Summarizations for Symbolic Executions

JOXAN JAFFAR, ANDREW E. SANTOSA and RĂZVAN VOICU

School of Computing
National University of Singapore
Republic of Singapore
{joxan,andrews,razvan}@comp.nus.edu.sg

Abstract

Consideration of execution paths is basic in program analysis and verification because it represents the process of exact propagation through the program fragment at hand. This is challenged by the fact that there are exponentially many paths in general. In this paper, we consider a straight-line program fragment annotated with initial and final assertions, and present an optimization technique for this problem based on the use of *summarizations*: partial descriptions of the input-output behavior of a program fragment. The main advantage of a summarization is that, during program reasoning, invocations of the program fragment using different contexts may be handled by the one summarization.

The key idea is a method for deriving summarizations dynamically. Traditionally, summarizations are sought for predefined procedures. In contrast, we compute, opportunistically and on-the-fly, summarizations of arbitrary program fragments. The objective is to to discover its most general form, and to discover its most specific conclusion. This enhances the likelihood that the summarization is both frequently applicable and effective in future.

1. Introduction

We consider the problem of exact propagation through a loop-free program fragment with respect to given initial and final assertions. This is tantamount to examining the execution paths of the program fragment. Traditionally, a final assertion is a predicate on variable values at the end of the program fragment. The objective then is a *safety* proof of the final assertion. The main challenge toward this goal is that there is in general a large number of possible execution paths of the program fragment that need be considered. A main advantage, however, is that proofs of larger programs can be composed of smaller safety proofs once they are obtained[1]. In other words, the final assertion serves to culminate the proof of one program fragment into an assertion to be used for the proof of another fragment.

In this paper, we start by a generalization of the problem: given an initial assertion for a program fragment, *discover* what the final assertion is. Typically, the final assertion is an abstract description of the input-output behavior of the program fragment. For example, we could discover a bound for a given variable, or, in the tradition of predicate abstraction [5], which finite subset of a universe of predicates hold. In general, the abstract formulation of the final assertion also serves, as in the traditional case of a safety proof, to culminate the proof of a program fragment into a simple assertion that can then be used in a bigger proof.

The key to our algorithm is the use of *summarizations*. A summarization is a partial description of the input-output behavior of a program fragment. A program fragment may be executed under different *contexts*, that is, the program state under which the fragment's execution begins. This is because the start of the fragment may be reached via various execution paths. A key advantage of a summarization is that, during analysis, invocations of the program fragment using different contexts may be handled by the one summarization, without the need to re-analyze the program fragment for the various contexts. The main challenge in obtaining summarizations, however, is twofold: to discover its most general form, and to discover its most specific conclusion.

In this paper, we present a method for deriving summarizations dynamically. Traditionally, summarizations are sought for predefined procedures. In contrast, our method computes, opportunistically and on-the-fly, expressive summarizations of arbitrary program fragments. Our method can thus be used to augment an existing analysis algorithm, which may or may not use a given abstraction function, to perform a symbolic exploration of the state space with greater efficiency.

As an illustration, consider a program P_0 with the objective to summarize a certain program fragment P of P_0. The fragment P is in fact invoked several times with different contexts when executing P_0. As a trivial example, suppose the program fragment P were `if (x == 0) count++`, and that we were just interested in the variable `count`. A summarization of P would state that the final value $count'$ of `count`, is equal to $count + 1$ in case the context implied $x = 0$; otherwise, the summarization would state that $count' = count$. However, it is often the case that the number of contexts that need be considered is much smaller. Consider generalizing the above example to a series of n if-statements. Then, even if the number of contexts arising from executing the fragment is 2^n, there are in fact only n possible outcomes for the final value $count'$. Continuing this example, suppose that we were interested in just the bound for `count`, as opposed to its precise value. The summarization could then say that `count` is incremented by at most 2, *regardless* of the context. In general, the challenge is to coordinate the hypothesis and the conclusion of a summarization so that it is just expressive enough to obtain the final discovery.

After presenting the main ideas informally in the next section, we will present a framework for representing computation trees of symbolic traces. Each trace contains a set of constraints representing both the history and the set of program states under consideration. An important property of the framework is that it is compo-

[1] For loops, we of course also need to deal with invariants.

sitional, crucial to the reasoning about the effects of contexts on program fragments. We then present the main algorithm which is based on the operation of constraint deletion. We finally demonstrate the algorithm on several benchmark examples.

1.1 Related Work

There have been approaches in using summarization for program analysis. In this section we discuss their representatives.

Summarization is typically applied to syntactically identifiable program fragments, such as blocks, procedures or transactions in the presence of concurrency. An example for procedures can be found in [3], which presents an approximation method for the semantics of recursive procedures. Similarly, [18] presents approaches to interprocedural dataflow analysis of sequential programs, where one is based on summarizing each procedure into an input-output relation, which, in the process of program analysis, can replace the procedure at each call location. Similar to our approach, in [16], summarization is performed on the fly. However, this work only summarizes procedures, not arbitrary program fragments, as input-output functions. By focusing only on dataflow analysis problems (which in a sense defines a finite abstraction), it provides a polynomial-time analysis algorithm. The paper [1] concerns predicate-abstraction, that is, procedures are abstracted with a predicate which represents input-output relations of the function. In [15], a method for summarizing procedures in concurrent programs is presented. What is actually summarized is a *transaction*, a sequence of statements of a thread which consists of, for example, a sequence of lock acquisitions, shared data updates and lock releases, and hence can be treated as atomic. A transaction may span across a procedure boundary. A summarization already created can be reused whenever the thread is to invoke the transaction.

In summary, our approach differs mainly in that summarizations are computed opportunistically and on the fly, for arbitrary program fragments. A key feature is that our summarization is exact in the sense it describes the strongest postcondition of the fragment at hand.

Our work is also related to the areas of program verification, model checking and theorem proving. There are two main similarities. One is that we employ search. Recently, an important alternative to proving safety of programs is to translate the verification problem into a boolean formula that can then be subjected to a SAT [2] or SMT [11] solver At the heart of these solvers is a DPLL-based algorithm for traversing the solution space. In our work, we use a symbolic transition system which generates formulas on-the-fly in the search for solutions.

The second similarity is in the way we generalize a formula while preserving certain critical properties. This is analogous to the notion of *Craig interpolation*, which generally seeks the most liberal hypothesis for a given conclusion. The use of such interpolants is now an important component of model checkers [13] and theorem-proving [14]. In fact, our algorithm generalizes a formula in case the new formula can be shown to have a similar search profile to a previous search. In more detail, our formulation shares similarity with the specific method of generalization of finding a minimal unsatisfiable subset of an unsatisfiable formula. See eg. [4].

2. The Basic Idea

In this section we illustrate, by means of a simple example, the principles of our optimization technique.

Consider the program fragment P

```
⟨0⟩ if (b1) x := x + 1 else x := x + 2
⟨1⟩ if (b2) x := x + 3 else x := x + 4 ⟨2⟩
```

and suppose we are interested in producing a summarization of this fragment, which is an abstract input-output description. An analysis procedure would then be able to use such a description directly, instead of traversing the program fragment to compute an analysis for program point ⟨2⟩. Since in the analysis process the need to traverse P may arise multiple times, the availability of a summarization would make the process more efficient.

A summarization can be produced with a variety of purposes in mind. The first, and most straightforward, is proving a safety property, which translates into checking whether the program fragment's output state satisfies a given abstraction across all its (abstract) input contexts. A second, and more sophisticated purpose is the discovery of an abstract property over a (possibly infinite) abstract domain. Such a summarization would be very useful, for instance, in computing worst case execution times of program fragments. We shall illustrate both these approaches in the remainder of this section. To that end, let us first introduce the required terminology. We shall symbolically describe a set of states in the form

$$(k,\tilde{x}),\Psi$$

where k denotes the program counter, the sequence $\tilde{x}$ denote program variables[2], and Ψ is a *constraint* or conjunction of constraints on these variables. Call such an expression $(k,\tilde{x}),\Psi$ a (symbolic) program *state*. The semantics of such a state is simply the set of instances of $(k,\tilde{x})$ for which the constraint Ψ is *true*. In this example, $\tilde{x}$ contains x, and possibly other variables, and the expressions b_1 and b_2 are some unspecified boolean conditions unconnected to x. Where $\mathcal{G}$ is a state $(k,\tilde{x}),\Psi$, we write $cons(\mathcal{G})$ to denote the constraints Ψ in $\mathcal{G}$.

A notion of "strongest postcondition" can now be obtained. Consider an initial state

$$\mathcal{G} \equiv (0,x_0), x_0 = 0$$

where x_0 is indicates the initial value of program variable `x` (zero, in this case). This state can be propagated through the `then` branch of the first statement yielding

$$\mathcal{G}_0 \equiv (1,x_1), x_0 = 0, b_1, x_1 = x_0 + 1$$

and a final propagation through the second `then` branch yields

$$\mathcal{G}_{00} \equiv (2,x_2), x_0 = 0, b_1, x_1 = x_0 + 1, b_2, x_2 = x_1 + 3$$

Note that, since the value of program variable `x` has changed throughout the propagations, the symbol x_0 in $\mathcal{G}$ has been replaced by x_1 in $\mathcal{G}_0$, and the symbol x_1 in $\mathcal{G}_0$ has in turn been replaced by x_2 in $\mathcal{G}_{00}$. Since this state $\mathcal{G}_{00}$ is final, we shall call x_2 a "final" variable. The symbols x_0 and x_1 that appear in $\mathcal{G}_{00}$ still refer to the value of `x` at program points 0 and 1, and become *auxiliary* variables in the state $\mathcal{G}_{00}$.

The propagation operation is defined on all edges of the program's control flow graph. For instance, we could propagate $\mathcal{G}$ along the `else` branch of the first statement to produce a state $\mathcal{G}_1$. Moreover, we could regard $\mathcal{G}_0$ and $\mathcal{G}_1$ as children of $\mathcal{G}$ in a *computation tree* of $\mathcal{G}$ w.r.t. the program P. Further strongest postcondition propagation steps can be applied to $\mathcal{G}_0$ and $\mathcal{G}_1$ resulting in the states $\mathcal{G}_{00}$, $\mathcal{G}_{01}$, $\mathcal{G}_{10}$, and $\mathcal{G}_{11}$. This computation tree is depicted in Figure 1.

The process of building such a tree proceeds depth-first, producing the first sequence of states $\mathcal{G}$, $\mathcal{G}_0$, $\mathcal{G}_{00}$. The last state cannot be further propagated, and since the constraints attached to this state

[2] We use the term *program* or *system variable* to refer to variables that appear in imperative programs and whose values may change throughout a program's execution. We use the term *variable* to refer to logic variables, which are interpreted as placeholders for values. A (logic) variable may be used to represent the value of a program variable at a specific program point.

$$
\begin{array}{l}
\mathcal{G} \equiv (0,x_0), x_0 = 0 \\
\quad \mathcal{G}_0 \equiv (1,x_1), x_0 = 0, b_1, x_1 = x_0 + 1 \\
\quad\quad \mathcal{G}_{00} \equiv (2,x_2), x_0 = 0, b_1, x_1 = x_0 + 1, b_2, x_2 = x_1 + 3 \\
\quad\quad \mathcal{G}_{01} \equiv (2,x_2), x_0 = 0, b_1, x_1 = x_0 + 1, \neg b_2, x_2 = x_1 + 4 \\
\quad \mathcal{G}_1 \equiv (1,x_1), x_0 = 0, \neg b_1, x_1 = x_0 + 2 \\
\quad\quad \mathcal{G}_{10} \equiv (2,x_2), x_0 = 0, \neg b_1, x_1 = x_0 + 2, b_2, x_2 = x_1 + 3 \\
\quad\quad \mathcal{G}_{11} \equiv (2,x_2), x_0 = 0, \neg b_1, x_1 = x_0 + 2, \neg b_2, x_2 = x_1 + 4
\end{array}
$$

Figure 1. Computation Tree

$$
\begin{array}{l}
\overline{\mathcal{G}}_0 \equiv (1,x_1), x_1 \leq x_0 + 2 \\
\quad \overline{\mathcal{G}}_{00} \equiv (2,x_2), x_1 \leq x_0 + 2, b_2, x_2 = x_1 + 3 \\
\quad \overline{\mathcal{G}}_{01} \equiv (2,x_2), x_1 \leq x_0 + 2, \neg b_2, x_2 = x_1 + 4
\end{array}
$$

Figure 2. Generalized Computation Tree

are satisfiable, we call this node of the tree *successful*. Hence, the sequence of states $\mathcal{G}$, $\mathcal{G}_0$, $\mathcal{G}_{00}$ (the "leftmost" path) represents a feasible computation path, and each solution to the state $\mathcal{G}_{00}$ is a tuple of legal values for the corresponding program variables at program point 2.

The depth-first traversal continues with propagating $\mathcal{G}_0$ to program point 2, yielding the state $\mathcal{G}_{01}$. Suppose for a moment that the constraint $\neg b_1, \neg b_2$ is in fact unsatisfiable. Then the sequence of states $\mathcal{G}, \mathcal{G}_1, \mathcal{G}_{11}$ (the "rightmost" path) encodes an infeasible computation to the *false* state $\mathcal{G}_{11}$.

The computation tree depicted in Figure 1 provides an accurate description of program fragment P's behavior, and represents the backbone of our summarization technique.

We now illustrate the process of obtaining a summarization of P with the purpose of proving a safety property of the final states of the program. Such a property is encoded as a constraint in a given possibly infinite family of constraints. We shall call this family an *abstract domain*, and each of its constraints an *abstraction*, in order to distinguish them from regular constraints. The objective therefore is to determine if the final states of the program satisfy one abstraction.

Our algorithm shall, in the process of traversing the computation tree starting from the initial state, construct a *summarization* of the various states encountered. More specifically, a summarization Σ of a state $\mathcal{G} \equiv (i,x), \Psi$ is a pair, written $(\overline{\mathcal{G}} \mapsto \Psi')$. The first component $\overline{\mathcal{G}}$ is a generalization[3] of $\mathcal{G}$, and is called the *coverage* of Σ. The second component Ψ' is a constraint and is the *answer* of Σ. The meaning of a summarization Σ is that the final states $\overline{\mathcal{G}}$ are entailed by the answer Ψ'. A summarization of the initial state, therefore, provides the abstraction of the whole program.

An arbitrary state $\mathcal{G}$ is *summarized* by such a summarization Σ if the coverage of Σ is a generalization of $\mathcal{G}$. If so, the appropriate specialization of Ψ' toward $\mathcal{G}$ becomes an answer for $\mathcal{G}$. More specifically, suppose $\mathcal{G}$ is summarized by a summarization $\Sigma \equiv (\overline{\mathcal{G}} \mapsto \Psi')$. Then $\overline{\mathcal{G}}$ is a generalization of $\mathcal{G}$, say $\mathcal{G} \equiv \overline{\mathcal{G}} \wedge \theta$. Then the final states of $\mathcal{G}$ entail the constraint $cons(\overline{\mathcal{G}}) \wedge \theta \wedge \Psi'$.

We now provide two examples of the usage of summarization, one in proving safety, another in discovering safety.

[3] We formalize the notion of generalization in Section 3. Here it suffices to say that the set of program states represented symbolicly by $\mathcal{G}$ is a subset of that of $\overline{\mathcal{G}}$.

1. *Search Space Reduction in Proving Safety.* We now exemplify the process of obtaining a summarization of P with the purpose of proving that our example program satisfies the abstraction $\mathcal{A} \equiv x_2 - x_0 \leq 6$. In other words, we are deriving an abstraction on the amount a variable is *incremented*, rather than an abstraction of its maximum value. Later in this section, we will also show how such an upper bound on the variation of a variable can be *discovered*.

   Consider a generalization $\overline{\mathcal{G}}_0$ of $\mathcal{G}_0$ where we replace the constraints $x_0 = 0, b_1, x_1 = x_0 + 1$ in $\mathcal{G}$ to become $x_1 \leq x_0 + 2$. That is, we drop the requirement that the initial value of x is 0, we drop the constraint b_1, and finally we relax the constraint between x_2 and x_1 so that their difference is at most 2, as opposed to exactly 1.

   Consider now the computation tree of the new state $\overline{\mathcal{G}}_0$, depicted in Figure 2. Notice that the behavior of this tree is the same as the subtree for $\mathcal{G}_0$ *as far as the abstraction is concerned*. That is, the increment of x contained in both trees (4) is the same.

   We may thus produce a summarization

   $$\Sigma \equiv (\overline{\mathcal{G}}_0 \mapsto \mathcal{A} \equiv x_2 - x_0 \leq 6)$$

   Now, instead of traversing the second subtree $\mathcal{G}_1$ in Figure 1, we may simply apply this summarization to $\mathcal{G}_1$ hence reducing search effort. First we verify that Σ is indeed a summarization of $\mathcal{G}_1$, and this follows easily from the fact that the coverage $\overline{\mathcal{G}}_0$ of Σ is a generalization of $\mathcal{G}_1$. Then we add the constraints $\Psi \equiv x_0 = 0, b_1, x_1 = x_0 + 2$ of $\mathcal{G}_1$ to $\overline{\mathcal{G}}_0$ and conclude that the summarized answer is $\Psi \wedge \mathcal{A}$. This entails $\mathcal{A}$, and so we conclude that running $\mathcal{G}_1$ is safe. This demonstrates the essential usefulness of the summarization.

2. *Discovering safety.* We have shown that a summarization can express an abstraction of the amount the value of a variable varies across P. Next, we will show that the upper bound of such a variation can be in fact *discovered* in the summarization process. Consider the abstract domain $\{x_2 - x_0 \leq \alpha : 0 \leq \alpha \leq 6\}$. That is, each abstraction implies that the increment of x is bounded by a nonnegative number α less than or equal 6. The objective now is not just to prove that our program is safe, but also to *discover* one such value α. Using the same arguments above, we can obtain the same summarization $\Sigma \equiv (\overline{\mathcal{G}}_0 \mapsto x_2 - x_0 \leq 6)$, and therefore finally conclude that $\alpha = 6$.

   However, consider the new scenario in which the constraint $\neg b_1 \wedge \neg b_2$ is unsatisfiable. In this case, a more precise answer is in fact $\alpha = 5$. This is because the path $\mathcal{G}, \mathcal{G}_1, \mathcal{G}_{11}$, which would have produced the largest increment of x, is no longer feasible.

   The moral of the story here is that while the use of summarizations preserves safety, it does not preserve *precision*.

   In order to preserve precision, we require an additional condition for a state $\mathcal{G}$ to be summarized by some summarization $\Sigma \equiv (\overline{\mathcal{G}} \mapsto \Psi)$. Presently, it is a notion of coverage only. This in fact means that the successful paths of $\mathcal{G}$ are a subset of the successful paths of $\overline{\mathcal{G}}$. For example, in Figure 2, the more general state $\overline{\mathcal{G}}_0$ has a successful path to $\overline{\mathcal{G}}_{01}$. In contrast, the more specific summarized state $\mathcal{G}_1$, in Figure 1, does not have this corresponding path.

   What is needed for precision is that the state summarized has *exactly* the paths of the summarizing state[4].

[4] We show later that we just need *some* of these paths.

Continuing this example, if we now include this new condition, the state $\mathcal{G}_1$ is no longer summarized and its tree must be traversed in the usual manner.

We finally extend the abstract domain to an *infinite* number of abstractions. Consider now abstract the domain $\{\text{“}x_2 - x_0 \leq \alpha'' : 0 \leq \alpha\}$, where an abstraction states that the increment of x is bounded by a nonnegative number α which is arbitrarily large.

The question now is how to summarize the state $\mathcal{G}_0$? Previously, after we traversed the subtree of $\mathcal{G}_0$, we determined that its successful paths entail the abstraction $x_2 - x_0 \leq 6$. The number 6 arose from the previous definition of the abstract domain. In this example, the subtree of $\mathcal{G}_0$ can only reasonably answer that $x_2 - x_0 \leq 5$. While this is indeed a correct answer to a summarization of $\mathcal{G}_0$, the problem is that the coverage will be restricted to $x_1 - x_0 \leq 1$. That is, the proposed summarization would be:

$$((1, x_1), x_1 - x_0 \leq 1 \mapsto x_2 - x_0 \leq 5)$$

This would then not cover the subsequently encountered state $\mathcal{G}_1$, and therefore an opportunity for optimization would be lost.

The remedy here is not to summarize $\mathcal{G}_0$ with the specific answer $x_2 - x_0 \leq 5$, but instead to use a generalization of this such that *safety is preserved*. For example, we could use the answer $x_2 - x_0 \leq 500$, which implies that the coverage is $(1, x_1), x_1 - x_0 \leq 496$. Alternatively, we could use the answer $x_2 - x_0 \leq 5000$, which implies that the coverage is $(1, x_1), x_1 - x_0 \leq 4996$. In general, we should use the answer $x_2 - x_0 \leq y + 6, y \geq 0$ where y is a new variable. This way the coverage becomes $(1, x_1), x_1 - x_0 \leq y + 2, y \geq 0$. That is, our summarization is

$$((1, x_1), x_1 - x_0 \leq y + 2, y \geq 0 \mapsto x_2 - x_0 \leq y + 6, y \geq 0)$$

and this is in fact the best possible summarization for our purpose of discovering an upper bound for the increment of x. To concretize this example, consider a new state $\mathcal{G}'_1$ which has the constraint $x_1 = x_0 + 10$. Using the summarization above, we obtain that the value of y is 8, and therefore the answer to this new state is 14.

Note that this summarization is not the best for all purposes: if we were interested in a *lower* bound for the increment of x, this summarization is not appropriate.

In summary for this section, we have exemplified an algorithm for summarizing program fragments toward the goal of satisfying a given abstract domain in the pursuit of safety. A summarization provides an answer to a state that preserves safety, and does so with a covering state which is as general as possible.

In the more common setting of finite abstract domains, the answer of a summarization can be easily chosen as one abstraction. However, if we demanded that only the *best* abstraction be returned in case the program is safe, then additional conditions on the use of summarization is needed. Even so, we have found experimentally that these additional conditions are not severe.

In the case of an infinite abstract domain, the choice of answer to a summarization is manual. However, this task is not hard for the small number of interesting infinite domains that we have considered. In this section, we considered the abstract domain to reflect an upper bound of the difference between two program variables.

In what follows, we present an algorithm for the three scenarios:

- *Proving Safety*: is a given abstraction satisfied?
- *Discovering Safety*: given a possibly infinite set of abstractions, *find* one abstraction which is satisfied.
- *Discovering Exact Safety*: given a possibly infinite set of abstractions, find the “best” one which is satisfied. That is, a more specific abstraction would be unsafe.

Section 4 deals with the first two, while Section 5 deals with the latter.

3. Constraint Transition Systems

This section presents a formalization of what we consider to be a typical abstract interpretation framework, augmented with our summarization algorithm. The essence of this section formalizes the notion of a computation tree.

3.1 Preliminaries

We start by defining a language of first-order formulas. Let $\mathcal{V}$ denote an infinite set of variables, each of which has a type in the domains $\mathcal{D}_1, \cdots, \mathcal{D}_n$, let Σ denote a set of *functors*, and Π denote a set of *constraint symbols*. Functors represent program operations such as arithmetic operations and array assignments, while constraints represent conditionals in program statements such as arithmetic relations, in addition to equalities. There is a special collection of *final* variables. A *term*[5] is either a constant (0-ary functor) in Σ or of the form $f(t_1, \cdots, t_m)$, $m \geq 1$, where $f \in \Sigma$ and each t_i is a term, $1 \leq i \leq m$. A *primitive constraint* is of the form $\phi(t_1, \cdots, t_m)$ where ϕ is a $m-ary$ constraint symbol and each t_i is a term, $1 \leq i \leq m$. A *constraint* is constructed from primitive constraints using logical connectives in the usual manner. Where Ψ is a constraint, we write $\Psi(\tilde{x})$ to denote that Ψ possibly refers to variables in $\tilde{x}$, and we write $\tilde{\exists}\Psi(\tilde{x})$ to denote the existential closure of $\Psi(\tilde{x})$ over variables away from $\tilde{x}$.

A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. A *renaming* is a substitution which maps each variable in the expression into a distinct variable. We write $[\tilde{x} \mapsto \tilde{y}]$ to denote such mappings.

A *grounding* is a substitution which maps each variable into a value in its domain. Where e is an expression containing a constraint Ψ, $[\![e]\!]$ denotes the set of its instantiations obtained by applying *all possible* groundings which satisfy Ψ.

3.2 A Reduction System

A program is represented as a transition system which can be executed symbolically. The following key definition serves two main purposes. First, it is a high level representation of the operational semantics, and in fact, it represents the exact *trace* semantics. Second, it is an *executable specification* against which an assertion can be checked.

We shall model computation by considering *n system variables* $v_1, \cdots, v_n$ with domains $\mathcal{D}_1, \cdots, \mathcal{D}_n$ respectively, and a program counter k ranging over program points. In this paper, we shall use just two example domains, that of integers, and that of integer arrays.

DEFINITION 1 (States and Transitions). *A* ground state *is of the form* $(k, d_1, \cdots, d_n)$ *where k is a program point and* $d_i \in \mathcal{D}_i, 1 \leq i \leq n$, *are values for the system variables. A* transition *is a pair of states.* □

DEFINITION 2 (Symbolic State). *A* symbolic state *(or simply, state) of a CTS is of the form:*

$$(k, \tilde{x}), \quad \Psi(\tilde{x})$$

[5] In this paper, we shall only be using simple integer terms and constraints as examples. In general, we can code data structures such as arrays and pointers.

$(0,x) \mapsto (1,x'), b_1, x' = x+1.$
$(0,x) \mapsto (1,x'), \neg b_1, x' = x+2.$
$(1,x) \mapsto (2,x'), b_2, x' = x+3.$
$(1,x) \mapsto (2,x'), \neg b_2, x' = x+4.$

Figure 3. CTS of Example in Section 2

where k is a program point, $\tilde{x}$ is a sequence of variables over system states, and Ψ is a constraint over some or all of the variables $\tilde{x}$, and possibly some additional variables. The variables $\tilde{x}$ are called the primary *variables of this state, while any additional variable in Ψ is called an* auxiliary *variable of the state. Where $\mathcal{G}$ is a state, we write $cons(\mathcal{G})$ to denote the constraint in $\mathcal{G}$. Finally, we write $\mathcal{G}(\tilde{x})$ to indicate that $\tilde{x}$ are the primary variables of $\mathcal{G}$.* □

DEFINITION 3 (Constraint Transition System). *A* constraint transition *of p is a formula*

$$(k,\tilde{x}) \mapsto (k_1,\tilde{x_1}), \Psi(\tilde{x},\tilde{x_1})$$

where $(k,\tilde{x})$ and $(k_1,\tilde{x}_1)$ are system states, and Ψ is a constraint over $\tilde{x}$ and $\tilde{x}_1$, and possibly some additional auxiliary variables. A constraint transition system *(CTS) of p is a finite set of constraint transitions of p.* □

Clearly the variables in a constraint transition may be renamed freely because their scope is local to the transition. We thus say that a constraint transition is a *variant* of another if one is identical to the other when a renaming substitution is performed. Further, we may *simplify* a constraint transition by renaming any one of its variables *x* by an expression *y* provided that $x = y$ in all groundings of the constraint transition.

The above formulation of program transitions is familiar in the literature for the purpose of defining a set of transitions. What is new, however, is how we use a CTS to define a *symbolic* transition sequences, and thereon, the notion of a proof.

Thus a state is just like the conclusion of a constraint transition.

We say that a state is *false* if its constraint is unsatisfiable. We shall also the notation *false* to denote a *false* state. We say that a state is *final* if *k* is the final program point, one from which there are no transitions. Running an initial state is therefore tantamount to asking the question: which of the values of $\tilde{x}$ that satisfy $\tilde{\exists}\Psi(\tilde{x})$ will lead to a state at the final point(s)? The idea is that we successively reduce one state to another until the resulting state is at a final state, and then inspect the results.

We say that a state $\overline{\mathcal{G}}$ *subsumes* another state $\mathcal{G}$ if $[\![\overline{\mathcal{G}}]\!] \supseteq [\![\mathcal{G}]\!]$. Equivalently, we say that $\overline{\mathcal{G}}$ is a *generalization* of $\mathcal{G}$. We write $\mathcal{G}_1 \equiv \mathcal{G}_2$ if $\mathcal{G}_1$ and $\mathcal{G}_2$ are generalizations of each other. Note that if $\overline{\mathcal{G}}$ is a *generalization* of $\mathcal{G}$, then there is a constraint Ψ such that $\overline{\mathcal{G}} \wedge \Psi \equiv \mathcal{G}$.

Given two states $\mathcal{G}_1 \equiv (k,\tilde{x}_1), \Psi_1$ and $\mathcal{G}_2 \equiv (k,\tilde{x}_2), \Psi_2$ sharing a common program point *k*, we write $G_1 \wedge G_2$ to denote the state $(k,\tilde{x}_1), \tilde{x}_1 = \tilde{x}_2, \Psi_1, \Psi_2$.

Next we define what it means for a CTS to "prove" a state.

DEFINITION 4 (Transition Step, Sequence and Tree). *Let there be a CTS for a program, and let $\mathcal{G} \equiv (k,\tilde{x}), \Psi$ be a state for this. A* transition step *from $\mathcal{G}$ may be obtained providing Ψ is satisfiable. It is obtained using a variant $(k,\tilde{y}) \mapsto (k_1,\tilde{y_1}), \Psi_1$ of a transition in the CTS in which all the variables are fresh. The result is a state of the form $(k_1,\tilde{y_1}), \Psi, \tilde{x} = \tilde{y}, \Psi_1$ We say that this new state is a false state if the constraint $\Psi, \tilde{x} = \tilde{y}, \Psi_1$ is unsatisfiable*[6].

A transition sequence *is a finite sequence of transition steps which terminate in either a final state or a false state. A* transition path *is a finite sequence of transitions corresponding to a transition sequence. Intuitively, a path denotes the "skeleton" of a sequence. A* transition tree *is defined from transition sequences in the obvious way.* □

We shall impose a special condition on transition steps: if a step results in a final state, then the primary variables of the final state are the *final* variables. We say that a transition sequence or path is *successful* if it terminates in a final state; otherwise, the sequence or path is *false*.

DEFINITION 5 (Answer). *Let $\mathcal{G}$ be a state and let $\mathcal{G}_1, \cdots, \mathcal{G}_n$, $n \geq 0$, denote the final states in all of the successful paths starting at $\mathcal{G}$. The answer $\text{ANS}(\mathcal{G})$ of a state $\mathcal{G}$ is the disjunction $cons(\mathcal{G}_1) \vee \cdots \vee cons(\mathcal{G}_n)$. Note that $\text{ANS}(\mathcal{G}) \equiv$ false if there are no successful paths starting at $\mathcal{G}$.* □

4. Summarizations

DEFINITION 6 (Summarization). *A summarization is a pair comprising a state $\overline{\mathcal{G}}$ and a constraint Ψ over $var(\overline{\mathcal{G}})$, final variables and possibly auxiliary variables, such that $\text{ANS}(\overline{\mathcal{G}}) \models \Psi$. We shall call $\overline{\mathcal{G}}$ the* coverage *and Ψ the* answer *of the summarization. We write $\overline{\mathcal{G}} \mapsto \Psi$ to denote such a summarization.*

A state $\mathcal{G}$ is summarized *by a summarization $\Sigma \equiv (\overline{\mathcal{G}} \mapsto \Psi)$ if $\overline{\mathcal{G}}$ is a generalization of $\mathcal{G}$.* □

The general purpose of a summarization $\Sigma = (\overline{\mathcal{G}} \mapsto \Psi)$ is to produce an abstract answer for a state $\mathcal{G}$ for which $\overline{\mathcal{G}}$ is a generalization. That is, $\mathcal{G} \equiv \overline{\mathcal{G}} \wedge \theta$ for some θ. We write $\mathcal{G}\Sigma$ to denote the answer to $\mathcal{G}$ provided by Σ. In this case, $\mathcal{G}\Sigma$ is $\Psi' \equiv cons(\overline{\mathcal{G}}) \wedge \theta \wedge \Psi$, and it is the case that $\text{ANS}(\mathcal{G}) \models \Psi'$.

In this paper, summarizations are used to optimize the process of discovering certain abstract properties about a program. These abstract properties are specified simply by a possibly infinite family $\widetilde{\mathcal{A}}$ of *abstractions*, each of which is a formula over the final variables. We shall call the family of abstractions an *abstract domain*.

DEFINITION 7 (Safety). *A state $\mathcal{G}$ is* safe wrt *a possibly infinite abstract domain $\widetilde{\mathcal{A}}$ if $\text{ANS}(\mathcal{G}) \models \mathcal{A}$ for some $\mathcal{A} \in \widetilde{\mathcal{A}}$. A program is* safe wrt *$\widetilde{\mathcal{A}}$ if all of its final states are safe wrt $\widetilde{\mathcal{A}}$.* □

The simplest example of abstract property is a "safety" property, and this corresponds to having an abstraction family of just one abstraction. In Section 2 for example, we dealt with the abstraction $x_2 - x_0 \leq 6$. Some further examples:

EXAMPLE 1 (Example Abstract Domains).

- *"SIGN"*
  Consider a finite number of abstractions where the formulas are conjunctions of basic constraints of the form $x_i \sim 0$ where the x_i range over a finite set of (interesting) variables, and $\sim$ is one of $<$ and $\geq$. Thus if there are two variables x, y, the set of abstractions is $\{x < 0 \wedge y < 0, x < 0 \wedge y \geq 0, x \geq 0 \wedge y < 0, x \geq 0 \wedge y \geq 0\}$. Note here that we have not included the cases where the sign of x or y is not known[7]*. Thus a program is safe only if every trace results in a definite sign for both x and y, and that x has the same sign in each trace, and similarly for y.*
- *"PARITY"*
  Next consider that each abstraction states the parity of a selected set of variables. If these were x and y, then the abstractions are $\{even(x) \wedge even(y), even(x) \wedge odd(y), odd(x) \wedge$

[6] This particular treatment is not usual in traditional CLP [8], from where these definitions are adapted.

[7] If we did, the abstract domain would be "disjunction-closed".

```
s = 0
if (b_1) s = s + α_1
...
if (b_n) s = s + α_n
```

Figure 4. Sum of Subsets

even$(y), odd(x) \wedge odd(y)\}$ *where the predicates* $odd()$ *and* $even()$ *have the obvious meanings. Once again, a state in which the parity of x or y is not determined is considered unsafe.*

- *"INTERVAL"*
  Consider now the infinite set of abstractions: $\{\alpha \leq x \wedge x \leq \beta : \alpha, \beta \in \mathcal{R}\}$. *Here we are interested in a real number interval bounding the final values of x.* □

The main technical result in this paper is an algorithm which, given an initial state $\mathcal{G}$, computes summarizations on the fly, toward the goal of proving that the final states, ie. $\text{ANS}(\mathcal{G})$, satisfy certain abstract properties.

DEFINITION 8 (Safe Summarization). *A* summarization Σ *of* $\mathcal{G}$ *is* safe wrt $\widetilde{\mathcal{A}}$ *if there is an abstraction* $\mathcal{G}\Sigma \models \mathcal{A}$ *for some* $\mathcal{A} \in \widetilde{\mathcal{A}}$. □

The main property of a safe summarization is illustrated by:

LEMMA 1. *Let* $(\overline{\mathcal{G}} \mapsto \mathcal{A})$ *be a safe summarization. Let* $\mathcal{G}$ *be a specialization of* $\overline{\mathcal{G}}$, *that is,* $\mathcal{G} \equiv (\overline{\mathcal{G}} \wedge \theta)$. *Then* $(\mathcal{G} \mapsto cons(\overline{\mathcal{G}}) \wedge \theta \wedge \mathcal{A})$ *is a safe summarization of* $\mathcal{G}$. □

The algorithm is presented in Figure 5. Its input is a state $\mathcal{G}$, and its return value, if any, is a pair $(\overline{\mathcal{G}} \mapsto \mathcal{A})$ denoting a safe summarization of $\mathcal{G}$. The algorithm may abort if a summarization of the input state cannot be produced. The algorithm comprises two key operations (GEN) and (JOIN). The first (GEN) deals with computing a weakest precondition of a state, which essentially necessitates the computation of a "generalization" of the parent state. The second operation (JOIN) requires that the computed answers of summarizations of descendant states be combined into a summarization answer for the parent state.

Define *weakest precondition* $\text{WP}(R, \mathcal{G})$ of a state $\mathcal{G}$ produced by a transition R to be a state $\mathcal{G}'$ such that there is a proof step from $\mathcal{G}'$ to a variant of $\mathcal{G}$ using the transition R. Formally,

DEFINITION 9 (Weakest Precondition). *Suppose that a transition R and a state* $\mathcal{G}$ *are of the form:*

$$(k, \tilde{x}) \mapsto \Psi_R(\tilde{x}, \tilde{x}'), (k', \tilde{x}') \quad \textit{and} \quad (k', \tilde{x}), \Psi$$

respectively. Then

$$\text{WP}(R, \mathcal{G}) \stackrel{\text{def}}{=} (k, \tilde{x}), \forall \tilde{x}'. \Psi_R(\tilde{x}, \tilde{x}') \longrightarrow \Psi[\tilde{x}'/\tilde{x}] \quad \square$$

In general, it is not practical to compute this function $\text{WP}()$ precisely. Consider, for example, the computation of $\text{WP}(R, false)$, where R is as in Definition 9, precisely. We would require a state $(i, \tilde{x}), \Psi$ such that $\Psi \wedge \Psi_R$ is unsatisfiable, but for *any* generalization $\overline{\Psi}$ of Ψ, $\overline{\Psi} \wedge \Psi_R$ is satisfiable. In general, such Ψ would only be describable as a (large) disjunction in the available constraint language.

A concrete example of the intractability of $\text{WP}()$ is given in Figure 4, where all propositional combinations of the b_i are satisfiable. Implementing the well-known NP-complete sum-of-subsets problem: given a set S of n numbers $\{\alpha_1, \cdots, \alpha_n\}$ and another number β, does a subset of S sum to β? Since the program takes no input, this problem can be reduced to determining if the weakest precondition of the program w.r.t the constraint $s = \beta$ is empty or not.

A possibly practical approach may be obtained from the literature on *minimal unsatisfiable subsets* which attempts to find, given an unsatisfiable set of constraints, a subset thereof which is also unsatisfiable. See eg. [4]. Note that such a set is not unique. To apply this technique, we simply use the constraints attached to the parent state of this *false* state in question, conjoin the constraint from the transition R, and consider all of these constraints as a set. In case we were computing the weakest precondition $\text{WP}(R, \mathcal{G})$ of a non-false state instead of $\text{WP}(R, false)$, this approach can still be used in case the constraint in $\mathcal{G}$ can be efficiently negated. That is, we accumulate the constraints in the parent state of $\mathcal{G}$, the constraint in the transition R and finally, the constraint $\neg cons(\mathcal{G})$, and consider all of these as a set.

In a more general setting than finding minimal unsatisfiable subsets is the notion of a *Craig interpolant*, which we informally describe as follows: given $\Psi \models \Psi'$, find a generalization $\overline{\Psi}$ of Ψ such that $\overline{\Psi} \models \Psi'$ also. This converges with the minimal subsets approach when $\Psi' \equiv false$. The use of such interpolants for model-checking [13] and theorem-proving [14] is now commonplace.

In summary for (GEN), it is in general not practical to compute the *weakest* precondition of state as in the idealized algorithm. Indeed, computing the weakest precondition is intractably hard, as we exemplify later. Thus in practice, one would compute just a precondition of the descendant $\overline{\mathcal{G}}_i$ which is at least as general as the parent $\mathcal{G}$, as opposed to the weakest precondition. In section 6, we demonstrate such an algorithm based upon an efficient implementation of constraint deletions.

The step (JOIN) serves to combine several constraints into one such that the result is safe. Note that although these constraints are in fact safe summarizations of states that are descendants of the current state, the disjunction of these constraints is not necessarily safe. (Recall the abstraction family SIGN above, and suppose there are two constraints to be joined: $even(x) \wedge even(y)$ and $even(x) \wedge odd(y)$. Clearly there can be no join of these two constraints, and our algorithm would abort at this point.) The operation JOIN can in principle be implemented simply as a disjunction of its arguments. However, this would not be scalable, for the number of disjunctions would increase exponentially.

Fortunately, it is often not necessary to represent precisely this disjunction. Recall that the obligation of a summarization's answer is just that it *entails* an abstraction. Thus for example, if we were dealing with a safety property and therefore a single abstraction $\mathcal{A}$, then JOIN() can simply return $\mathcal{A}$ itself (This is assuming that the disjunction is in fact safe; if not, JOIN() must say so, and the algorithm aborts.).

Now if we were dealing with a finite or infinite abstract domain, JOIN() can simply return one of the abstractions. However, it may be the case that the abstraction used may not be the most precise possible. More specifically, recall that a summarization $\Sigma = (\overline{\mathcal{G}} \mapsto \Psi)$ should be such that for any instance of $\mathcal{G}$ of $\overline{\mathcal{G}}$, $\mathcal{G}\Sigma$ should entail an abstraction. The problem however is that $\mathcal{G}\Sigma$ may not entail the most precise abstraction. Recall that in the example of Section 2, in the scenario where $\neg b_1 \wedge \neg b_2$ was unsatisfiable, we computed a summarization whose answer was $x_2 - x_0 \leq 6$ for the state $\mathcal{G}_0$, and when we apply this to the state $\mathcal{G}_1$, we finally concluded that $x_2 - x_0 \leq 6$. This is safe, but not precise, for $x_2 - x_0 \leq 5$ is a better abstraction.

In summary for (JOIN), one needs to combine two summarized answers with a constraint must trade off two things: to be

- precise enough to ensure safety, and yet
- abstract enough so that the disjunction can be efficiently represented.

SOLVE($\mathcal{G} \equiv ((k, \tilde{x}_k), \Psi)$) **returns** $(\overline{\mathcal{G}} \mapsto \Psi')$

- $\mathcal{G}$ is *false*: **return** $(\mathit{false} \mapsto \mathit{false})$
- $\mathcal{G}$ is final:
  if (Ψ is unsafe) **abort**
  return $((k, \tilde{x}_k) \mapsto \Psi)$
- $\mathcal{G}$ is summarized by some memoized Σ: **return** Σ (SAFE?)
- $\mathcal{G}$ is composite:
  for each derivation $\mathcal{G} \overset{R}{\mapsto} \mathcal{G}_i(\tilde{x}_{k+1}), 1 \le i \le n$
  let the constraint in R be $\phi_i(\tilde{y}, \tilde{y}')$
  let $(\overline{\mathcal{G}_i} \mapsto \Psi_i) =$ SOLVE($\mathcal{G}_i$)
  let $\mathcal{H}_i(\tilde{x}_k) =$ WP$(R, \overline{\mathcal{G}_i})$ (GEN)
  let $\mathcal{H} = \mathcal{H}_1 \wedge \cdots \wedge \mathcal{H}_n$
  let $\Psi' =$ JOIN$(\phi_1 \wedge \Psi_1, \ \cdots, \phi_n \wedge \Psi_n)$ (JOIN)
  memoize the summarization $\Sigma \equiv (\mathcal{H} \mapsto \Psi')$
  return Σ

Figure 5. Idealized Algorithm for Safe Summarizations

While the choice of any implementation of (JOIN) which preserves safety will ultimately produce a safe summarization, this choice does affect the precision. In the next section, we refine the algorithm to ensure that the answers to summarizations are, in some sense, exact.

We now exemplify possible implementations of JOIN() for the example abstractions in Section 2. To simplify matters, assume that JOIN() takes two arguments $\mathcal{A}_1$ and $\mathcal{A}_2$.

For an abstract domain representing just a single abstraction $\mathcal{A}$, JOIN(), can simply be $\mathcal{A}$. Note that there is no need to check that both $\mathcal{A}_1$ and $\mathcal{A}_2$ imply $\mathcal{A}$, because the current state $\mathcal{G}$ is in the precondition of descendant states which are already safely summarized to the answer $\mathcal{A}$.

For finite or infinite abstract domains, it is safe that JOIN() returns any abstraction in this domain as long as it is safe. In general, of course, the most precise should be chosen. However, as explained above, even if JOIN() consistently chooses the most precise abstraction possible and eventually constructs a summarization, it is not the case that this summarization will exactly represent those states within its coverage.

Reconsider now the abstract domain "SIGN" of Example 1. An implementation for JOIN() is straightforward: simply determine the sign of the two variables x and y in each of $\mathcal{A}_1$ and $\mathcal{A}_2$. We abort if the sign of one of them is not determined, or if the sign of one of them is different in $\mathcal{A}_1$ and $\mathcal{A}_2$. Otherwise, the return of JOIN$(\mathcal{A}_1, \mathcal{A}_2)$ is the single constraint expressing the sign of both x and y that is evident in both $\mathcal{A}_1$ and $\mathcal{A}_2$. For the abstract domain "PARITY", implementation is equally straightforward: simply determine the parity of the two variables x and y in each of $\mathcal{A}_1$ and $\mathcal{A}_2$. the return of JOIN$(\mathcal{A}_1, \mathcal{A}_2)$ is the single constraint parity of both x and y that is evident in both $\mathcal{A}_1$ and $\mathcal{A}_2$.

Finally consider the domain "INTERVAL". An implementation of JOIN$(\mathcal{A}_1, \mathcal{A}_2)$ would simply construct an interval from the two intervals indicated in $\mathcal{A}_1$ and $\mathcal{A}_2$. That is, where $\mathcal{A}_1$ is of the form $\alpha_1 \le x \le \beta_1$ and $\mathcal{A}_2$ is of the form $\alpha_2 \le x \le \beta_2$, obviously JOIN$(\mathcal{A}_1, \mathcal{A}_2)$ is the single constraint $\mathit{min}(\alpha_1, \alpha_2) \le x \le \mathit{max}(\beta_1, \beta_2)$. Thus in all three cases, the implementation of JOIN() is straightforward.

THEOREM 1 (Safety). *Given a state $\mathcal{G}$ and a family of abstractions $\widetilde{\mathcal{A}}$ for var$(\mathcal{G})$, the algorithm in Figure 5 returns a safe summarization for $\mathcal{G}$ in case $\mathcal{G}$ is safe wrt $\widetilde{\mathcal{A}}$; otherwise, the algorithm aborts.*

PROOF OUTLINE: *We proceed by induction. First, the base cases: both the false state and final state cases return exact summarizations to the parent state is easy to see. Now assume, in the processing of state $\mathcal{G}$, that the return values $(\overline{\mathcal{G}_i} \mapsto \Psi_i)$ are exact summarizations of the descendant states $\mathcal{G}_i$, $1 \le i \le n$. Now, since $\mathcal{G}$ runs to $\mathcal{G}_i$ and since $\overline{\mathcal{G}_i}$ is more general than $\mathcal{G}_i$, it follows from step (GEN) that $\mathcal{G}$ is in the weakest precondition of $\overline{\mathcal{G}_i}$. Thus $\phi_i \wedge \Psi_i$, $1 \le i \le n$, is safe. Finally, since (JOIN) combines these as a disjunction, the result Ψ' is safe.* □

The key reason why this idealized algorithm has potentially good performance is that once it has summarized a state $\mathcal{G}$, it can use this summarization for future encountered states $\mathcal{G}'$ whose *final paths are contained in those of* $\mathcal{G}$. that is, $\mathcal{G}'$ does not allow a path which was infeasible in $\mathcal{G}$. Now, even if $\mathcal{G}'$ were not summarized, it means the work about to be performed on $\mathcal{G}$ accounts for a path that is not yet explored. Further, even if $\mathcal{G}'$ were not summarized, there is every opportunity for one of its descendants to be summarized (by a descendant of $\mathcal{G}$) so long as this descendant is not on the same said path.

Even so, in general, the search process can be intractable, as explained above for the program in Figure 4.

5. Exact Summarizations

In most settings, proving safety is just a decision problem. In our setting, we seek not just to prove safety, but to *discover* the abstraction which establishes the safety. More importantly, we seek to discover the *best* abstraction which establishes the safety. The motivation for this is partly due to the fact that we are dealing with straight-line programs and we wish to ensure that our algorithm performs exact propagation with respect to a given abstract domain.

This section extends the previous to the case where we desire not just a safe summarization, but, in some sense, an "exact" summarization.

Given an abstract domain $\widetilde{\mathcal{A}}$, we say that an abstraction $\mathcal{A} \in \widetilde{\mathcal{A}}$ is *strictly more precise* that another $\mathcal{A}' \in \widetilde{\mathcal{A}}$ if $\mathcal{A} \models \mathcal{A}'$ but not $\mathcal{A}' \not\models \mathcal{A}$.

DEFINITION 10 (Exact Summarization). *A* summarization Σ *of* $\mathcal{G}$ *is* exact wrt *an abstract domain $\widetilde{\mathcal{A}}$ if $\mathcal{G}\Sigma \models \mathcal{A}$ for some $\mathcal{A} \in \widetilde{\mathcal{A}}$. Furthermore, for any abstraction $\mathcal{A}'$ in $\widetilde{\mathcal{A}}$ which is strictly more precise than $\mathcal{A}$ is* unsafe *for $\mathcal{G}$, that is,* ANS$(\mathcal{G}) \not\models \mathcal{A}'$. □

The algorithm in Figure 5 does not in general produce an exact summarization for the input state $\mathcal{G}$. Recall that a summarization $\Sigma \equiv (\overline{\mathcal{G}} \mapsto \Psi)$ guarantees that a state $\mathcal{G}$ covered by $\overline{\mathcal{G}}$, say $\mathcal{G} \equiv \overline{\mathcal{G}} \wedge \theta$, has its answers covered by $\theta \wedge \Psi$. Thus if Σ were a safe summarization, then $\mathcal{G}$ is safe. That is, $\mathcal{G}\Sigma$ implies an abstraction. In contrast, if Σ were an *exact* summarization, while $\mathcal{G}\Sigma$ is safe and entails some abstraction $\mathcal{A}$, it is in general *not* the case that $\mathcal{A}$ is a maximally precise abstraction. Recall that we have exemplified this in Section 2.

We now present a new and refined definition of summarization. Previously, a summarization was a pair, the first of component of which is coverage, and the second was an answer constraint. We now refine the definition of summarization so that there are now two components of coverage.

DEFINITION 11 (Summarization, Second Version). *A summarization is a triple comprising a state $\overline{\mathcal{G}}$, a set $\widetilde{\Psi}$ of constraints over var$(\mathcal{G})$ and final variables, and a constraint Ψ over var$(\mathcal{G})$ and final variables such that* ANS$(\overline{\mathcal{G}}) \models \Psi$. *We shall call $\overline{\mathcal{G}}$ the* MAX coverage, *$\widetilde{\Psi}$ the* MIN coverage, *and Ψ the* answer *of the summarization. We write $(\overline{\mathcal{G}}, \widetilde{\Psi}) \mapsto \Psi$ to denote such a summarization.*

SOLVE($\mathcal{G} \equiv ((k,\tilde{x}_k),\Psi)$) **returns** $((\overline{\mathcal{G}},\widetilde{\Psi}) \mapsto \Psi')$

- $\mathcal{G}$ is *false*: **return** $((false,\{\}) \mapsto false)$
- $\mathcal{G}$ is final:
  if (Ψ is unsafe) **abort**
  return $(((k,\tilde{x}_k),\{\Psi\}) \mapsto \Psi)$
- $\mathcal{G}$ is summarized by some memoized Σ: **return** Σ (SAFE?)
- $\mathcal{G}$ is composite:
  for each derivation $\mathcal{G} \overset{R}{\mapsto} \mathcal{G}_i(\tilde{x}_{k+1}), 1 \le i \le n$
  let the constraint in R be $\phi_i(\tilde{y},\tilde{y}')$
  let $((\overline{\mathcal{G}}_i,\widetilde{\Psi}_i) \mapsto \Psi_i) = \text{SOLVE}(\mathcal{G}_i)$
  let $\mathcal{H}_i(\tilde{x}_k) = \text{WP}(R,\overline{\mathcal{G}}_i)$ (GEN)
  let $\mathcal{H} = \mathcal{H}_1 \wedge \cdots \wedge \mathcal{H}_n$
  let $\widetilde{\Psi} = \text{REP}(\phi_1 \wedge \widetilde{\Psi}_1, \cdots, \phi_n \wedge \widetilde{\Psi}_n)$ (REP)
  let $\Psi' = \text{JOIN}(\phi_1 \wedge \Psi_1, \ \cdots, \phi_n \wedge \Psi_n)$ (JOIN)
  memoize the summarization $\Sigma \equiv ((\mathcal{H},\widetilde{\Psi}) \mapsto \Psi')$
  return Σ

Figure 6. Idealized Algorithm for Exact Summarization

A state $\mathcal{G}$ is summarized *by $\Sigma = ((\overline{\mathcal{G}},\widetilde{\Psi}) \mapsto \Psi)$ if*

- *(a) $\overline{\mathcal{G}}$ is a generalization of $\mathcal{G}$ (as before), and*
- *(b) $\mathcal{G} \wedge \Psi_i$, for all $\Psi_i \in \widetilde{\Psi}$, is satisfiable.* □

Suppose $\mathcal{G} \equiv \overline{\mathcal{G}} \wedge \theta$. As before, we write $\mathcal{G}\Sigma$ to denote the answer of $\mathcal{G}$ as given by the summarization Σ, and this is $cons(\overline{\mathcal{G}}) \wedge \theta \wedge \Psi$. The reason that the condition in (a) is called MAX is that it represents a "maximum generalization" of a potentially summarized state. In (b) however, the condition represents a "minimum specialization" of a potentially summarized state.

We now refine our algorithm and obtain the algorithm in Figure 6. The first change is the definition of whether a state is summarized by a previously computed summarization, and this is given above in Definition 11. The major change however, is the (REP) step, which serves to produce, given a family of constraints, one set of constraints. The notation $\phi \wedge \widetilde{\Psi}$ used in the new step (REP) denotes the set of constraints obtained by conjoining ϕ with each constraint in $\widetilde{\Psi}$.

The input to REP() is a family representing the various representative sets $\widetilde{\Psi}_1, \cdots, \widetilde{\Psi}_n$ obtained from summarizations of the descendant states $\mathcal{G}_1, \cdots, \mathcal{G}_n$. These summarizations produce answers for $\mathcal{G}_1, \cdots, \mathcal{G}_n$ which together imply that a certain abstraction, say $\mathcal{A}$, is the most precise one that applies. The function REP() then computes, from this family, a representative set of constraints which, when conjoined with $\mathcal{G}$, demonstrates that $\mathcal{A}$ holds. It thus follows that REP() computes a best set of constraints that demonstrates "exact safety".

We next exemplify that computing REP() can be efficient.

Consider some examples from Example 1 for the function $\text{REP}(\widetilde{\Psi}_1, \cdots, \widetilde{\Psi}_n)$. For the abstract domains "SIGN" and "PARITY", the return value can simply be *any subset* of constraints $\widetilde{\Psi}$ in $\widetilde{\Psi}_1 \cup \cdots \cup \widetilde{\Psi}_n$ that precisely describes the signs/parities appearing in all the constraints. Note that the choice of which set constraint does however affect how often a summarization is effective on summarizing a candidate state. Clearly we could use an arbitrary set of constraints larger than $\widetilde{\Psi}$. This would increase the likelihood of a summarization being able to cover a candidate state, but at the expense of increased cost in the storage of this representative set.

Next consider the domain "INTERVAL". Recall that the proposed $\text{JOIN}(\mathcal{A}_1, \mathcal{A}_2, \cdots, \mathcal{A}_n)$, where $\mathcal{A}_i$ is of the form $\alpha_i \le x \le \beta_i$, $1 \le i \le n$, is to choose the interval $\alpha \le x \le \beta$ such that $\alpha = min(\alpha_1, \cdots, \alpha_n)$ and $\beta = max(\alpha_1, \cdots, \alpha_n)$. Suppose α_{min} were the least number in $\{\alpha_1, \cdots, \alpha_n\}$, and β_{max} the greatest number in $\{\beta_1, \cdots, \beta_n\}$.

Consider $\text{REP}(\widetilde{\Psi}_1, \cdots, \widetilde{\Psi}_n)$. Following the interval logic above, each set $\widetilde{\Psi}_i$ would contain at most two constraints Ψ_i^{min} and Ψ_i^{max} corresponding to the two answers that give rise to the interval representing the answer to the summarization of descendant state $\mathcal{G}_i$. We now choose, amongst these $2 * n$ constraints, just the *one or two* constraints that demonstrates that the new interval is indeed $\alpha \le x \le \beta$.

We now return to the example in Section 2, and consider the infinite abstract domain $\{x_2 - x_0 \le \alpha : 0 \le \alpha\}$. Recall that after analyzing the subtree for $\mathcal{G}_0$, we obtained the summarization

$$((1,x_1), x_1 - x_0 \le y + 2, y \ge 0 \mapsto x_2 - x_0 \le y + 6, y \ge 0)$$

of $\mathcal{G}_0$. This was derived by considering two subtrees of $\mathcal{G}_0$. corresponding to the adding of the constraints $b_2, x_2 = x_1 + 3$ toward $\mathcal{G}_{00}$, and the constraint $\neg b_2, x_2 = x_1 + 4$, toward $\mathcal{G}_{01}$. For the representative set of $\mathcal{G}_0$, we would need to consider just one, the latter. This is because any candidate state $\mathcal{G}'$ which is summarized by this summarization Σ, would indeed produce the exact increment of x in the answer $\mathcal{G}'\Sigma$. If of course this candidate state did not satisfy $\neg b_2, x_2 = x_1 + 4$, then it is not considered summarized, and its tree would have to be analyzed. In either case, we would produce, at the end, an exact summarization.

The following theorem statement and proof outline are slight modifications of those of Theorem 1.

THEOREM 2 (Exact Summarization). *Given a state $\mathcal{G}$ and a family of abstractions $\widetilde{\mathcal{A}}$ for $var(\mathcal{G})$, the algorithm in Figure 6 returns an exact summarization for $\mathcal{G}$ in case $\mathcal{G}$ is safe wrt $\widetilde{\mathcal{A}}$; otherwise, the algorithm aborts.*

PROOF OUTLINE: *We proceed by induction. First, the base cases: both the false state and final state cases return safe summarizations to the parent state is easy to see. Now assume, in the processing of state $\mathcal{G}$, that the return values $((\overline{\mathcal{G}}_i,\widetilde{\Psi}_i) \mapsto \Psi_i)$ are exact summarizations of the descendant states $\mathcal{G}_i$, $1 \le i \le n$. As before, since $\mathcal{G}$ runs to $\mathcal{G}_i$ and since $\overline{\mathcal{G}}_i$ is more general than $\mathcal{G}_i$, it follows from step (GEN) that $\mathcal{G}$ is in the weakest precondition of $\overline{\mathcal{G}}_i$. Thus $\phi_i \wedge \Psi_i$, $1 \le i \le n$, is safe, and since (JOIN) combines these as a disjunction, the result Ψ' is safe.*

But now, we use the fact that the set of constraints: $\{\phi_i \wedge \widetilde{\psi}_i : 1 \le i \le n\}$ which is input to REP(), will produce as output a constraint whose abstraction is the same as the abstraction of the input set. The theorem now follows easily. □

6. Experimental Evaluation

In our experimental evaluation, we have employed CLP technology [7], which provides an efficient mechanism for the internal representation of an incrementally growing a proof sequence as a set of constraints, and for structure sharing of common constraints across different proof sequences. One important feature of this technology is an efficient projection algorithm. Our experimental system is based on the CLP($\mathcal{R}$) system [10] with an adaptation of the Fourier-Motzkin algorithm [9].

Our implementation of the GEN operation uses a process of constraint deletion to compute preconditions. A constraint is deemed to be redundant, and therefore deletable, at a node in the computation tree, if its removal does not change the status of any of the leaf nodes in the corresponding subtree. That is, upon the removal of a redundant constraint, *false* state remain *false*, and

Problem	Array Size	No Summarization Nodes	No Summarization Time (s)	Safe Summarization Nodes	Safe Summarization Time (s)	Safe Summarization Answer	% Space (Time) Reduction	Exact Summarization Nodes	Exact Summarization Time (s)	Exact Summarization Answer	% Space (Time) Reduction
Unrestricted	5	2233	11.22	58	0.04	10	97.40% (99.64%)	58	0.05	10	97.40% (99.55%)
	10	∞		218	0.92	45		218	0.96	45	
	15	∞		478	6.92	105		478	7.04	105	
Binary	4	381	0.70	162	0.27	4	57.48% (61.43%)	169	0.30	4	55.64% (57.14%)
Elements	6	2825	27.47	729	4.64	9	74.19% (83.11%)	873	6.54	9	69.10% (76.19%)

Table 1. Bubble Sort

Problem	No Summarization Nodes	No Summarization Time (s)	Safe Summarization Nodes	Safe Summarization Time (s)	Safe Summarization Answer	% Space (Time) Reduction	Exact Summarization Nodes	Exact Summarization Time (s)	Exact Summarization Answer	% Space (Time) Reduction
decoder	344	0.31	38	0.02	22	88.95% (93.55%)	132	0.19	22	61.63% (38.71%)
sqrt	923	4.25	236	1.37	209	74.43% (67.76%)	253	1.43	141	72.59% (66.35%)
qurt	1104	14.47	273	2.52	220	75.27% (82.58%)	290	2.60	152	73.73% (82.03%)
janne_complex	1517	17.93	410	2.13	105	72.97% (88.12%)	683	4.36	81	55.98% (75.68%)

Table 2. Some Random Programs

```
for (i=0; i<n-1; i++) {
    for (j = 0; j<n-i-1; j++) {
        if (a[j] > a[j+1])
            swap(a,j,j+1);
    }}
```

Figure 7. Bubble Sort

subsumed states remain subsumed. In principle, this obligation is as hard as discovering a minimal unsatisfiable subset of constraints in an unsatisfiable collection; see eg. [4]. However, in practice (and in all the examples in this paper), a simple algorithm suffices: consider the constraints one at a time, and perform the necessary redundancy test. We further suggest that is probable that faster heuristic algorithms can be developed based on the intuition that often, the only possible candidates are easily detectable.

For the JOIN operation of the algorithm in Figure 5, we use numeric bounds, as illustrated in Section 2. Whenever it is known that a minimal or maximal bound of the variation of a variable is sought, we aggregate interval constraints in the obvious way. These two very straightforward abstraction algorithms account for much of the efficiency of our approach. Nevertheless, new abstraction methods can be easily embedded into our main algorithm in a natural way.

Our prototype summarization system has been implemented as a pure CLP($\mathcal{R}$) [10] program, with emphasis on its meta-level facilities [6]. We performed our experiments on a Pentium 4 system, with a 2.8GHz clock, 512Mb RAM, and running Linux 2.4.22. The tests have been performed on several randomly chosen programs, instrumented so that each primitive action would increment a resource counter.

For the bubble sort program in Figure 7, we have performed a variety of tests, considering several array sizes, as well as the case when the array elements are binary (i.e. restricted to two values only). In our experiments, we have assumed that the value of `n` is given. In the case when the array elements are unrestricted, a proof tree would have very few *false* states (just those corresponding to leaving the loop early). Here our algorithm, proceeding along the lines of the abstractions described in section 2, would have a linear performance in the size of the array.

If however the array elements are restricted in some way, performance is far less predictable. When the array elements are restricted to binary values, the number of false states (corresponding to impossible combinations of swap operations) is far larger, and in practice, unpredictable. In this case, our algorithm exhibits less impressive performance because of fewer subsumed nodes, but nevertheless provides significant improvement.

Our prototype simply generates the entire state space, and for the purpose of verifying an error condition, an on-the-fly error state reachability check can be easily added.

Results are shown in Table 1. For both unrestricted and binary elements, we use fixed array sizes given in the 2nd column. The "Nodes" columns in the table contain the numbers of nodes in the search tree of our prototype, and "Time" is the running time in seconds. Note the linear growth in nodes traversed for the unrestricted version. The binary version of bubble sort introduced more constraints, as expected, leading to a more complex analysis which decreases the amount of reduction. Nevertheless, summarizations produced huge savings.

We also tested our prototype by generating the state-space search tree of several programs for WCET analysis benchmarking, which exhibit a variety of program control structures. The decoder program is taken from the ADPCM encoder and decoder that appears in [17]. The `sqrt` and `qurt` programs are from the SNU RT Benchmark Suite [19], and the `janne_complex` program is from the Mälardalen Benchmark Suite [12]. The results are shown in Table 2. All the experiments show a significant amount of reduction.

7. Conclusion

We considered the problem of exact propagation through a straight-line program in the pursuit of a final assertion. The final assertion is a user-defined function and so we in fact address the problem of discovering a final abstraction, in addition to just proving safety in terms of a given final condition. Our method is fundamentally a path enumeration method, but the main contribution is an optimization based on the use of summarizations that are obtained dynamically. The method is designed to produce, for each summarization, the most general context for its use, and the most specific conclusion for its result. Importantly, these summarizations apply to arbitrary program fragments and hence there is a fine-grain resolution to their potential usefulness. Finally, we demonstrated the efficiency of the optimization on several programs.

References

[1] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Transactions on Programming Languages and Systems*, 27(2):314–343, 2005.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *5th TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[3] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.Neuhold, editor, *Formal Description of Prog. Concepts*. North-Holland, 1978.

[4] Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *ACM PPDP*, pages 32–43, 2003.

[5] S. Graf and H. Saïdi. Construction of abstract state graphs of infinite systems with PVS. In O. Grumberg, editor, *9th CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[6] N. Heintze, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. Meta programming in CLP($\mathcal{R}$). *Journal of Logic Programming*, 33(3):221–259, December 1997.

[7] J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Projecting CLP($\mathcal{R}$) constraints. In *New Generation Computing*, volume 11, pages 449–469. Ohmsha and Springer-Verlag, 1993.

[8] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, May/July 1994.

[9] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Output in CLP($\mathcal{R}$). In *Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Japan*, volume 2, pages 987–995, 1992.

[10] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.

[11] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In T. Ball and R. B. Jones, editors, *18th CAV*, volume 4144 of *LNCS*, pages 424–437. Springer, 2006.

[12] Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[13] K. L. McMillan. Interpolation and SAT-based model checking. In Jr. W. A. Hunt and F. Somenzi, editors, *15th CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[14] K. L. McMillan. An inerpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[15] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *31st POPL*. ACM Press, 2004.

[16] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd POPL*, pages 49–61. ACM Press, 1995.

[17] R. Richey. *Adaptive Differential Pulse Code Modulation Using PICmicro Microcontrollers*. Microchip Technology, Inc., 1997.

[18] M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[19] SNU real-time benchmarks. URL `http://archi.snu.ac.kr/real-time/benchmark/`.