Boolean and Cartesian Abstraction for Model Checking C Programs

Thomas Ball   Andreas Podelski*   Sriram K. Rajamani

Software Productivity Tools
Microsoft Research

Abstract. We show how to attack the problem of model checking a C program with recursive procedures using an abstraction that we formally define as the composition of the Boolean and the Cartesian abstractions. It is implemented through a source-to-source transformation into a 'Boolean' C program; we give an algorithm to compute the transformation with a cost that is exponential in its theoretical worst-case complexity but feasible in practice.

1 Introduction

Abstraction is a key issue in model checking. Much attention has been given to *Boolean abstraction* (a.k.a. existential abstraction or predicate abstraction); see e.g. [10,15,6,16,13,11]. The idea of Boolean abstraction is to map states to 'abstract' states according their evaluation under a finite set of predicates (boolean expression over program variables) on states. The predicates induce an 'abstract' system with a transition relation over the abstract states. An approximation of the set of reachable concrete states (in fact, an inductive invariant) is obtained through a fixpoint of the 'abstract' post operator.

Motivated by the fact that computing the Boolean abstraction (i.e. computing the transition relation between 'abstract' states) is prohibitively costly, we propose a new abstraction, obtained by adding the *Cartesian abstraction* on top of the Boolean abstraction. The Cartesian abstraction underlies the *attribute independence* in certain kinds of program analysis (see [9]). It is used to approximate a set of tuples by the smallest Cartesian product containing this set. The new abstraction is induced by predicates over states, but it cannot be defined by a mapping over states (i.e., a state cannot be assigned a unique abstract value). We use the framework of *abstract interpretation* [8] and Galois connections to specify our abstraction as the formal composition of two abstractions.

We present an algorithm for computing the 'ideal' abstract post operator ("$\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}}$") wrt. the new abstraction (defined through a Galois connection). The algorithm is exponential in its worst-case complexity, but it is feasible in practice; it is the first algorithm in this context of abstract model checking that does not compute the value explicitly for each 'abstract' state. This gain in efficiency must, in theory, be traded with a *loss of precision*. We identify the single causes of loss of precision under the Cartesian abstraction. To eliminate most of these,

* On leave from Max Planck Institute, Saarbrücken, Germany.

we introduce three *refinements* of $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ that are based on standard concepts from program analysis. We have an implementation that combines all three refinements and that makes the loss of precision practically negligible.

The use of Cartesian abstraction allows us to represent the abstract post operator of a C program in form of a *Boolean program* [2]. Boolean program are C programs where all expressions and all variables range over the three truth values 1, 0 and $*$ (for 'unknown'). As C programs, Boolean programs have the usual update statements, and they may have procedures with call-by-value parameter passing, local variables, and recursion.

We next explain the specific context of our work. The SLAM project[1] at Microsoft Research is building processes and tools for checking temporal properties of system software written in common programming languages, such as C.

The existence of both *infinite control and infinite data* in (even sequential) software makes model checking of software difficult. Infinite control comes from procedural abstraction and recursion. Infinite data comes from the existence of unbounded data types such as integers and pointer-based data structures. Infinite control and unbounded arithmetic data has been studied in model checking in isolation, namely for pushdown systems resp. protocols, parameterized systems or timed and hybrid systems (see e.g. [17]). However, the combination of unbounded stack-based control and unbounded data has not been handled before.[2]

The SLAM project addresses this fundamental problem through a separation of concerns that abstracts infinite data domains through Cartesian and Boolean abstractions, and then uses well-known techniques [22,19] to analyze the resultant Boolean program, which has infinite control (but 'finite data'). The data are abstracted according to their evaluation under a given a set $\mathcal{P}$ of predicates on states of the C program.

Our working hypothesis is that for many interesting temporal properties of real-life system software, we can find suitable predicates such that the abstraction is precise enough to prove the desired invariant. Refinement can be accomplished by the addition of new predicates.

Given an invariant Inv to check on a C program, the SLAM process has three phases, starting with an initial set of predicates $\mathcal{P}$ and repeating the phases iteratively, halting if the invariant Inv is either proved or disproved (but possibly non-terminating):

1. construct an abstract post operator under the abstraction induced by $\mathcal{P}$;
2. model check the Boolean program that represents the abstract post operator;
3. discover new predicates and add them to the set $\mathcal{P}$ in order to refine the abstraction.

In this paper, we address the issue of abstraction in Phase (1). In principle, Phases (2) and (3) will follow the lines of other work on interprocedural program

[1] `http://research.microsoft.com/slam/`

[2] There are other promising attempts at model checking for software, of course, such as the Bandera project, for example, where *non-recursive* procedures are handled through inlining [7].

analysis [22,19], and abstraction refinement [4,15]). For more detail on Phase (2), see [2].

We note that the specific context of the SLAM project has the following consequences for the abstraction of the post operator and its computation in Phase (1):

- It is important to give a concise definition of the abstract post operator, not only to guide its implementation but also to guide the refinement process (i.e. to help identify the cause of imprecision in a given abstraction).
- The abstract post operator must be computed for its entire domain. That is, it cannot be restricted a priori to a subset of its domain. At the moment when the abstract post operator for a statement within a procedure is computed, it is generally impossible to foresee which values the statement will be applied to.

In the work on Boolean abstraction that is most closely related to ours, Graf and Saïdi [13] define an approximation of the Boolean abstraction of the `post` operator; our abstraction can be used to formalize that approximation in terms of a Galois connection, using a new abstract domain.

The procedure of [13] computes the image of their abstract `post` operator for each 'abstract state' with a linear number of calls to a theorem prover (in the number n of predicates inducing the Boolean abstraction). This is better than computing the image of the standard Boolean abstraction of the `post` operator, which requires exponentially many calls to a theorem prover; but still, it is only feasible if done 'on demand', i.e. for each reachable 'abstract' state (and if the number of those remains small).

In our setting, the procedure of [13] would require a fixed number $2^n \cdot 2 \cdot n$ of calls to a theorem prover. In this paper, we give a procedure with $O(2^n) \cdot 2 \cdot n$ calls; i.e., in comparison with [13], we replace the fixed (or best-case) factor 2^n by a worst-case factor $O(2^n)$, which makes all the difference for practical concerns.

2 Example C Program

In this paper, we are concerned with two SLAM tools: (1) `c2bp`, which takes a C program and a set of predicates, and produces an abstract post operator represented by a Boolean program [1], and (2) `bebop`, a model checker for Boolean programs. [2] We illustrate `c2bp` and `bebop` using a simple C program P shown in the left-hand-side of Figure 1. The property we want to check is that the assertion in line 9 is never reached, regardless of the context in which `foo` is called. The right-hand-side of Figure 1 shows the Boolean program B that `c2bp` produces from P, given the set of predicates { `(z==0)` , `(x==y)` }. The Boolean variables `b1` and `b2` represent the predicates `(z==0)` and `(x==y)`, respectively. Each statement of the C program is translated into a corresponding statement of the Boolean program. For example, the statement, `z = 0;` in line 2 is translated to `b1 := 1;`. The translation of the statement `x++;` in line 5 states that if `b2` is `1` before the statement, then it guaranteed to be `0` after the statement,

```
        int x, y, z, w;

        void foo()
        {
[1]       do {
[2]         z = 0;
[3]         x = y;
[4]         if (w){
[5]           x++;
[6]           z = 1;
            }
[7]       } while(x!=y)
[8]       if(z){
[9]             assert(0);
          }
        }
```

```
          decl b1, b2;
        /* b1 stands for predicate (z==0) and
           b2 stands for predicate (x==y) */
           void foo()
           begin
[1]          do
[2]            b1 := 1;
[3]            b2 := 1;
[4]            if (*) then
[5]              b2 := choose(0,b2);
[6]              b1 := 0;
               fi
[7]          while( b2 )
[8]          if (!b1) then
[9]            assert(0);
             fi
           end

           bool choose(e1,e2)
           begin
[10]         if (e1) then
[11]           return(1);
[12]         elsif (e2) then
[13]           return(0);
[14]         else
[15]           return(*);
             fi
          end
```

Fig. 1. An example C program, and the Boolean program produced by `c2bp` using predicates `(z==0)` and `(x==y)`

otherwise the value of `b2` after the statement is unknown, represented by `*` in line 15. The Boolean program B can be now fed to `bebop`, with the question: "is line 9 reachable in B?", and `bebop` answers "no". We thus conclude that line 9 is not reachable in the C program P as well.

3 Correctness

We fix a program (e.g. a C program) generating a transition system with a set States of states $s_1, s_2, \ldots$ and a transition relation $s \longrightarrow s'$. The operator post on sets of states is defined as usual: $\mathsf{post}(S) = \{s' \mid \text{ exists } s \in S : \ s \longrightarrow s'\}$.

In Section 7 we will use the 'weakest precondition' operator $\widetilde{\mathsf{pre}}$ on sets of states: $\widetilde{\mathsf{pre}}(S') = \{s \mid \text{ for all } s' \text{ such that } s \longrightarrow s' : \ s' \in S'\}$.

In order to define correctness, we fix a subset init of *initial* states and a subset unsafe of *unsafe* states (its complement $\mathsf{safe} = \mathsf{States} - \mathsf{unsafe}$ is the set of *safe* states). The set of reachable states (reachable from an initial state) is the least fixpoint of post that contains init, also called the closure of init under post, $\mathsf{post}^\star(\mathsf{init}) = \mathsf{init} \cup \mathsf{post}(\mathsf{init}) \cup \ldots$.

The given program is *correct* if no unsafe state is reachable; i.e., if $\mathsf{post}^\star(\mathsf{init}) \subseteq \mathsf{safe}$. A *safe (inductive) invariant* is a set of states S that contains the set of initial states, is a closure under the operator post and is contained in the set of all safe states, formally: $S \subseteq \mathsf{safe}$, $S \supseteq \mathsf{post}(S)$, and $S \supseteq \mathsf{init}$.

Correctness is established by computing a safe invariant. One way to do so is to find an 'abstraction' $\mathsf{post}^\#$ of the operator post and compute the closure of $\mathsf{post}^\#$ on init (and check that it is a subset of safe). In the next section, we will make the idea of abstraction formally precise.

4 Boolean Abstraction

For the purpose of this paper, we fix a finite set $\mathcal{P}$ of state predicates $\mathcal{P} = \{p_1, \ldots, p_n\}$. A predicate p_i denotes the subset of states that satisfy the predicate, $\{s \in \mathsf{States} \mid s \models p_i\}$. The predicate is usually defined by a Boolean expression over program variables.

We distinguish the terms approximation and abstraction. The set $\mathcal{P}$ of state predicates defines the *Boolean approximation* of a set of states S as $\mathsf{Boolean}(S)$, the smallest set containing S that can be denoted by a Boolean expression over predicates in $\mathcal{P}$ (formed as usual with the Boolean operators $\wedge, \vee, \neg$); this set is sometimes referred as the Boolean covering of the set. This approximation can be defined through an abstract domain and two functions α_{bool} and γ_{bool} that we define below (following the abstract interpretation framework [8]); namely, the Boolean approximation of a set of states S is the set of states $\mathsf{Boolean}(S) = \gamma_{\mathsf{bool}}(\alpha_{\mathsf{bool}}(S))$. The two functions are used to directly define the operator $\mathsf{post}^\#_{\mathsf{bool}}$ on the abstract domain as an *abstraction* of the fixpoint operator post over sets of states.

Having fixed $\mathcal{P}$, we define the *abstract domain* $\mathsf{AbsDom}_{\mathsf{bool}}$ as the set of all sets V of bitvectors v of length n (one bit per predicate $p_i \in \mathcal{P}$, for $i = 1, \ldots, n$), $\mathsf{AbsDom}_{\mathsf{bool}} = 2^{\{0,1\}^n}$, together with subset inclusion as the partial ordering. The abstraction function is the mapping from the *concrete domain* 2^{States}, the set of sets of states (again with subset inclusion as the partial ordering), to the abstract domain, assigning a set of states S the set of bitvectors representing the Boolean covering of S,

$$\begin{aligned}\alpha_{\mathsf{bool}} :\ & 2^{\mathsf{States}} \rightarrow \mathsf{AbsDom}_{\mathsf{bool}} \\ & S \mapsto \{\langle v_1, \ldots, v_n\rangle \mid S \cap \{s \mid s \models v_1 \cdot p_1 \wedge \ldots \wedge v_n \cdot p_n\} \neq \emptyset\}\end{aligned}$$

where $0 \cdot p_i = \neg p_i$ and $1 \cdot p_i = p_i$. The meaning function is the mapping

$$\begin{aligned}\gamma_{\mathsf{bool}} :\ & \mathsf{AbsDom} \rightarrow 2^{\mathsf{States}}, \\ & V \mapsto \{s \mid \text{exists } \langle v_1, \ldots, v_n\rangle \in V :\ s \models v_1 \cdot p_1 \wedge \ldots \wedge v_n \cdot p_n\}.\end{aligned}$$

Given $\mathsf{AbsDom}_{\mathsf{bool}}$ and the function α_{bool} (which forms a Galois connection together with the function γ_{bool}), the 'best' abstraction of the operator post is the operator $\mathsf{post}^{\#}_{\mathsf{bool}}$ on sets of bitvectors defined by

$$\mathsf{post}^{\#}_{\mathsf{bool}} = \alpha_{\mathsf{bool}} \circ \mathsf{post} \circ \gamma_{\mathsf{bool}}$$

where the functional composition $f \circ g$ of two functions f and g is defined from right to left; i.e., $f \circ g(x) = f(g(x))$.

5 Cartesian Abstraction

Given the vector domain $D_1 \times \ldots \times D_n$, the *Cartesian approximation* $\mathsf{Cartesian}(V)$ of a set of vectors V is the smallest Cartesian product of subsets of $D_1, \ldots, D_n$ that contains the set. It can be defined by the Cartesian product of the projections $\Pi_i(V)$, $\mathsf{Cartesian}(V) = \Pi_1(V) \times \ldots \times \Pi_n(V)$, where $\Pi_1(V) = \{v_1 \mid \langle v_1, \ldots, v_n \rangle \in V\}$ etc.. In order formalize the Cartesian approximation of a fixpoint operator, one uses the abstraction function from the concrete domain of sets of tuples to the abstract domain of tuples of sets (with pointwise subset inclusion as the partial ordering),

$$\begin{aligned} \alpha_{\mathsf{cartesian}} : 2^{D_1 \times \ldots \times D_n} &\rightarrow 2^{D_1} \times \ldots \times 2^{D_n} \\ V &\mapsto \langle \Pi_1(V), \ldots, \Pi_n(V) \rangle \end{aligned}$$

and the meaning function $\gamma_{\mathsf{cartesian}}$ mapping a tuple of sets $\langle M_1, \ldots, M_n \rangle$ to their Cartesian product $M_1 \times \ldots \times M_n$. I.e., we have $\mathsf{Cartesian}(V) = \gamma_{\mathsf{cartesian}} \circ \alpha_{\mathsf{cartesian}}(V)$.

In general, one has to account formally for the empty set (i.e., introduce a special bottom element $\perp$ and identify each tuple of sets that has at least one empty component); in the context of the fixpoints considered here (we look at the smallest fixpoint that is greater than a given element, e.g. $\alpha_{\mathsf{bool}}(\mathsf{init})$), we can gloss over this issue.

We next formalize the Cartesian approximation for sets of bitvectors. The nonempty sets of Boolean values are of one of three forms: $\{0\}$, $\{1\}$ or $\{0,1\}$. It is convenient to write 0 for $\{0\}$, 1 for $\{1\}$ and $*$ for $\{0,1\}$, and thus represent a tuple of sets of Boolean values by what we call a *trivector*, which is an element of $\{1, 0, *\}^n$. We therefore introduce the *abstract domain of trivectors*, $\mathsf{AbsDom}_{\mathsf{cartesian}} = \{0, 1, *\}^n$ (again, we gloss over the issue of a special trivector $\perp$). The partial ordering $<$ is the pointwise extension of the partial order given by $0 < *$ and $1 < *$; i.e., for two trivectors $\langle v_1, \ldots, v_n \rangle$ and $\langle v'_1, \ldots, v'_n \rangle$, $\langle v_1, \ldots, v_n \rangle < \langle v'_1, \ldots, v'_n \rangle$ if $v_1 < v'_1$, $\ldots$, $v_n < v'_n$. The Cartesian abstraction $\alpha_{\mathsf{cartesian}}$ maps a set of bitvectors V to a trivector,

$$\alpha_{\mathsf{cartesian}} : \mathsf{AbsDom}_{\mathsf{bool}} \rightarrow \mathsf{AbsDom}_{\mathsf{cartesian}},\ V \mapsto \langle v_1, \ldots, v_n \rangle$$

where, for $i = 1, \ldots, n$, (a) $v_i = 0$ if $\Pi_i(V) = \{0\}$; (b) $v_i = 1$ if $\Pi_i(V) = \{1\}$; (c) $v_i = *$ if $\Pi_i(V) = \{0,1\}$.

The meaning $\gamma_{\mathsf{cartesian}}(v)$ of a trivector v is the set of bitvectors that are smaller than v (wrt. the partial ordering giving on trivectors given above); i.e., it is the Cartesian product of the n sets of bit values denoted by the components of v. The meaning function $\gamma_{\mathsf{cartesian}} : \mathsf{AbsDom}_{\mathsf{cartesian}} \rightarrow \mathsf{AbsDom}_{\mathsf{bool}}$ forms a Galois connection with $\alpha_{\mathsf{cartesian}}$.

6 The Abstract Post Operator $\mathsf{post}^{\#}_{\mathsf{b{\cdot}c}}$ over Trivectors

We define a new Galois connection by composing the ones considered in the previous two sections,

$$\alpha_{\mathsf{b{\cdot}c}} : 2^{\mathsf{States}} \rightarrow \mathsf{AbsDom}_{\mathsf{cartesian}},\ \alpha_{\mathsf{b{\cdot}c}} = \alpha_{\mathsf{cartesian}} \circ \alpha_{\mathsf{bool}}$$

$$\gamma_{\mathsf{b{\cdot}c}} : \mathsf{AbsDom}_{\mathsf{cartesian}} \rightarrow 2^{\mathsf{States}},\ \gamma_{\mathsf{b{\cdot}c}} = \gamma_{\mathsf{bool}} \circ \gamma_{\mathsf{cartesian}}$$

and the abstract post operator over trivectors, $\mathsf{post}^{\#}_{\mathsf{b{\cdot}c}} : \mathsf{AbsDom}_{\mathsf{cartesian}} \rightarrow \mathsf{AbsDom}_{\mathsf{cartesian}}$, defined by $\mathsf{post}^{\#}_{\mathsf{b{\cdot}c}} = \alpha_{\mathsf{b{\cdot}c}} \circ \mathsf{post} \circ \gamma_{\mathsf{b{\cdot}c}}$.

We have thus given a formalization of the fixpoint operator that implicitly defines the invariant Inv_1 given by $\mathcal{I}_1$ in [13]; i.e., the invariant is the meaning (under $\gamma_{\mathsf{b{\cdot}c}}$) of the least fixpoint of $\mathsf{post}^{\#}_{\mathsf{b{\cdot}c}}$ that is not smaller than the abstraction of init (under $\alpha_{\mathsf{b{\cdot}c}}$), or $\mathsf{Inv}_1 = \gamma_{\mathsf{b{\cdot}c}}(\mathsf{post}^{\#\,\star}_{\mathsf{b{\cdot}c}}(\alpha_{\mathsf{b{\cdot}c}}(\mathsf{init})))$. The invariant Inv_1 is represented abstractly by one trivector, i.e. it is the Cartesian product of sets each described by p, $\neg p$ or $p \vee \neg p$ (i.e. true) where p is a predicate of the set $\mathcal{P}$.

7 The c2bp Algorithm to Compute $\mathsf{post}^{\#}_{\mathsf{b{\cdot}c}}$

The algorithm takes as input the transition system (defining the operators post and $\widetilde{\mathsf{pre}}$) and the set of n predicates $\mathcal{P}$; as output it produces the representation of $\mathsf{post}^{\#}_{\mathsf{b{\cdot}c}}$ in the form of a Boolean program over n 'Boolean' variables $v_1, \ldots, v_n$ (whose values range over the domain $\{0, 1, *\}$). Each statement of the Boolean program is a multiple assignment statement of the form $\langle v_1, \ldots, v_n\rangle := \langle e_1, \ldots, e_n\rangle$, where $e_1, \ldots, e_n$ are expressions over $v_1, \ldots, v_n$ that are evaluated to a value in $\{0, 1, *\}$. We write $e[v_1, \ldots, v_n]$ for e if we want to stress that e is an expression over $v_1, \ldots, v_n$. The Boolean program represents the operator $\mathsf{post}^{\#}_{\mathsf{c2bp}}$ over trivectors by

$$\mathsf{post}^{\#}_{\mathsf{c2bp}}(\langle v_1, \ldots, v_n\rangle) = \langle v_1', \ldots, v_n'\rangle \ \text{ if } \ v_1' = e_1[v_1, \ldots, v_n], \ldots, v_n' = e_n[v_1, \ldots, v_n].$$

We will now explain how the algorithm computes the expressions $e_i[v_1, \ldots, v_n]$, for each $i = 1, \ldots, n$. We first define the *Boolean under-approximation* of a set S wrt. $\mathcal{P}$ as the greatest Boolean expression over predicates in $\mathcal{P}$ whose denotation is contained in S; formally, $\mathsf{F}(S) = \nu E \in \mathsf{BoolExpr}(\mathcal{P}).\ \{s \mid s \models E\} \subseteq S$. That is, the set of states denoted by $\mathsf{F}(S)$ is $\mathsf{States} - (\gamma_{\mathsf{bool}} \circ \alpha_{\mathsf{bool}})(\mathsf{States} - S)$. For the purpose of defining the algorithm, the set $\mathsf{BoolExpr}(\mathcal{P})$ consists of Boolean expressions in the form of disjunctions of conjunctions of possibly negated predicates from $\mathcal{P}$. The ordering $e < e'$ is such that each disjunct of e implies some disjunct of e' (e.g., p_1 is greater than $p_1 \wedge p_2 \vee p_1 \wedge \neg p_2$).

By repeated calls to a theorem prover,[3] the algorithm computes the two Boolean expressions $E_i(0)$ and $E_i(1)$ over the predicates $p_1, \dots, p_n$

$$\begin{aligned} E_i(0) &= \mathsf{F}(\widetilde{\mathsf{pre}}(\{s \mid s \models \neg p_i\})), \\ E_i(1) &= \mathsf{F}(\widetilde{\mathsf{pre}}(\{s \mid s \models p_i\})). \end{aligned}$$

We define the two Boolean expressions $e_i(1)$ and $e_i(0)$ over the variables $v_1, \dots, v_n$ by direct correspondence from the two Boolean expressions $E_i(0)$ and $E_i(1)$ over the predicates $p_1, \dots, p_n$.

The expression e_i over the variables $v_1, \dots, v_n$ that defines the i-th value of the successor trivector of the Boolean program is $e_i = \mathsf{choose}(e_i(1), e_i(0))$, where the symbol choose stands for an if-then–elseif-then–else combinator on two Boolean expressions; i.e., the expression $\mathsf{choose}(e, e')$ applied to two Boolean expressions e and e', each over the variables $v_1, \dots, v_n$, evaluates as follows:

$$\begin{aligned} \mathsf{choose}(e[v_1, \dots, v_n], e'[v_1, \dots, v_n]) = \quad & \text{if} \;\; \langle v_1, \dots, v_n \rangle \models e \text{ then } 1 \\ & \text{elseif} \;\; \langle v_1, \dots, v_n \rangle \models e' \text{then } 0 \\ & \text{else } * \end{aligned}$$

The satisfaction of a Boolean expression e by a trivector $\langle v_1, \dots, v_n \rangle$ is defined as one expects, namely $\langle v_1, \dots, v_n \rangle \models e$ if all bitvectors in $\gamma_{\mathsf{bool}}(\langle v_1, \dots, v_n \rangle)$ satisfy e. Thus, for example, $\langle 0, 1, * \rangle \models \neg v_1 \wedge v_2$ but $\langle 0, 1, * \rangle \not\models v_3$ and $\langle 0, 1, * \rangle \not\models \neg v_3$. (The extension of the Boolean operators to the domain $\{0, 1, *\}$ is defined accordingly.)

Proposition 1 (Correctness). *The result of the* c2bp *algorithm is a Boolean program representing the Boolean and Cartesian abstraction of the operator* post*, i.e.* $\mathsf{post}^\#_{\mathsf{c2bp}} = \mathsf{post}^\#_{\mathsf{b\cdot c}}$.

Proof. We define the n abstraction functions $\alpha^{(i)}_{\mathsf{b\cdot c}}$ by

$$\alpha^{(i)}_{\mathsf{b\cdot c}}(M) = \begin{cases} 1 \text{ if } M \subseteq \{s \mid s \models p_i\} \\ 0 \text{ if } M \subseteq \{s \mid s \models \neg p_i\} \\ * \text{ if neither} \end{cases}$$

and the i-th abstract post function $\mathsf{post}^{\#\,(i)}_{\mathsf{b\cdot c}}$ by $\mathsf{post}^{\#\,(i)}_{\mathsf{b\cdot c}} = \alpha^{(i)}_{\mathsf{b\cdot c}} \circ \mathsf{post} \circ \gamma_{\mathsf{b\cdot c}}$.

Since the value of any nonempty set of states S under the abstraction $\alpha_{\mathsf{b\cdot c}}$ is the trivector

$$\alpha_{\mathsf{b\cdot c}}(S) = \langle \alpha^{(1)}_{\mathsf{b\cdot c}}(S), \dots, \alpha^{(n)}_{\mathsf{b\cdot c}}(S) \rangle,$$

[3] We consider the theorem prover as an oracle, which does exist for most practical concerns. It is easy to see that theoretically such an oracle does not exist and that $\mathsf{post}^\#_{\mathsf{b\cdot c}}$ (or $\mathsf{post}^\#_{\mathsf{bool}}$) cannot be computed; i.e., the problem of deciding whether an operator is equal to $\mathsf{post}^\#_{\mathsf{b\cdot c}}$ (or $\mathsf{post}^\#_{\mathsf{bool}}$) is undecidable.

we can express the abstract post operator $\mathsf{post}^{\#}_{\mathsf{b \cdot c}}$ over trivectors as the tuple of the abstract post functions, each mapping trivectors to values in $\{0, 1, *\}$,

$$\mathsf{post}^{\#}_{\mathsf{b \cdot c}}(\langle v_1, \dots, v_n\rangle) = \langle {\mathsf{post}^{\#}_{\mathsf{b \cdot c}}}^{(1)}(\langle v_1, \dots, v_n\rangle), \dots, {\mathsf{post}^{\#}_{\mathsf{b \cdot c}}}^{(n)}(\langle v_1, \dots, v_n\rangle)\rangle.$$

Now, we can represent the abstract post operator $\mathsf{post}^{\#}_{\mathsf{b \cdot c}}$ in terms of the sets $V_i(0)$, $V_i(1)$ and $V_i(*)$, defined as the inverse images of the values 0, 1 or $*$, respectively, under the i-th abstract post functions ${\mathsf{post}^{\#}_{\mathsf{b \cdot c}}}^{(i)}$.

$$\begin{aligned}
V_i(0) &= \{\langle v_1, \dots, v_n\rangle \mid {\mathsf{post}^{\#}_{\mathsf{b \cdot c}}}^{(i)}(\langle v_1, \dots, v_n\rangle) = 0\}\\
V_i(1) &= \{\langle v_1, \dots, v_n\rangle \mid {\mathsf{post}^{\#}_{\mathsf{b \cdot c}}}^{(i)}(\langle v_1, \dots, v_n\rangle) = 1\}\\
V_i(*) &= \{\langle v_1, \dots, v_n\rangle \mid {\mathsf{post}^{\#}_{\mathsf{b \cdot c}}}^{(i)}(\langle v_1, \dots, v_n\rangle) = *\}\\
&= \mathsf{AbsDom}_{\mathsf{cartesian}} - (V_i(0) \cup V_i(1))
\end{aligned}$$

The statement of the proposition can now be expressed by the fact that the sets $V_i(0)$, $V_i(1)$ and $V_i(*)$ are exactly the sets of trivectors that satisfy the Boolean expressions $e_i(0)$, $e_i(1)$ or neither.

$$\begin{aligned}
V_i(0) &= \{\langle v_1, \dots, v_n\rangle \mid \langle v_1, \dots, v_n\rangle \models e_i(0)\}\\
V_i(1) &= \{\langle v_1, \dots, v_n\rangle \mid \langle v_1, \dots, v_n\rangle \models e_i(1)\}\\
V_i(*) &= \{\langle v_1, \dots, v_n\rangle \mid \langle v_1, \dots, v_n\rangle \not\models e_i(0),\ \langle v_1, \dots, v_n\rangle \not\models e_i(1)\}
\end{aligned} \tag{1}$$

That is, in order to prove the proposition we need to prove (1).

Since $\mathsf{AbsDom}_{\mathsf{bool}}$ is a complete distributive lattice, the membership of a trivector $\langle v_1, \dots, v_n\rangle$ in $V_i(0)$ is equivalent to the condition that $\gamma_{\mathsf{cartesian}}(\langle v_1, \dots, v_n\rangle)$ is contained in $B_i(0)$, the largest set of bitvectors that is mapped to the value 0 by the function $\alpha^{(i)}_{\mathsf{b \cdot c}} \circ \mathsf{post}\gamma_{\mathsf{bool}}$. That is, if we define

$$B_i(0) = \nu B \in \mathsf{AbsDom}_{\mathsf{bool}}.\ \alpha^{(i)}_{\mathsf{b \cdot c}} \circ \mathsf{post} \circ \gamma_{\mathsf{bool}}(B) = 0$$

then

$$V_i(0) = \{\langle v_1, \dots, v_n\rangle \in \mathsf{AbsDom}_{\mathsf{cartesian}} \mid \gamma_{\mathsf{bool}}(\langle v_1, \dots, v_n\rangle) \subseteq B_i(0)\}. \tag{2}$$

By definition of $\alpha^{(i)}_{\mathsf{b \cdot c}}$, we can express the set of bitvectors $B_i(0)$ as

$$B_i(0) = \nu B \in \mathsf{AbsDom}_{\mathsf{bool}}.\ \mathsf{post} \circ \gamma_{\mathsf{bool}}(B) \subseteq \{s \mid s \models \neg p_i\}.$$

The operators post and $\widetilde{\mathsf{pre}}$ form a Galois connection, i.e. $\mathsf{post}(S) \subseteq S'$ if and only if $S \subseteq \widetilde{\mathsf{pre}}(S')$. Therefore, we can write $B_i(0)$ equivalently as

$$B_i(0) = \nu B \in \mathsf{AbsDom}_{\mathsf{bool}}.\ \gamma_{\mathsf{bool}}(B) \subseteq \widetilde{\mathsf{pre}}(\{s \mid s \models \neg p_i\}).$$

Thus, $B_i(0)$ is exactly the set of all bitvectors that satisfy the Boolean expression $e_i(0)$.

$$B_i(0) = \{\langle v_1, \dots, v_n\rangle \in \{0, 1\}^n \mid \langle v_1, \dots, v_n\rangle \models e_i(0)\}$$

This fact, together with (2), yields directly the characterization of $V_i(0)$ in (1). The other two statements in (1) follow in the similar way. □

Complexity. We need to compute $\mathsf{F}(S)$ for $2n$ sets S that are either of the form $S = \widetilde{\mathsf{pre}}(\{s \in \mathsf{States} \mid s \models p_i\}$ or of the form $S = \widetilde{\mathsf{pre}}(\{s \in \mathsf{States} \mid s \models \neg p_i\}$.

In order to compute each $\mathsf{F}(S)$, we need to find all minimal implicants of S in the form of a *cube*, i.e. a conjunction $\mathcal{C} = \bigwedge_{i \in I} \ell_i$ of possibly negated predicates (i.e., ℓ_i is p_i or $\neg p_i$) such that $\{s \mid s \models \mathcal{C}\} \subseteq S$. We use some quick syntactic checks to find which of the predicates p_i can possibly influence S (i.e. such p_i or $\neg p_i$ can appear in a minimal implicant); usually, there are only few of those. 'Minimal' here means: if an implicant $\mathcal{C}$ is found, no larger conjunction $\mathcal{C} \wedge p_j$ needs to be considered. Also, if $\mathcal{C}$ is incompatible with S (i.e., $\{s \mid s \models \mathcal{C}\} \cap S = \emptyset$), no larger conjunction needs to be considered (since no conjunction $\mathcal{C} \wedge p_j$ can be an implicant).

8 Loss of Precision under Cartesian Abstraction

We will next analyze in what way precision may get lost through the Cartesian abstraction. It is important to distinguish that loss from the one that incurs from the Boolean abstraction. The latter is addressed by adding new predicates in the refinement phase.

'Loss of precision' is made formally precise in the following way (see [8,12]). Given a concrete and an abstract domain, an abstraction α and a meaning γ, we say that the operator F does not lose precision under the abstraction to $F^{\#}$ if $\gamma \circ F^{\#} = F \circ \gamma$ (i.e., does not lose precision on the abstract value a if $\gamma \circ F^{\#}(a) = F \circ \gamma(a)$).

In our setting, F will always be instantiated by $\mathsf{post}^{\#}_{\mathsf{bool}}$. In this section, the phrase 'the Cartesian abstraction does not lose precision' is short for '$\mathsf{post}^{\#}_{\mathsf{bool}}$ does not lose precision under the abstraction to $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$', i.e. $\gamma_{\mathsf{cartesian}} \circ \mathsf{post}^{\#}_{\mathsf{b\cdot c}} = \mathsf{post}^{\#}_{\mathsf{bool}} \circ \gamma_{\mathsf{cartesian}}$. We define an operator F on sets to be *deterministic* if it maps a singleton set to the empty set or another singleton set. The following observation will be used in Section 9.3:

Proposition 2. *If the operator* $\mathsf{post}^{\#}_{\mathsf{bool}}$ *is deterministic, then the Cartesian abstraction does not lose precision on trivectors* $\langle v_1, \ldots, v_n \rangle$ *such that* $v_i \neq *$, *for* $1 \leq i \leq n$.

Example 1. We take the (simple and somewhat contrived) example of the C program with one statement `x = y` updating `x` by `y` and the set of predicates $\mathcal{P} = \{p_1, p_2, p_2\}$ where p_1 expresses "$x > 5$", p_2 expresses "$x < 5$" and p_3 expresses "$y = 5$". Note that the conjunction of $\neg p_1$ and $\neg p_2$ expresses $x = 5$. The image of the trivector $\langle 0, 0, 0 \rangle$ under the abstract post operator $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ is the trivector $\langle *, *, 0 \rangle$. Therefore, $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}(\langle 0, 0, 0 \rangle) = \langle *, *, 0 \rangle$ because $\mathsf{post}^{\#}_{\mathsf{b\cdot c}} =$

$\alpha_{\mathsf{cartesian}} \circ \alpha_{\mathsf{bool}} \circ \mathsf{post} \circ \gamma_{\mathsf{bool}} \circ \gamma_{\mathsf{cartesian}}$ and by the following equalities.

$$\begin{aligned}
\gamma_{\mathsf{cartesian}}(\langle 0,0,0\rangle) &= \{\langle 0,0,0\rangle\} && \in \mathsf{AbsDom}_{\mathsf{bool}}\\
\gamma_{\mathsf{bool}}(\{\langle 0,0,0\rangle\}) &= \{\langle x,y\rangle \mid x=5,\ y\neq 5\} && \in 2^{\mathsf{States}}\\
\mathsf{post}(\{\langle x,y\rangle \mid x=5,\ y\neq 5\}) &= \{\langle x,y\rangle \mid x=y,\ y\neq 5\} && \in 2^{\mathsf{States}}\\
\alpha_{\mathsf{bool}}(\{\langle x,y\rangle \mid x=y,\ y\neq 5\}) &= \{\langle 1,0,0\rangle, \langle 0,1,0\rangle\} && \in \mathsf{AbsDom}_{\mathsf{bool}}\\
\alpha_{\mathsf{cartesian}}(\{\langle 1,0,0\rangle, \langle 0,1,0\rangle\}) &= \langle *,*,0\rangle && \in \mathsf{AbsDom}_{\mathsf{cartesian}}
\end{aligned}$$

The meaning of the trivector $\langle *,*,0\rangle$ is a set of four bitvectors that properly contains the image of the Boolean abstraction of the post operator $\mathsf{post}^{\#}_{\mathsf{bool}}$ applied to the meaning of the trivector $\langle 0,0,0\rangle$.

$$\begin{aligned}
\gamma_{\mathsf{cartesian}}(\mathsf{post}^{\#}_{\mathsf{b\cdot c}}(\langle 0,0,0\rangle)) &= \{\langle 0,0,0\rangle, \langle 1,0,0\rangle, \langle 0,1,0\rangle, \langle 1,1,0\rangle\}\\
&\supset \{\langle 1,0,0\rangle, \langle 0,1,0\rangle\}\\
&= \mathsf{post}^{\#}_{\mathsf{bool}}(\gamma_{\mathsf{cartesian}}(\langle 0,0,0\rangle))
\end{aligned}$$

That is, the Cartesian abstraction loses precision by adding the bitvector $\langle 0,0,0\rangle$ (expressing $x = 5$ through the negation of both, $x < 5$ and $x > 5$) to the two bitvectors $\langle 1,0,0\rangle$ and $\langle 0,1,0\rangle$ that form the image of the Boolean abstract post operator. (The added bitvector $\langle 1,1,0\rangle$ is semantically inconsistent and will be eliminated by standard methods in Boolean abstraction; see [13].) Note that the concrete operator post is *deterministic*; the loss of precision in the Cartesian abstraction occurs because $\mathsf{post}^{\#}_{\mathsf{bool}}$ is not deterministic ($\mathsf{post}^{\#}_{\mathsf{bool}}(\langle 0,0,0\rangle) = \{\langle 1,0,0\rangle, \langle 0,1,0\rangle\}$; as an aside, post does not lose precision under the Boolean abstraction).

Example 2. The next example is simpler than the previous one but it is not relevant in the context of C programs where the transition relation is deterministic. Nondeterminism arises in the interleaving semantics of concurrent systems. Take a program with Boolean variables x and y (standing e.g. for 'critical') and the transition relation specified by the assertion $x' = \neg y'$ (as usual, a primed variable stands for the variable's value after the transition). For simplicity of presentation, we here identify states and bitvectors. The image of every nonempty set of bitvectors under $\mathsf{post}^{\#}_{\mathsf{bool}}$ is the set of bitvectors $\{\langle 0,1\rangle, \langle 1,0\rangle\}$. The image of every trivector under $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ is the trivector $\langle *,*\rangle$ whose meaning is the set of all bitvectors. Here again, $\mathsf{post}^{\#}_{\mathsf{bool}}$ is not deterministic. Unlike the previous example, the concrete operator post is not deterministic as well.

Example 3. The next example shows, in the setting of a deterministic transition relation, that precision can get lost if $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ is applied to a trivector with components having value $*$. Take a program with 2 Boolean variables x_1, x_2 and the transition relation specified by the statement "assume($x_1 = x_2$)"; its post operator, defined by $\mathsf{post}(V) = \{\langle v_1, v_2\rangle \in V \mid v_1 = v_2\}$, is equal to its

Boolean abstraction. The image of the trivector $\langle *, * \rangle$ under $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ is the trivector $\langle *, * \rangle$. The image of its meaning $\gamma_{\mathsf{cartesian}}(\langle *, * \rangle)$ under $\mathsf{post}^{\#}_{\mathsf{bool}}$ is the set of bitvectors $\{\langle 0, 0 \rangle, \langle 1, 1 \rangle\}$.

We will come back to this example in Section 9.3; there, we will also consider the general version of the same program with $n \geq 2$ Boolean variables $x_1, \ldots, x_n$ and the transition relation specified by the assertion $x_1 = x_2 \wedge x'_1 = x_1 \wedge \ldots \wedge x'_n = x_n$. The image of the trivector $\langle *, \ldots, * \rangle$ under $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ is the trivector $\langle *, \ldots, * \rangle$. The image of its meaning under $\mathsf{post}^{\#}_{\mathsf{bool}}$ is the set of all bitvectors whose first two components are equal.

9 Refinement for $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$

In this section, we apply standard methods from program analysis and propose refinements of the Cartesian abstraction; these are orthogonal to the refinement of the Boolean abstraction by iteratively adding new predicates.

9.1 Control Points

We now assume a preprocessing step on the program to be checked which introduces new control points (locations). Each conditional statement (with, say, condition ϕ) is replaced by a nondeterministic branching (each nondeterministic edge going to a different location), followed by a (deterministic) edge enforcing the condition ϕ or its negation ("assume(ϕ)" or "assume($\neg\phi$)") as a blocking invariant, followed by a (deterministic) edge with the update statement, followed by "joining" edges to the location after the original conditional statement.

Until now, we implicitly assumed predicates p_ℓ for every control point ℓ of the program (expressing that a state is at location ℓ). This would lead to a great loss of precision under the abstraction considered above. Instead, one formalizes the concrete domain as the sequence $(2^{\mathsf{States}})^{\mathsf{Loc}}$ of state spaces indexed by program locations $\ell \in \mathsf{Loc}$. Its elements are vectors $\boldsymbol{S} = \langle \boldsymbol{S}[\ell] \rangle_{\ell \in \mathsf{Loc}}$ of sets of states, i.e. $\boldsymbol{S}[\ell] \in 2^{\mathsf{States}}$. From now on, a state $s \in \mathsf{States}$ consists only of the environment of the data variables of the program. Accordingly, the abstract domain is the sequence $(\mathsf{AbsDom}_{\mathsf{cartesian}})^{\mathsf{Loc}}$.[4]

The post operator is now a tuple of post operators post_ℓ, one for each location ℓ of the control flow graph, $\mathsf{post} = \langle \mathsf{post}[\ell] \rangle_{\ell \in \mathsf{Loc}}$, where $\mathsf{post}[\ell]$ is defined in the standard way. We define the abstract post operator accordingly as the tuple $\mathsf{post}^{\#}_{\mathsf{b\cdot c}} = \langle \mathsf{post}^{\#}_{\mathsf{b\cdot c}}[\ell] \rangle_{\ell \in \mathsf{Loc}}$.

If ℓ is the "join" location after a conditional statement and its two predecessors are ℓ_1 and ℓ_2, then $\mathsf{post}[\ell](\boldsymbol{S}) = S[\ell_1] \cup S[\ell_2]$. We define the ℓ-th abstract

[4] Note that we don't need to model the procedure stack associated with the state. This is because the stack is implicitly present in the semantics of the Boolean program, and hence does not need to be abstracted by `c2bp`. Procedure call and return are handled essentially in the same way as assignment statements. See [1] for details.

post operator, $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}[\ell](\langle \ldots, v[\ell_1], \ldots, v[\ell_2], \ldots \rangle) = v[\ell_1] \sqcup v[\ell_2]$ where $v \sqcup v'$ is the least upper bound of the two trivectors v and v' in $\mathsf{AbsDom}_{\mathsf{cartesian}}$.

In all other cases, there is a unique predecessor location for ℓ, and $\mathsf{post}[\ell]$ is defined by the transition relation for the unique edge leading into ℓ. The ℓ-th abstract post operator is then defined (and computed) as described in the preceding sections, $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}[\ell] = \alpha_{\mathsf{b\cdot c}} \circ \mathsf{post}[\ell] \circ \gamma_{\mathsf{b\cdot c}}$.

Specializing the observations in Section 8, we now study the loss of precision of $\mathsf{post}^{\#}_{\mathsf{bool}}[\ell]$ under the Cartesian abstraction specifically for each kind of location ℓ. There is no loss of precision if the edge leading into ℓ is one of the two nondeterministic branches corresponding to a conditional since all data values are unchanged.

If the edge corresponds to an "assume(ϕ)" statement (its post operator is defined by $\mathsf{post}(S) = \{s \in S \mid s \models \phi\}$ for $S \subseteq \mathsf{States}$), then there is a loss of precision exactly if ϕ expresses a dependence between variables (such as $x = y$ as in Example 3); Proposition 2 applies, since the operator $\mathsf{post}^{\#}_{\mathsf{bool}}[\ell]$ is deterministic; we have $\mathsf{post}^{\#}_{\mathsf{bool}}(V) = V \cap \alpha_{\mathsf{bool}}(\{s \in \mathsf{States} \mid s \models \phi\}$.

If the edge corresponds to an update statement, then (and only then) the operator $\mathsf{post}^{\#}_{\mathsf{bool}}[\ell]$ may not be deterministic (even if the concrete operator $\mathsf{post}[\ell]$ is deterministic; see Example 1). If ℓ is a "join" location, then the loss of precision is apparent: the union of two Cartesian products gets approximated by a Cartesian product. This loss of precision gets eliminated by the refinement of the next section.

9.2 Disjunctive Completion

Following standard methods from program analysis [8], we go from the abstract domain of trivectors $\mathsf{AbsDom}_{\mathsf{cartesian}}$ to its *disjunctive completion*, which we may model as the abstract domain of *sets* of trivectors, $\mathsf{AbsDom}_{\mathsf{b\cdot c\cdot\vee}} = 2^{\{0,1,*\}^n}$ with the partial ordering $\sqsubset$ obtained by extending the ordering $<$ on trivectors, i.e., for two sets V and V' of trivectors, we have $V \sqsubset V'$ if for all trivectors $v \in V$ there exists a trivector $v' \in V'$ such that $v < v'$. For our purposes, the least element of the abstract domain $\mathsf{AbsDom}_{\mathsf{b\cdot c\cdot\vee}}$ is the set $\{\alpha_{\mathsf{cartesian}} \circ \alpha_{\mathsf{bool}}(\mathsf{init})\}$.

Note that the two domains $\mathsf{AbsDom}_{\mathsf{bool}}$ and $\mathsf{AbsDom}_{\mathsf{b\cdot c\cdot\vee}}$ are not isomorphic; we have that $V_1 = \{\langle 0,*\rangle, \langle 1,*\rangle\}$ is strictly smaller than $V_2 = \{\langle *,*\rangle\}$. The *reduced quotient* of $\mathsf{AbsDom}_{\mathsf{b\cdot c\cdot\vee}}$ (obtained by identifying sets with the same meaning, such as V_1 and V_2) is isomorphic to $\mathsf{AbsDom}_{\mathsf{bool}}$; there, the fixpoint test is exponentially more expensive than in $\mathsf{AbsDom}_{\mathsf{b\cdot c\cdot\vee}}$ (but may be practically feasible if symbolic representations are used).

The abstract post operator $\mathsf{post}^{\#}_{\mathsf{b\cdot c\cdot\vee}}$ over sets of trivectors $V \in \mathsf{AbsDom}_{\mathsf{b\cdot c\cdot\vee}}$ is the canonical extension of the abstract post operator over trivectors to a function over sets of trivectors, i.e., for $V \in 2^{\{0,1,*\}^n}$, $\mathsf{post}^{\#}_{\mathsf{b\cdot c\cdot\vee}}(V) = \{\mathsf{post}^{\#}_{\mathsf{b\cdot c}}(v) \mid v \in V\}$.

9.3 The Focus Operation

Assuming the refinement to the disjunctive completion, we now introduce the *focus* operation (the terminology stems from an—as it seems to us, related—

operation in shape analysis via 3-valued logic [20]). This operation can be used to eliminate all loss of precision under Cartesian abstraction unless the Boolean abstraction of the post operator $\mathsf{post}[\ell]$ at location ℓ is nondeterministic (as in Examples 1 and 2).

The idea of the focus operator can be explained at hand of Example 3. Here, the assertion defining the operator post associated with the "assume($x_1 = x_2$)" statement (which corresponds to the assertion "$x_1 = x_2 \wedge x_1' = x_1 \wedge x_2' = x_2$") expresses a dependence between the variables x_1 and x_2. Therefore, one defines the focus operation $\mathsf{focus}[1,2]$ that, if applied to a trivector of length $n \geq 2$, replaces the value $*$ in its first and second components; i.e.,

$$\mathsf{focus}[1,2](\langle v_1, v_2, v_3, \dots, v_n\rangle) = \\ \{\langle v_1', v_2', v_3, \dots, v_n\rangle \mid v_1', v_2' \in \{0,1\},\ v_1' \leq v_1,\ v_2' \leq v_2\}.$$

We extend the operation from trivectors v to sets of trivectors V in the canonical way. We are now able to define the 'focussed' abstract post operator $\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}\cdot\vee\cdot[1,2]}$ as follows (refining the operator $\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}}$ given in the previous section).

$$\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}\cdot\vee\cdot[1,2]}(V) = \{\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}}(v) \mid v \in \mathsf{focus}[1,2](V)\}$$

Continuing Example 3, we have that $\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}\cdot\vee\cdot[1,2]}(\{\langle *, *\rangle\}) = \{\langle 0,0\rangle, \langle 1,1\rangle\}$, which means that the operator post does not lose precision under the 'focussed' abstraction (i.e., the meaning function composed with $\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}\cdot\vee\cdot[1,2]}$ equals post composed with the meaning function). Note that in general, the focus operation may yield trivectors with components $*$. Continuing Example 3 for the general version of the program with $n \geq 2$ Boolean variables, we have $\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}\cdot\vee\cdot[1,2]}(\{\langle *, *, *, \dots, *\rangle\}) = \{\langle 0, 0, *, \dots, *\rangle, \langle 1, 1, *, \dots, *\rangle\}$.

The definitions above generalize directly to focus operations in other than the first two and more than two components. The following observation follows directly from Proposition 2.

Proposition 3. *For every deterministic operator* post*, there exists a focus operation such that* post *does not lose precision under the 'focussed' Cartesian abstraction.*

The abstract post operator $\mathsf{post}^{\#}_{\mathsf{slam}}$ used in SLAM results from combining the three refinements presented in Sections 9.1, 9.2 and 9.3, with the *total* focus operation $\mathsf{focus}[1, 2, \dots, n]$. I.e., for each each control point ℓ in the program, we have: $\mathsf{post}^{\#}_{\mathsf{slam}}[\ell] = \mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}\cdot\vee\cdot[1,\dots,n]}[\ell]$.

By Proposition 3, for every ℓ such that $\mathsf{post}^{\#}_{\mathsf{bool}}[\ell]$ is deterministic, the abstraction to $\mathsf{post}^{\#}_{\mathsf{slam}}[\ell]$ does not lose precision. A symbolic model checker such as bebop can realize the disjunctive completion and the total focus operation by representing and manipulating a set of trivectors V always in its 'focussed' version, i.e. the set of bitvectors $\mathsf{focus}[1, 2, \dots, n](V)$. In a symbolic representation, the gain of precision obtained by using the disjunctive completion and the total focus operation comes at no cost. More precisely, the two Boolean formulas representing V and $\mathsf{focus}[1, 2, \dots, n](V)$ simplify to the same form (e.g., *true* represents $\{\langle *, \dots, *\rangle\}$ as well as $\{0,1\}^n$).

10 Conclusion

Abstraction is probably the single most important issue in model checking software. Our work goes beyond the standard abstraction used in model checking, the so-called Boolean abstraction. We use the abstract domain of trivectors (with a third truth value $*$) in order to define a new abstraction function $\alpha_{\mathsf{b\cdot c}}$ in terms of Boolean and Cartesian abstraction, and an abstract post operator $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ in terms of a Galois connection. We present a practically feasible algorithm to compute the new abstraction, represented as a Boolean program. Previous algorithms on related Boolean abstractions were practical only when restricted to a small subset of states; that restriction is not possible in our setting, which addresses programs with recursive procedures.

We have implemented both the tools `c2bp` and `bebop`. We have used `c2bp` and `bebop` to successfully check properties of a Windows NT device driver for the serial port. The driver has a few thousand lines of C code. More details and a case study on using SLAM tools to check properties of Windows NT device drivers will appear in a forthcoming paper.

The new abstraction trades a crucial gain of efficiency with a loss of precision (by ignoring dependencies between the Boolean variables). We single out the different causes of a proper loss of precision and are able to eliminate all but one. It may be interesting to determine general conditions ensuring that no proper loss of precision can ever occur, phrased e.g. in terms of separability [18].

The formal machinery developed here has potentially other applications in designing new abstractions for model checking software, in explaining existing approaches to pointer analysis based on 3-valued logic [20], and in classifying data-flow analysis problems modeled as model checking problems [23,21]. Previous work relating the Boolean abstraction to bisimulation and temporal properties (e.g. [5,10,6,16]) should be re-examined in the light of the new abstraction, perhaps in terms of 3-valued transition systems [14].

Acknowledgements. We thank Todd Millstein and Rupak Majumdar for their work on `c2bp`, and Bertrand Jeannet and Laurent Mauborgne for their helpful comments.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of SIGPLAN Conference on Proramming Language Design and Implementation, 2001 (to appear)*. ACM, 2001.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.
3. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.
5. E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *POPL 92: Principles of Programming Languages*, pages 343–354. ACM, 1992.
6. R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in abstractions of model checking. In *SAS 95: Static Analysis*, LNCS 983, pages 51–63. Springer-Verlag, 1995.
7. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000: International Conference on Software Engineering*, pages 439–448. ACM, 2000.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.
9. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA 95: Functional Programming and Computer Architecture*, pages 170–181. ACM, 1995.
10. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: abstractions preserving ACTL*, ECTL*, and CTL*. In *PROCOMET 94: Programming Concepts, Methods, and Calculi*, pages 561–581. Elsevier Science Publishers, 1994.
11. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV 00: Computer-Aided Verification*, LNCS 1633, pages 160–171. Springer-Verlag, 1999.
12. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, March 2000.
13. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
14. M. Huth, R. Jagadeesan, and D. A. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In *ESOP 01: European Symposium on Programming*. Springer-Verlag, 2001. To appear.
15. R. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.
16. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design Volume*, 6(1):11–44, 1995.
17. A. Podelski. Model checking as constraint solving. In *SAS 00: Static Analysis*, LNCS 1824, pages 221–37. Springer-Verlag, 2000.
18. T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November-December 1998.
19. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.
20. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.
21. D. Schmidt. Data flow analysis is model checking of abstract interpretation. In *POPL 98: Principles of Programming Languages*, pages 38–48. ACM, 1998.
22. M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.
23. B. Steffen. Data flow analysis as model checking. In *TACS 91: Theoretical Aspects of Computer Science*, LNCS 536, pages 346–365. Springer-Verlag, 1991.