Boolean and Cartesian Abstraction for Model Checking C Programs

Thomas Ball
Microsoft Research
Microsoft Corp.
tball@microsoft.com

Andreas Podelski[1]
Max Plank Institute
Saarbrücken, Germany
podelski@mpi-sb.mpg.de

Sriram K. Rajamani
Microsoft Research
Microsoft Corp.
sriram@microsoft.com

Abstract. The problem of model checking a specification in form of a C program with recursive procedures and many thousands of lines of code has not been addressed before. In this paper, we show how we attack this problem using an abstraction that is formalized with the Cartesian abstraction. 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. [5, 12, 19, 8, 20, 17, 13]. The idea of Boolean abstraction is to map 'concrete' states to 'abstract' states according their evaluation under a finite set of predicates ("Boolean expressions"). 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 invariant) is obtained through a fixpoint of the 'abstract' post operator.

We build upon the work in Boolean abstraction (in particular on the work of Graf and Saidi [17]) and propose a new abstraction function ("$\alpha_{\mathsf{b\cdot c}}$"). The new abstraction is also induced by predicates over states but it cannot be defined by a mapping over states. We use the framework of *abstract interpretation* [10] to formally specify our abstraction compositionally, by juxtaposing the Boolean abstraction with the *Cartesian abstraction.* The Cartesian abstraction formalizes the idea of ignoring dependencies between components of tuples. That is, it is used to approximate a set of tuples by the smallest Cartesian product containing this set. For example, the Cartesian abstraction of the set of tuples $\{\langle 0,1\rangle,\langle 1,0\rangle\}$ is $\langle *,*\rangle$, which represents the set $\{\langle 0,0\rangle,\langle 0,1\rangle,\langle 1,0\rangle,\langle 1,1\rangle\}$ ($*$ is the "don't care" value). The Cartesian abstraction underlies the *attribute independence* in certain kinds of program analysis [11].

The specification of the new abstraction through a Galois connection (obtained by composing the 'Boolean' and 'Cartesian' Galois connections), defines an 'ideal' abstract post operator ("$\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$"). We present a new algorithm for computing $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$, prove it correct and compare its performance with another algorithm that can be inferred directly from the work of [17]. We introduce three *refinements* of $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ ('control points', 'disjunctive completion' and 'focus'). This opens a space of design choices for abstract post operators that are defined and implemented using $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$. We have an implementation of two tools `c2bp` and `bebop` in which we combine all three refinements.

[1] This work performed while on leave at Microsoft Research.

The use of Cartesian abstraction allows us to represent the abstract post operator of a C program in form of a *Boolean program.* A Boolean program is a C program in which all variables have 'Boolean' type; more precisely, they can take the values 1 or 0 or the "don't care" value $*$. Boolean programs may have procedures with call-by-value parameter passing, local variables, and recursion – we therefore say that they have *infinite control.*

The SLAM Project. We next explain the specific context of our work. The SLAM project at Microsoft Research is an effort to build processes and tools for checking temporal properties of system software (such as device drivers and operating system components) written in common programming languages (such as C). We focus on invariant checking, to which model checking of safety properties can be reduced.

Although model checking has been effective in the domains of hardware circuits and network protocols, model checking of software introduces some new challenges. Even for sequential programs, the existence of both *infinite control and infinite data* 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. Model checking infinite control (in the form of pushdown automata) has been extensively studied (see [27, 3, 15, 14]). Model checking unbounded arithmetic data has been studied in model checking for protocols, parameterized systems or timed and hybrid systems [18, 1]. However, the combination of unbounded stack-based control and unbounded data has not been handled before. [2] We believe that this is a fundamental problem in model checking of programs written in C or other general-purpose programming languages.

The SLAM project addresses this fundamental problem through a separation of concerns that firsts abstracts infinite data domains through Cartesian and Boolean abstractions, and then uses well-known techniques to analyze the resultant Boolean program model, which has infinite control (but 'finite data'). That is, the data abstraction is induced by a set $\mathcal{P}$ of *predicates* that are evaluated according to the data values of the program variables, e.g. p_1 defined by $(x > 5)$ and p_2 defined by $(x == *p)$ where x and p are program variables.

Our working hypothesis is that for many interesting temporal properties of real-life system software, we can find suitable predicates such that the Boolean program model arising from the predicate abstraction is precise enough to prove the desired invariant. Further, abstractions induced by predicates seem attractive since refinement is done in a mechanical way when new predicates are added.

Given an invariant Inv to check on a C program, the SLAM process has three phases, starting with an initial set of predicates $\mathcal{P}$ (e.g. for expressing only the value of the program counter) 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.

[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 [9].

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 analysis [25, 22], and abstraction refinement [4, 19]). For more detail on Phase (2), see [2].

In summary, 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).
- Because of our separation of concerns, 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.

Related Work. The work of Graf and Saidi [17] is very close to ours and in part inspired it. Their abstraction scheme (Section 2.2 in [17]) does not use the Cartesian abstraction to the domain of trivectors. Instead, they define an abstraction α' as an approximation of the Boolean abstraction (on the same domain); the abstraction function α' in [17] does not form a Galois connection with the meaning function. Since that setting accounts for the same degree of approximation (loss of precision) as our setting, their procedure for computing the abstract post operator can be carried over to our setting, in principle. Our specific context (see above) prevents this practically.

The procedure of [17] goes through *all* elements in a subset of the domain of the abstract post operator and explicitly computes each image. This is feasible only if the subset is small. In the context of [17] (which does not address the issue of recursive procedures), the subset can be chosen as the set of all reachable 'abstract states' (bitvectors, essentially).

As explained above, the restriction of the abstract post operator to a subdomain is not possible in our interprocedural setting. Thus, the procedure of [17] would require a fixed number $2^n \cdot 2 \cdot n$ of calls to a theorem prover (n is the number of predicates inducing the abstraction; in our example application, $n = 35$). This means that the procedure of [17] is practically not applicable in our context. In this paper, we give a procedure with $O(2^n) \cdot 2 \cdot n$ calls, i.e. where the fixed (or best-case) factor 2^n is replaced by a worst-case factor $O(2^n)$, which makes all the difference for practical concerns (in our setting).

Outline. Section 2 explains the SLAM tools `c2bp` and `bebop` using a simple example. Section 3 fixes the notation. Sections 4 and 5 introduce Boolean and Cartesian abstractions, and Section 6 defines the abstract post operator $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$. Section 7 contains the algorithm implemented by the tool `c2bp`, and shows that the abstract post operator computed by `c2bp` is indeed $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$. Section 8 studies loss of precision under Cartesian abstraction, and Section 9 discusses refinements to minimize this loss. Particular implementation choices of these refinements leads to the abstract operator $\mathsf{post}^{\#}_{\mathsf{slam}}$ implemented by `bebop`. Section 10 gives the current implementation status of the SLAM tools and concludes the paper.

```
    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 (*)
           begin
[5]          b2 := H(0,b2);
[6]          b1 := 0;
           end
[7]      while(b2)
[8]      if (!b1)
[9]        assert(0);
       end

       boolean H(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)`

2 Example C Program

In this paper, we are concerned with the 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, and (2) `bebop`, a model checker for Boolean programs. 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, 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 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 section, we fix a finite set $\mathcal{P}$ of state predicates ("Boolean expressions"),

$$\mathcal{P} = \{p_1, \ldots, p_n\}.$$

A predicate p_i denotes the subset of states that satisfy the predicate, $\{s \in \mathsf{States} \mid s \models p_i\}$.

The set $\mathcal{P}$ of state predicates directly defines (1) an 'abstract transition system' and (2) an abstraction of the operator post. The respective conservative approximations of the set of reachable states coincide in the two cases. We briefly explain (1) for intuition; we need to build up on (2) in the next sections.

Abstract Transition System. The set $\mathcal{P}$ introduces: an equivalence relation over states, a partition of the state space, a homomorphism and a new 'abstract' transition system that is the quotient of the 'concrete' transition system (all these objects can be defined in terms of each other). More precisely, two states in States are *equivalent* if they satisfy the same set of predicates in $\mathcal{P}$. The equivalence classes form a *partitioning* of the state space; each class is characterized by which of the n predicates hold and which don't. An equivalence class may therefore be represented by a bitvector (of length n). One might call this bitvector an *abstract state*. We can define a *homomorphism* that maps every state s to the bitvector v_s for its equivalence class; each transition $s \longrightarrow s'$ in the concrete transition system is 'homomorphically' mapped to the transition $v_s \longrightarrow v_{s'}$ in the abstract transition system. The "meaning" of that transition is the set of transitions between all pairs of states in the corresponding equivalence classes; i.e., the approximation consists of adding all those transitions to the original transition system. The abstract transition system simulates (but in general does not bisimulate) the concrete one. Issues like preservance of bisimulation or temporal properties under the Boolean abstraction have been studied thoroughly; see e.g. [5, 12, 8, 20].

Abstract post *operator.* Since we are interested in invariant checking (and not at more general temporal properties), we are concerned with the abstraction of the post operator (and not with the construction of an abstract transition system as the target of a model checking procedure). The framework of abstract interpretation [10] yields the systematic construction of an abstract post operator.

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 formalized through an abstract domain and the meaning γ_{bool} and abstraction α_{bool}, two functions that we define below; 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. This again defines a specific approximation of the set of all reachable states, namely γ_{bool} applied to the fixpoint closure of $\mathsf{post}^{\#}_{\mathsf{bool}}$ on init.

Given $\mathcal{P}$, the *abstract domain* $\mathsf{AbsDom}_{\mathsf{bool}}$ is 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,

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

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

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

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., $f \circ g(x) = f(g(x))$.

The least fixpoint of $\mathsf{post}^{\#}_{\mathsf{bool}}$ that contains the abstraction of the set of initial states, which we can also write as ${\mathsf{post}^{\#}_{\mathsf{bool}}}^{\star}(\alpha_{\mathsf{bool}}(\mathsf{init}))$, is exactly the set of reachable states of the abstract transition system defined above. Its meaning, i.e. the set of states $\gamma_{\mathsf{bool}}({\mathsf{post}^{\#}_{\mathsf{bool}}}^{\star}(\alpha_{\mathsf{bool}}(\mathsf{init})))$ is an invariant of the (concrete) program.

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 $i = 1, \ldots, n$,

$$v_i = \begin{cases} 0 \text{ if } \Pi_i(V) = \{0\} \\ 1 \text{ if } \Pi_i(V) = \{1\} \\ * \text{ if } \Pi_i(V) = \{0, 1\}. \end{cases}$$

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 bitvalues 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 $\mathbf{post}^{\#}_{\mathsf{b \cdot c}}$ 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 implicitely defines the invariant Inv_1 given by $\mathcal{I}_1$ in [17]; 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

$$\mathsf{Inv}_1 = \gamma_{\mathsf{b\cdot c}}({\mathsf{post}^{\#}_{\mathsf{b\cdot c}}}^{\star}(\alpha_{\mathsf{b\cdot c}}(\mathsf{init}))).$$

The invariant Inv_1 is represented abstractly by one trivector, i.e. it is the Cartesian product of sets each desribed 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 $\mathbf{post}^{\#}_{\mathsf{b\cdot c}}$

Given the transition system (defining the operators post and $\widetilde{\mathsf{pre}}$) and the set of n predicates $\mathcal{P}$, the tool c2bp produces a Boolean program over n 'Boolean' variables $v_1, \ldots, v_n$. Each statement of the C program corresponds to 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 an operator $\mathsf{post}^{\#}_{\mathsf{c2bp}}$ over trivectors defined in the following way.

$$\mathsf{post}^{\#}_{\mathsf{c2bp}}(\langle v_1, \ldots, v_n \rangle) = \langle v'_1, \ldots, v'_n \rangle \quad \text{if} \quad v'_1 = e_1[v_1, \ldots, v_n], \ldots, v'_n = e_n[v_1, \ldots, v_n]$$

In this section, we present the expressions $e_i[v_1, \ldots, v_n]$ that are computed by c2bp. We then prove that the tool c2bp is *correct*, meaning that the operator $\mathsf{post}^{\#}_{\mathsf{c2bp}}$ is exactly the operator $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ (the operator $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$ is the *specification* of the tool c2bp).

The expression e_i over the variables $v_1, \ldots, v_n$ that defines the i-th value of the successor trivector of the Boolean program is

$$e_i = \mathsf{H}(e_i(1), e_i(0)).$$

Here, the function H applied to two Boolean expressions e and e' over the variables $v_1, \ldots, v_n$ yields a third expression $\mathsf{H}(e, e')$ over the variables $v_1, \ldots, v_n$ that evaluates as follows:

$$\mathsf{H}(e[v_1, \ldots, v_n], e'[v_1, \ldots, v_n]) = \begin{cases} 1 \text{ if } \langle v_1, \ldots, v_n \rangle \models e \\ 0 \text{ if } \langle v_1, \ldots, v_n \rangle \models e' \\ * \text{ if neither} \end{cases}$$

The satisfaction of a Boolean expression e by a trivector $\langle v_1, \ldots, v_n \rangle$ is defined as one expects, namely $\langle v_1, \ldots, v_n \rangle \models e$ if all bitvectors in $\gamma_{\mathsf{bool}}(\langle v_1, \ldots, v_n \rangle)$ 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 two Boolean expressions $e_i(1)$ and $e_i(0)$ over the variables $v_1, \ldots, v_n$ are obtained by direct correspondence from the two Boolean expressions $E_i(0)$ and $E_i(1)$ over the predicates $p_1, \ldots, p_n$ that are computed, in the form of disjunctions of conjunctions of possibly negated predicates, according to the following definition.

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

Here, the function F assigns each set of states S a representation of its Boolean *under*-approximation, i.e. 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 $\mathsf{States} - (\gamma_{\mathsf{bool}} \circ \alpha_{\mathsf{bool}})(\mathsf{States} - S)$. The ordering $e < e'$ on Boolean expressions is such that each disjunct of e implies some disjunct of e' (e.g., p_1 is greater than $p_1 \wedge p_2 \vee p_1 \wedge \neg p_2$).

Proposition 1. *The operator* $\mathsf{post}^{\#}_{\mathsf{c2bp}}$ *(represented by the Boolean program computed by* $\mathsf{c2bp}$*) is exactly the operator* $\mathsf{post}^{\#}_{\mathsf{b{\cdot}c}}$*, the Cartesian of the Boolean abstraction of the operator* post*.*

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

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

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

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

Now, we can represent the abstract post operator $\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}}$ 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 ${\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}}}^{(i)}$.

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

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.

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

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

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

then

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

By definition of $\alpha^{(i)}_{\mathsf{b}\cdot\mathsf{c}}$, 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 $e_i(0)$.

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

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. $\square$

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).

In the worst case, computing $\mathsf{F}(S)$ may require exponentially many calls to the theorem prover. In practice (for the reasons above), we find that we can compute $\mathsf{F}(S)$ with far fewer number of theorem prover calls.

By restricting the size of the sets I to at most 3, we obtain a sound approximation. In practice, in all examples we have seen so far, this restriction (which reduces the worst case to a cubic number of calls to the theorem prover) does not lose any precision. Consequently, `c2bp` (with or without the restriction) is able to handle C programs with several hundred lines and about 35 predicates in a few minutes.

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 [10, 16]). Given a concrete and an abstract domain, an abstraction α and a meaning γ, we say that the operator F does not loose precision under the abstraction to $F^\#$ [on the abstract value a] if $\gamma \circ F^\# = F \circ \gamma$ [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 Cartesian abstraction does not loose precision' is short for '$\mathsf{post}^\#_{\mathsf{bool}}$ does not loose precision under the abstraction to $\mathsf{post}^\#_{\mathsf{b\cdot c}}$'. We define function F to be *deterministic* if it maps singleton sets to singleton sets. It will become useful to establish the following observation.

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

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 $\langle *,*,0\rangle$,

$$\mathsf{post}^{\#}_{\mathsf{b\cdot c}}(\langle 0,0,0\rangle) = \langle *,*,0\rangle$$

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 looses 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 [17].) Note that the concrete operator post is *deterministic*; the loss of precision in the Cartesion 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 loose 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 assertion $x_1 = x_2 \wedge x_1' = x_1 \wedge x_2' = x_2$ (corresponding to a 'filter' that enforces a condition, here the equality between the two variables). 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.

In this section, the formalization of the Cartesian abstraction has proven useful in analyzing the loss of precision incurred by replacing the 'classical' Boolean abstract post operator $\mathsf{post}^{\#}_{\mathsf{bool}}$ with the 'more efficient' operator $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$. It may also be useful for determining general conditions ensuring that no proper loss of precision occurs, phrased e.g. in terms of separability [21]; we leave this for further work.

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

Our general methodology is to start with a coarse approximation of the post operator and to refine it gradually, with the goal of a yes/no answer to the model checking problem. We distinguish two kinds of refinement according to the two abstractions defining $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}$. In this section, we apply standard methods from program analysis (see [10]) to a new setting (of 'abstract model checking') 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 implicitely 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. $\boldsymbol{S}[\ell] \in 2^{\mathsf{States}}$. 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 $(\mathsf{AbsDom}_{\mathsf{cartesian}})^{\mathsf{Loc}}$.

Note that we don't model the procedure stack associated with the state. This is because, the stack is implicity present in the semantics of the Boolean program, and hence does not need to be abstracted by `c2bp`. All that `c2bp` needs to do in the presence of procedure calls is to model how the actual parameters relate to the formal parameters, and how the return value of the procedure relates to the return value at the caller. These are handled essentially in the same way as assignment statements.

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 post operator, $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}[\ell](\langle \dots, v[\ell_1], \dots, v[\ell_2], \dots \rangle) = v[\ell_1] \sqcup v[\ell_2]$ 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 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 (which corresponds to an assertion where primed variables only appear in the form $x' = x$), then there is a loss of precision exactly if ϕ expresses a dependence between variables (such as $x = y$ as in Example 2); since the operator $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}[\ell]$ is deterministic, Proposition 2 applies. If the edge corresponds to an update statement, then (and only then) the operator $\mathsf{post}^{\#}_{\mathsf{b\cdot c}}[\ell]$ may not be deterministic (even if the concrete operator $\mathsf{post}[\ell]$ is deterministic).

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 [10], 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 [23]). This operation can be used to eliminate all loss of precision under Cartesian abstraction except for post operators $\mathsf{post}[\ell]$ at locations ℓ with a *nondeterministic* statement (on the incoming edge). Examples of such nondeterministic statements are given 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.,

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

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 $\mathsf{post}^{\#}_{\mathsf{b \cdot c \cdot \vee \cdot}[1,2]}$ as follows (refining the operator $\mathsf{post}^{\#}_{\mathsf{b \cdot c}}$ given in the previous section).

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

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

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 loose precision under the 'focussed' Cartesian abstraction.*

The abstract post operator $\mathsf{post}^{\#}_{\mathsf{slam}}$ that is used in SLAM results from combining the three refinements presented in Sections 9.1, 9.2 and 9.3, with the *total* focus operation $\mathsf{focus}[1, 2, \ldots, n]$ in each component. For each control point ℓ in the program, we have:

$$\mathsf{post}^{\#}_{\mathsf{slam}}[\ell] = \mathsf{post}^{\#}_{\mathsf{b \cdot c \cdot \vee \cdot}[1,\ldots,n]}[\ell].$$

bebop is a symbolic model checker. It implements the disjunctive completion and the total focus operation symbolically.

10 Conclusion

Abstraction is probably the single most important issue in model checking software. The SLAM project at Microsoft Research is an effort to check temporal safety properties of software, by automatically constructing abstractions of the software, and doing model checking on the abstractions. This paper formally describes the abstraction implemented by the SLAM tools `c2bp` and `bebop`. The technical contributions are the formalization of the abstract post operator $\mathsf{post}^{\#}_{\mathsf{b}\cdot\mathsf{c}}$ in terms of Boolean and Cartesian abstraction, and an algorithm (for computing this operator) that is practical in our setting which addresses programs with recursive procedures. The formal machinery developed here has potentially other applications in designing new abstractions for model checking software, and also in classifying data-flow analysis problems in the spirit of [26, 24].

We have implemented both the tools `c2bp` and `bebop`. `c2bp` is written in OCAML and interfaces with existing theorem provers to compute the F function. It also uses inputs from a flow-insensitive points-to-analysis, in order to compute $\widetilde{\mathsf{pre}}$ in the presence of pointers and aliasing. `bebop` is written in C++ and interfaces with existing BDD packages. We have managed to use `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, and a particular property we checked requires analyzing about 350 lines of C code. We introduced 35 predicates manually, and c2bp was able to process 350 lines of C code with about 35 predicates in under a minute, and generate a Boolean program. `bebop` was able to check the property on this Boolean program in a few seconds. All tools were run on an 800MHz Pentium III PC with 512MB of RAM. More details and a case study on using SLAM tools to check properties of Windows NT device drivers will appear in a forthcoming paper.

Acknowledgement

We thank Todd Millstein and Rupak Majumdar for their work in the `c2bp` implementation. We thank Bertrand Jeannet and Laurent Mauborgne for helpful comments on preliminary versions of this paper.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 00: SPIN Workshop*, Lecture Notes in Computer Science 1885, pages 113–130. Springer-Verlag, 2000.
3. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97: Concurrency theory*, volume 1243 of *Lecture Notes in Computer Science (LNCS)*, pages 135–150. Springer-Verlag, 1997.
4. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, Lecture Notes in Computer Science 1855, pages 154–169. Springer-Verlag, 2000.
5. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 343–354. ACM Press, 1992.
6. E. M. Clarke. Synthesis of resource invariants for concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(3):338–358, 1980.

7. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *In Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 52–71.
8. R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in abstractions of model checking. In *Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science (LNCS)*, pages 51–63. Springer-Verlag, 1995.
9. 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*, 2000.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages*. ACM Press, 1977.
11. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA: Functional Programming and Computer Architecture*, pages 170–181. ACM Press, 1995.
12. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: abstractions preserving ACTL*, ECTL*, and CTL*. In E.-R. Olderog, editor, *IFIP Working Conference on Programming Concepts, Methods, and Calculi*. Elsevier Science Publishers, 1994.
13. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *11th International Conference on Computer-Aided Verification*. Springer-Verlag, July 1999.
14. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. Technical Report Technical Report TUM-I0002, SFB-Bericht 342/1/00 A, Technische Universitat Munchen, Institut fur Informatik, February 2000.
15. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *INFINITY' 97: Verification of Infinite-state Systems*, July 1997.
16. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, March 2000.
17. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, 1997.
18. T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: the next generation. In *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 56–65. IEEE Computer Society Press, 1995.
19. R. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.
20. 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, Issue 1*, 1995.
21. T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November-December 1998.
22. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 49–61, January 1995.
23. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *PLDI: Programming Language Design and Implementation*, pages 105–118, 1998.
24. D. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proceedings of the Twenty Fifth Annual Symposium on Principles of Programming Languages*, pages 38–48. ACM Press, 1998.
25. M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.
26. B. Steffen. Data flow analysis as model checking. In *Theoretical Aspects of Computer Science (TACS'91)*, volume 536, Sendai (Japan), September 1991.
27. B. Steffen and O. Burkart. Model checking for context-free processes. In *CONCUR'92, Stony Brook (NY)*, volume 630 of *Lecture Notes in Computer Science (LNCS)*, pages 123–137, Heidelberg, Germany, 1992. Springer-Verlag.