two methods described in Section 5, and control-flow as described above. We are currently focusing on a satisfiability-based formulation of cartesian abstraction, to improve our WP-based implementation. As future work, our priority is to implement (i) a laboratory for experimenting with different techniques and optimizations over a common backbone; and (ii) a refinement loop combining abstraction methods, including transition refinement and counterexample validation.

References

1. Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Procs. of the 7th International SPIN Workshop (SPIN'00)*, pages 113–130, London, UK, 2000. Springer-Verlag.
2. Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.
3. John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
4. Gerard Basler, Matthew Hague, Daniel Kroening, C.-H. Luke Ong, Thomas Wahl, and Haoxian Zhao. Boom: taking boolean program model checking one step further. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 145–149, Berlin, Heidelberg, 2010. Springer-Verlag.
5. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.
6. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. *IEEE Trans. Softw. Eng.*, 30(6):388–402, June 2004.
7. Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52:74–84, November 2009.
8. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Procs. of POPL'92*, pages 343–354, New York, USA, 1992. ACM.
9. Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using sat. *Formal Methods in System Design*, 25(2-3):105–127, 2004.
10. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
11. Ron Cytron, Jeanne Ferrante, BK Rosen, Mark N Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
12. Daniela da Cruz, Maria João Frade, and Jorge Sousa Pinto. Verification conditions for single-assignment programs. In *Proceedings of the 27th ACM Symposium On Applied Computing (SAC'12)*, pages 1264–1270. ACM, 2012.
13. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.
14. Valentine Kabanets and Jin-Yi Cai. Circuit minimization problem. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 73–79, New York, NY, USA, 2000. ACM.
15. Richard L. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, EECS Department, University of California, Berkeley, 1989.