Abstraction on Demand

JOXAN JAFFAR ANDREW E. SANTOSA RĂZVAN VOICU

SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
REPUBLIC OF SINGAPORE
{JOXAN,ANDREWS,RAZVAN}@COMP.NUS.EDU.SG

Abstract

We present an analysis technique for structured sequential programs based upon abstract reasoning over symbolic traces. The novelty is that the technique performs abstraction *on demand*, as opposed to systematically, in the pursuit of a safety proof. The idea is to maintain precision to enhance the likelihood of proof, performing abstraction only when efficiency is at stake. The technique is automatic in the sense that no user input is required.

The main technical result is analysis of bounded programs, those whose symbolic traces are bounded in length, where the analysis is exact. For example, loop-free programs fall into this class. While the problem is exponential-time in general, we introduce a notion of *dynamic summarizations* which can enable substantial pruning of the search space of traces. Summarizations also allow our analysis to be compositional, key to extending the bounded-program analysis technique to the general case.

Finally, the technique is *tunable*, primarily for two reasons. First, the technique is compositional. Second, the technique at its core is a function which generalizes a set of constraints, and custom functions may be directly used.

1. Introduction

We present an analysis technique for structured sequential programs based upon abstract reasoning over symbolic traces. The novelty is that the technique performs abstraction *on demand*, as opposed to systematically, in the pursuit of a safety proof. The idea is to maintain precision to enhance the likelihood of proof, performing abstraction only when efficiency is at stake. The technique is automatic in the sense that no user input is required.

The main technical result is analysis of bounded programs, those whose symbolic traces are bounded in length, where the analysis is exact. For example, loop-free programs fall into this class. While the problem is exponential-time in general, we introduce a notion of *dynamic summarizations* which can enable substantial pruning of the search space of traces.

A summarization is a partial description of the input-output behavior of a program fragment. A key advantage of a summarization is that, during analysis, invocations of the program fragment under different contexts may be handled by the one summarization, without the need to re-analyze the program fragment for the various contexts. In this paper, we present a method for deriving summarizations dynamically. In fact, it computes, opportunistically and on-the-fly, expressive summarizations of arbitrary program fragments. Our method can 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.

In general, we consider a program P_0, and the objective is to summarize a certain program fragment P of P_0, which is 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 $count'$.

Another important feature of a summarization is that it allows our analysis technique to be *compositional*. This feature is next used to extend the bounded-program analysis technique to the general case. The important new problem, is, of course, the loop. Here we assume, recursively, that the loop body is already analyzed and the result is realized as a summarization. We then attempt to propagate the context in which this loop is called through the loop body so that the context continues to hold after the loop body. In other words, we check if the context is "loop invariant". If not, we employ a generalization algorithm to the context, and repeat the process. This process can be easily made to terminate because the context can only reduce, in the worst case, to the empty context.

Finally, the technique is *tunable*, primarily for two reasons. First, the technique is compositional, and second, the generalization functions may be customized.

Because of compositionality, program fragments may be analyzed with different specializations of the technique. For example, while the basic algorithm is exact on a loop-free program, a judicious partition of the program into two fragments can significantly improve performance while preserving a proof. Proofs can then be pieced together using summarizations. Now, for a single program fragment, the technique, at its core, depends on functions which generalize a set of constraints. User intervention here is easily admitted, and can greatly influence the efficiency of a proof. In its original specification, this algorithm removes constraints one by one, and the number of constraints is proportional to the length of the symbolic trace being analyzed. It is straightforward to use instead a different abstraction algorithm, for example, one that is based on predicate abstraction [5]. While in principle even a straight-line program will require exponential time to perform an

arbitrarily accurate analysis, summarizations seem to be an effective remedy. For loops, the abstraction algorithm can be specialized by providing the appropriate invariant.

We elaborate in section 5 below.

1.1 Related Work

The framework of symbolic traces used here was introduced in [9]. There the emphasis was applying abstraction selectively, but not on-demand, as is the case here.

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, [14] 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. The work [12] performs summarizations on the fly. However, it 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. The work [11] presents a method for summarizing procedures in concurrent programs. 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.

Our approach differs mainly in that summarizations are computed opportunistically and on the fly, for arbitrary program fragments.

In this paper, we perform dynamic abstract interpretation in the sense that the abstract function used for pruning search is computed not statically, but on the fly. The most well-known dynamic abstraction method is CEGAR, see eg: [2]. Here predicate abstraction was used to provide a family of abstraction functions, and the most general abstraction function was used first in a standard abstract interpretation algorithm in an attempt at proof. Upon failure, the specific counterexample was then used in order to obtain a (slightly) more specific abstraction function, and the process is repeated. The idea is that more general abstraction functions, while providing for less accuracy, provide for a more efficient search space. More specific abstraction functions are thus on-demand. A strength of CEGAR is that refinement proceeds automatically under the framework of predicate abstraction. Possible weaknesses are that the refinement operation may not be easily tunable to the problem at hand, and more significantly in relation to this paper, CEGAR is not compositional.

We take a dual philosophy: we start with more specific abstraction functions (in fact no abstraction at all) and so are more likely to achieve proof. However, when circumstances threaten efficiency, we then resort to more general abstraction functions. While we do not explicitly specify an automatic way to iterate through a space of abstraction functions, this is straightforward to implement, in principle.

2. Dynamic Summarizations for Bounded Programs

Here we present informally our algorithm for the summarization of a bounded program. Consider the program *P*

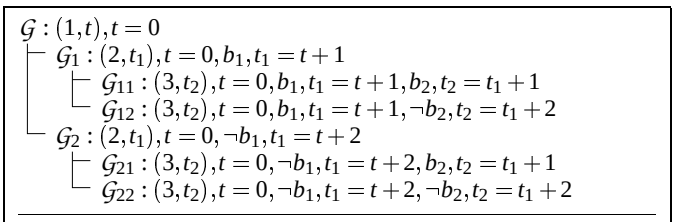

Figure 1. A First Example

```
⟨0⟩ count = 0
⟨1⟩ if b1 then count++ else count = count + 2
⟨2⟩ if b2 then count++ else count = count + 2
final = count ⟨3⟩
```

where we assume that the unspecified predicates `b1` and `b2` are not related to each other, that is, all the four combinations of truth values are possible across the various contexts in which the program fragment may be executed.

We shall symbolically describe a set of states in the form $(k,c,f),\Psi$, where the expressions k,c,f represent the values of the program counter and variables `count` and `final`, respectively. The symbol Ψ is simply a *constraint* or conjunction of constraints on these expressions. Call such an expression $(k,c,f),\Psi$ a *goal*. The semantics of such a goal is simply the set of instances of (k,c,f) for which the constraint Ψ is *true*.

A notion of strongest postcondition can now be obtained. For example, the set of states described by $\mathcal{G} \equiv (1,c,f), c = 0$ can be propagated along the "then" branch of the first `if` statement resulting in $\mathcal{G}_1 = (2,c_1,f), c = 0, c_1 = c+1, b_1$, substituting c and f for `count` and `final` respectively. Propagating the same along the "else" branch of the same `if` statement results in $\mathcal{G}_2 \equiv (2,c_2,f), c = 0, c_2 = c+2, \neg b_1$. Note that through propagation, the symbol c that appeared originally in $\mathcal{G}$ has now become an *auxiliary* symbol in $\mathcal{G}_1$ and $\mathcal{G}_2$ (because it no longer appears in the expression $(2,c_2,f)$).

We can regard $\mathcal{G}_1$ and $\mathcal{G}_2$ as the children of $\mathcal{G}$ in a *computation tree*. Further strongest postcondition propagation steps can be applied to $\mathcal{G}_1$ and $\mathcal{G}_2$, resulting in the goals $\mathcal{G}_{11}$, $\mathcal{G}_{12}$, $\mathcal{G}_{21}$, and $\mathcal{G}_{22}$. See figure 1 for a depiction of this computation tree.

It is important to note at this point that we have *not* applied constraint simplification during the propagation process, but rather preserved each constraint along every computation path. Thus in figure 1, the constraints in each symbolic state continue to appear verbatim in its successor states. For example, the constraints $c = 0, b1, c2 = c+1$ in $\mathcal{G}_1$ continue to appear in both $\mathcal{G}_{11}$ and $\mathcal{G}_{12}$. Moreover, it is equally important to note that, since the program constraints `b1` and `b2` are independent, all the four sets of states $\mathcal{G}_{11}$, $\mathcal{G}_{12}$, $\mathcal{G}_{21}$, and $\mathcal{G}_{22}$ are in fact *non-empty*. That is, the computation paths depicted in the tree are *feasible*.

We now define the *summarization* of a program fragment as a formula of the form $\mathcal{G}_{entry} \vdash \mathcal{G}_{exit}$, with the meaning that if the program fragment is entered with a state satisfying $\mathcal{G}_{entry}$, then it will be exited with a state satisfying $\mathcal{G}_{exit}$. The usefulness of such a summarization is given by the fact that, whenever the program fragment is entered in a more specific context, for example one satisfying $\mathcal{G}'_{entry} \equiv \mathcal{G}_{entry}, \Phi$, where Φ is a set of constraints, the program fragment will be exited in a context satisfying $\mathcal{G}'_{exit} \equiv \mathcal{G}_{exit}, \Phi$. Thus, a static analysis procedure would make use of the following *subsumption principle*: when the entry point of the summarized program fragment is reached, in a context $\mathcal{G}'_{entry}$ that is *subsumed* by $\mathcal{G}_{entry}$, the corresponding context $\mathcal{G}'_{exit}$ can be immediately computed for the exit point.

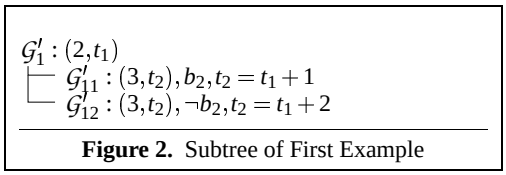

Figure 2. Subtree of First Example

Next, we show how we can derive a summarization for P, where the variables of interest are `count` and `final`. First consider the second `if` statement of P as a stand-alone program fragment, denoted P'. The sub-tree rooted at $\mathcal{G}_1$ is a possible computation tree for P'. Note that the constraints $c = 0, b_1, c_2 = c+1$ appear at all the three nodes of this subtree, Now, consider *deleting* these constraints from all the nodes. We then obtain a valid computation tree for P', depicted in Figure 2. Note that the new tree has the *same shape* as the original subtree[1]. Thus, the constraints $c = 0, b_1, c_2 = c+1$ are, in fact, redundant, since they do not contribute to the input-output relation of P', as described by the computation tree at hand.

More formally, define the *redundant constraints* of a computation tree to be those constraints that appear at all nodes, and whose removal does not lead to new computation paths being added to the tree.

The original subtrees $\mathcal{G}_{11}$ and $\mathcal{G}_{12}$ have now become $\mathcal{G}'_{11} \equiv (3,c_3,f), c_3 = c_2 + 1, b_2, f = c_3$ and $\mathcal{G}'_{12} \equiv (3,c_3,f), c_3 = c_2 + 2, \neg b_2, f = c_3$ respectively, after redundant constraints are removed. Let Φ_1 denote the disjunction of these two sets of constraints: $(c_3 = c_2 + 1, b_2, f = c_3) \vee (c_3 = c_2 + 2, \neg b_2, f = c_3)$

We can now derive our first summarization as:

$$(2,c_2,f) \vdash (3,c_3,f), \Phi_1 \tag{1}$$

Note that our transformation from $\mathcal{G}_1$ to $\mathcal{G}'_1$ preserves all the information relevant to the state transformation performed by P'. Because of this, as it shall be discussed later, the summarization of P' captures its *exact* input-output relationship.

This summarization may be used on subsequent traversals of the program fragment, as long as the context in which the program fragment is entered is *subsumed* by the set of states at the root of the summarizing tree.

Let us continue the process, and now derive a summarization for the entire program fragment P. The main component of finding this summarization is finding the redundant constraints for the computation tree rooted at $\mathcal{G}$. To simplify the language, we shall often refer to the tree rooted at a goal as just simply the goal itself. For example, we may say that the constraint $c = 0$ is redundant at $\mathcal{G}$, when in fact we mean that $c = 0$ is redundant in the tree rooted at $\mathcal{G}$.

Now the set of redundant constraints at $\mathcal{G}$, is in fact the intersection of the sets of redundant constraints at $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively. For $\mathcal{G}_1$, as we have seen above, the redundant constraints are $c = 0, b_1, c_2 = c+1$. For $\mathcal{G}_2$, we now apply a critical step. Instead of analyzing it as we did $\mathcal{G}_1$, we use the fact $\mathcal{G}_2$ is subsumed by the left hand side of the summarization (1). Therefore, any constraint deemed to be redundant at $\mathcal{G}_1$ at the time when (1) was derived, will also be redundant at $\mathcal{G}_2$ (if it appears). Thus, $c = 0$ is redundant at $\mathcal{G}_2$ since it is the only goal that was redundant at $\mathcal{G}_1$ and also appears in $\mathcal{G}_2$. Now, since $c = 0$ is redundant in both $\mathcal{G}_1$ and $\mathcal{G}_2$, it follows that it is redundant at $\mathcal{G}$, and it can be deleted from all the nodes of G.

To simplify the language, we shall denote by $\mathcal{G} - \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$ the symbolic set of states $\mathcal{G}$ from which the (redundant) constraints $\mathcal{C}_1, \ldots, \mathcal{C}_k$ have been removed. Thus, the current root of the computation tree is now $\mathcal{G} - \{c = 0\}$. Now, we can derive the summarization of P informally as:

$$\mathcal{G} - \{c = 0\} \vdash \text{(disjunction of leaf goals in the tree)}$$

Obviously, the disjunction of the leaves of $\mathcal{G} - \{c = 0\}$ is the union between the disjunction of leaves descending from $\mathcal{G}_1 - \{c = 0\}$, and $\mathcal{G}_2 - \{c = 0\}$ respectively. Since both these goals are subsumed by the left hand side of the summarization (1), we can use the subsumption principle given above to compute the required expressions. They are, for $\mathcal{G}_1 - \{c = 0\}$ and $\mathcal{G}_2 - \{c = 0\}$:

$(1,c,f) \vdash (3,c_3,f), b_1, c_2 = c+1, \Phi_1$ and
$(1,c,f) \vdash (3,c_3,f), \neg b_1, c_2 = c+2, \Phi_1$

respectively. Let Φ denote the disjunction of the constraints above: $(b_1, c_2 = c+1) \vee (\neg b_1, c_2 = c+2)$. We can now derive the following summarization for $\mathcal{G}$:

$$(1,c,f) \vdash (3,c_3,f), \quad \Phi, \Phi_1 \tag{2}$$

Note that the use of subsumption here has conjoined a disjunction Φ_1 with another, Φ. Thus, in general, whenever subsumption can be exploited, the size of the expression used for summarizing parent goals grows *linearly*.

We discussed above that removing constraints from a computation tree may, in general, change the shape of the tree. Next, we shall illustrate such a situation. Consider a modification of the example so that $b1 \equiv (count = 0)$. Thus $\mathcal{G}_2$ becomes *false*, since its constraints are unsatisfiable, and the nodes $\mathcal{G}_{21}$ and $\mathcal{G}_{22}$ would no longer appear in the tree (since they reside on infeasible paths). Now we can summarize $\mathcal{G}_1$ as before, and again, propagate this to $\mathcal{G}_2$ to obtain that $f - c_2 \in \{1,2\}, c = 0, \neg b_1, c_2 = c+2$. We however *cannot* perform the next step and declare a summarization of $\mathcal{G} - \{c = 0\}$, as we were able to do previously. This is because the removal of $c = 0$ would affect $\mathcal{G}_2$, which would no longer remain *false*, and therefore subject to possible further expansion.

We have indicated above that whenever subsumption applies and a goal is summarized rather than traversed, that the size of expressions used to summarize its parent node grows linearly. Even so, it may often be the case that even simpler expressions are required in order to achieve the desired level of performance. In such a case, various simplification techniques may be applied to the expression on the left hand side of the summarization. For example, we could project out auxiliary variables from the left hand side (and possibly other, not interesting variables, such as the variables that are local to a procedure). Or, we could simply delete the `if` conditions from the left hand side of the summarization. Applying both these simplifying operations to (2), we get the following much simpler summarization.

$$(1,c,f) \vdash (3,c_3,f), (f - c) \in \{2,3,4\}$$

In this particular case, this simplification to obtain

$$\mathcal{G}_{exit} \equiv (3,c_3,f), (f - c) \in \{2,3,4\}$$

still provides the strongest postcondition of $\mathcal{G}_{entry} \equiv (1,c,f)$ wrt the variables f and c. In general, however, the property that $\mathcal{G}_{exit}, \Phi$ is the strongest postcondition of $\mathcal{G}_{entry}, \Phi$ is no longer true for all constraints Φ. Because of this, the use of summarizations that have been subjected to simplification will result in loss of precision, as we now explain.

The simplification process may introduce spurious computation paths in the tree, but these spurious paths *would only lead to states that are already reachable via feasible paths*. For this reason, the property that $\mathcal{G}_{exit}$ is the postcondition of $\mathcal{G}_{entry}$ is preserved. However, when we add the arbitrary constraint Φ in the mix, the

[1] In general, by removing constraints from $\mathcal{G}_1$, we could add new paths to the tree, since certain formerly infeasible paths may now become feasible.

tree rooted at $\mathcal{G}_{entry}, \Phi$ may have fewer paths than the tree rooted at $\mathcal{G}_{entry}$, as some of the paths may have become infeasible. Yet, some of the states that are at the end of infeasible paths, and which should have disappeared from the $\mathcal{G}_{entry}, \Phi$ tree, remain reachable via the spurious paths introduced by simplifying the summarization. As a result, $\mathcal{G}_{exit}, \Phi$ remains a postcondition of $\mathcal{G}_{entry}, \Phi$, but not necessarily its strongest postcondition.

We now sketch the algorithm for the general case as follows. The algorithm is centered around computing the set of redundant constraints for the computation tree of a given program fragment *P*. Let us assume first that the computation tree at hand is finite. A naïve version of our algorithm would be to perform a post-order traversal of the tree, computing the set of redundant constraints for the current node as the intersection of the sets of redundant constraints of the children. In order to jump-start such an algorithm, we need to compute sets of redundant constraints for the leaves of the tree. In a computation tree, we have two kinds of leaves:

- *Exit leaves:* correspond to the exit point of the program fragment at hand, possibly containing unsatisfiable goals. All the constraints attached to such a node are redundant, since the shape of a one-node subtree cannot change.
- *Non-exit leaves:* contain unsatisfiable goals, which correspond to program points other than the exit point. Here the redundant constraints are those whose removal do not render the goal satisfiable. That is because a satisfiable goal would spawn new feasible paths, and thus change the shape of the tree.

Once the redundant constraints for every node have been computed, the following summarization can be derived for *P*.

$$\mathcal{G} - \mathcal{R} \vdash \bigvee_{l \in L} \mathcal{G}_l,$$

where $\mathcal{G}$ is the root of *P*'s computation tree, $\mathcal{R}$ is the set of redundant constraints at $\mathcal{G}$, L is the set of leaves in the tree, and $\mathcal{G}_l$ is the goal attached to leaf l.

This naïve algorithm can be optimized by computing summarizations for every subtree "on-the-fly", and then, whenever we visit a new node, check whether its goal is subsumed by the right hand side of a previously computed summarization.

Note that our algorithm does not require any user input. In addition to generating summarizations, it is also a generic engine for abstract interpretation. In order to prove that a particular safety property holds, it suffices to inspect the terminal nodes.

A final comment: it is easy to not just to prove a safety property for a bounded program, but to *discover* what they are. Our simple program above indicates that in more general programs where resource usage is modelled using a distinguished variable, we could discover a bound for this variable. We however do not take this point further in this paper.

3. Constraint Transition Systems

This section presents a formal framework for describing an existing abstract interpretation algorithm into which our summarization algorithm fits. 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*. A *term* 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 Proof System

Programs p will be 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 of p, and in fact, it represents the exact *trace* semantics of p. Second, it is an *executable specification* against which an assertion can be checked.

We shall model computation by considering *n system variables* $\tilde{v} = v_1, \cdots, v_n$ 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* system state *(or simply 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 (Constraint Transition System). *A* constraint transition *of p is a formula*

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

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.* □

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.

By similarity with logic programming, we use the term *goal* to denote a literal that can be subjected to an *unfolding process* in order to infer a logical consequence.

DEFINITION 3 (Goal). *A query or* goal *of a CTS is of the form:*

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

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 goal, while any additional variable in Ψ is called an* auxiliary *variable of the goal. Where $\mathcal{G}$ is a goal, we write $\mathcal{G}(\tilde{x})$ to hightlight the primary variables in $\mathcal{G}$.* □

Thus a goal is just like the conclusion of a constraint transition. We say the goal is a *initial goal* if k is the start program point. Similarly, a goal is a *final goal* if k is the terminal program point. For convenience, we assume there is only one transition involving a final program point, and it is of the form $p(k, \tilde{x}) \mapsto true$.

Running a initial goal is tantamount to asking the question: which values of $\tilde{x}$ which satisfy $\tilde{\exists}\Psi(\tilde{x})$ will lead to a goal at the final point? The idea is that we successively reduce one goal to

```
⟨0⟩  i=0;
⟨1⟩  while (i<99) {
⟨2⟩       j=0
⟨3⟩       while (j<99-i) {
⟨4⟩            if (a[j+1]<a[j])
                    { swap(a,j,j+1); t++; }
⟨5⟩            j++
⟨6⟩       }
⟨7⟩       i++
⟨8⟩  } ⟨9⟩
```

Figure 3. Bubble Sort

another until the resulting goal is final point, and then inspect the results.

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

DEFINITION 4 (Proof Step, Sequence and Tree). *Let there be a CTS for p, and let $\mathcal{G} = p(k,\tilde{x}),\Psi$ be a goal for this. A* proof step *from $\mathcal{G}$ may be obtained providing Ψ is satisfiable. It is obtained using a variant $p(k,\tilde{y}) \mapsto p(k_1,\tilde{y_1}),\Psi_1$ of a transition in the CTS in which all the variables are fresh. The result is a goal of the form*

$$p(k_1,\tilde{y_1}),\Psi,\tilde{x}=\tilde{y},\Psi_1$$

Note that this new goal is a false goal if the constraint $\Psi,\tilde{x}=\tilde{y},\Psi_1$ is unsatisfiable.

A proof sequence *is a finite or infinite sequence of proof steps. If a proof sequence ends with a final goal $p(k,\tilde{z}),\Psi$, we say that the variables $\tilde{z}$ are the* final variables *of the constraints Ψ, and we write $\Psi(\tilde{z})$.*

A proof tree *is defined from proof sequences in the obvious way. A tree is* complete *if every internal node representing a goal $\mathcal{G}$ is succeeded by nodes representing* every *goal obtainable in a proof step from $\mathcal{G}$.* □

3.3 Static Abstraction and Closed Proof Trees

Our algorithm accommodates the use of predefined or *static* abstraction. Given an arbitrary abstraction function $\mathcal{A}$ which maps a goal $p(k,\tilde{x}),\Psi$ into a goal $p(k,\tilde{x}),\Psi'$ where $\Psi \models \Psi'$, we say that a proof tree is obtained with static abstraction if the proof tree is obtained as before, except that when a goal $\mathcal{G}$ is obtained via a proof step, we replace $\mathcal{G}$ by the goal obtained by applying the abstraction function $\mathcal{G}$. Note that this allows, in particular, "intermittent" abstraction [9] where abstraction is performed only at selected goals.

A closed proof tree is, intuitively, a complete proof tree where there is no need to perform any more proof steps. For a typical abstract interpretation algorithm, the (static) abstraction function ensures that traversal always leads to a finite closed proof tree. This is done by limiting the number of constraint expressions that may be encountered, and using a simple form of memoization to avoid redundant traversal.

DEFINITION 5 (Closed Proof Tree). *A proof tree, possibly obtained by static abstraction, is* closed *if it is finite and every terminal goal $\mathcal{G}$ is either a final goal, a false goal, or it is subsumed by another goal in the tree.* □

Finally, we relegate the destination of all this work, the proof of a property, to defining that a certain program point is *unsafe*. An unsafe goal is one which involves an unsafe program point. We then say that a closed proof tree is *safe* if it does not contain an unsafe goal.

In Figure 3, we assume α is a paricular constant, and $swap(a,i,j)$ is a primitive function which swaps the i^{th} and j^{th} elements of the array A. Its CTS representation is in Figure 4. The final program point is 10, and the unsafe point 11, indicating that t exceeds a

```
(0,i,j,a,t) ↦(1,i',j,a,t), I'=0
(1,i,j,a,t) ↦(2,i,j,a,t), I<α
(1,i,j,a,t) ↦(9,i,j,a,t), I≥α
(2,i,j,a,t) ↦(3,i,j',a,t), J'=0
(3,i,j,a,t) ↦(4,i,j,a,t), J<α-I
(3,i,j,a,t) ↦(7,i,j,a,t), J≥α-I
(4,i,j,a,t) ↦(5,i,j,a',t'),
        a[j+1]<a[j], a'=swap(a,j+1,j), t'=t-1
(4,i,j,a,t) ↦(5,i,j,a,t), a[j+1]≥a[j]
(5,i,j,a,t) ↦(6,I,j',a,T), j'=j+1
(6,i,j,a,t) ↦(4,i,j,a,t), j<α-i
(6,i,j,a,t) ↦(7,i,j,a,t), j≥α-i
(7,i,j,a,t) ↦(8,i',j,a,t), i'=i+1
(8,i,j,a,t) ↦(2,i,j,a,t), i<α
(8,i,j,a,t) ↦(9,i,j,a,t), i≥α, t=0.
(9,i,j,a,t) ↦t ≤ β, p(10,i,j,a,t).
(9,i,j,a,t) ↦t > β, p(11,i,j,a,t).
```

Figure 4. CTS of Bubble Sort

certain bound β. The resource variable t here captures the time. We assume the program is called with some constraints Ψ on the array elements. Thus an accurate estimate for t is challenging. We shall return to this example in Section 6.

4. The Algorithm

We first deal with the case where the program is bounded, that is, all derivation sequences from the initial goal terminate. Then we show how to compose the analysis over two consecutive program fragments. Next we show how to augment the analysis of a bounded program when it is wrapped inside a simple loop. In the final subsection, we present the general algorithm.

In what follows, we assume we have a family $\mathcal{A},\mathcal{A}_1,\cdots$ of abstraction functions which is used to abstract or generalize a collection of constraints into a fixed set of abstract descriptions. We do not specify which of these functions is to be used in a given situation. In the function REDUNDANT() defined below, we generally seek the *most generalizing* function satisfying the condition of use. In the functions COMMON() and INVARIANT(), however, we generally seek the least generalizing function[2].

We will sometimes specify the use of an abstraction function on a goal $\mathcal{G}$ by means of a procedure to manipulate the syntax of $\mathcal{G}$. Most commonly, we shall speak of "deletion of constraints". For example, if $\mathcal{G}$ were of the form $\phi_1,\cdots,\phi_n,p(0,\tilde{x})$, we may tacitly define the abstraction function $\mathcal{A}$ by saying that it deletes a certain constraint, say ϕ_1, from $\mathcal{G}$. Clearly $\mathcal{A}(\mathcal{G}) \equiv \phi_2,\cdots,\phi_n,p(0,\tilde{x})$ is a more general goal than $\mathcal{G}$, that is, it subsumes $\mathcal{G}$. We will be taking such a function $\mathcal{A}$, which was originally contructed for $\mathcal{G}$, and applying it to a different goal $\mathcal{G}'$. We shall do this only in the following case. $\mathcal{G}'$ is a parent goal of $\mathcal{G}$. That is, the constraints of $\mathcal{G}'$ are an initial subsequence of those in $\mathcal{G}$. In this case, $\mathcal{A}(\mathcal{G}')$ would be obtained by applying those constraint manipulations of $\mathcal{A}$ on the constraints of $\mathcal{G}'$.

We write $\mathcal{A}_1 \otimes \mathcal{A}_2$ to denote the abstraction function which performs a generalization contained in *both* $\mathcal{A}_1$ and $\mathcal{A}_2$. This is used when $\mathcal{A}_1$ and $\mathcal{A}_2$ operate on two sequences ϕ,Ψ_1 and ϕ,Ψ_2 of constraints which share a common initial sequence ϕ. $(\mathcal{A}_1 \otimes \mathcal{A}_2)(\phi)$ is then obtained by replacing individual constraints in ϕ using replacements that are common in $\mathcal{A}_1$ and $\mathcal{A}_2$. We denote by $\top$ the special abstraction function where $(\mathcal{A} \otimes \top) = (\top \otimes \mathcal{A}) = \mathcal{A}$, for all $\mathcal{A}$.

[2] The function INVARIANT() has a comparable status with the abstraction refinement step of CEGAR, which dually, chooses the next most generalizing function.

4.1 The Basic Algorithm for a Bounded Program

We require two key functions.

The function REDUNDANT($\mathcal{G}$) generalizes $\mathcal{G}$ such that it remains *false*. More formally, it is obtained by using the assumed family of abstraction functions in order to transform various constraints (or groups of constraints) into some abstract description. As mentioned above, we generally seek the most general abstract description.

As an example implementation, consider removing each constraint in $\mathcal{G}$ (that is, tranforming it into *true*) if the removal keeps $\mathcal{G}$ *false*. More precisely, let $\mathcal{G}$ be of the form $c_1,\cdots,c_n,p(k,\tilde{x})$, let $\mathcal{C}=\emptyset$, and let $\mathcal{G}'$ initially be $\mathcal{G}$. Now for each $i=1,\cdots,n$, if $\mathcal{G}'-c_i$ is *false*, then let $\mathcal{C}:=\mathcal{C}\cup\{c_i\}$ and $\mathcal{G}':=\mathcal{G}'-c_i$, and continue this loop. The final set of constraints $\mathcal{C}$ will then represent those constraints in $\mathcal{G}$ which, if removed, will nevertheless keep $\mathcal{G}$ *false*. Implementing REDUNDANT() this way means that we simply remove constraints as the abstraction mechanism. But in general, REDUNDANT($\mathcal{G}$) is a mapping of constraints into abstract descriptions.

Next we define that the function COMMON($\tilde{x},\Psi_1,\Psi_2$) returns a constraint that is at least as general as the *projection* of Ψ_1 onto $\tilde{x}$ and Ψ_2 onto $\tilde{x}$. (This projection is quite simple to compute, because the general format of the constraints is a collection of equations obtained from assignments $x_n=f_n(x_{n-1}),\cdots,x_2=f_2(x_1)$, where x_i is chronologically newer than x_{i-1}, and a collection of predicates, each on some variables. We can then rewrite each variable in terms of older variables. Since the oldest variables are the ones to be projected onto, we can rewrite each other variable in terms of these.)

For COMMON($\tilde{x},\Psi_1,\Psi_2$), we again resort to the assumed family of abstraction functions to provide an abstract description, and clearly we seek the best abstract description. For example, COMMON($\{t',t\},t'=t+1,t'=t+2$) would be $t'\leq t+2$ assuming there is such an abstract description. However, a straightforward implementation is simply to collect the common constraints in the projections of Ψ_1 and Ψ_2 onto $\tilde{x}$.

The following is a formalization of the algorithm presented informally in section 2. As before, we consider a goal $\mathcal{G}$ which has not already been summarized, and our objective is to return a summarization of $\mathcal{G}$. The algorithm is largely organized as a post-order traversal of the proof tree, with a crucial step that allows certain branches to be pruned. The algorithm returns, for a given goal $\mathcal{G}$ a pair $(\mathcal{A},\Psi)$ where $\mathcal{A}$ is an abstraction function and Ψ a set of constraints.

ALGORITHM 1. *Let $\mathcal{G}\equiv p(k,\tilde{x}),\Psi$ be a node in the proof tree.*

- *$\mathcal{G}$ is false: Return* (REDUNDANT($\mathcal{G}$), *true*)
- *$\mathcal{G}$ is terminal:*
  If $\mathcal{G}$ is an unsafe goal, Return FAIL; else Return ($\top$, Ψ)
- *There is a summarization $(\mathcal{A}',\Psi')$ for some $\mathcal{G}'$ such that $\mathcal{A}'(\mathcal{G}')$ subsumes $\mathcal{G}$:*
  Let $\mathcal{A}$ be the most general abstraction such that $\mathcal{A}(G)$ remains subsumed by $\mathcal{A}'(G')$.
  Return $(\mathcal{A},\Psi)$
- *$\mathcal{G}$ is not subsumed by a previous summarization:*
  Let $(\mathcal{A}_1,\Psi_1),\ldots,(\mathcal{A}_m,\Psi_m)$ be returned for its descendants. Without losing generality, assume the final variables in the Ψ_i are the same, $\tilde{x}'$. Let ϕ_i, $1\leq i\leq m$, denote the constraints from the one step transition to G_i. Return
  $(\mathcal{A}_1\otimes\ldots\otimes\mathcal{A}_p,\ \text{COMMON}(\tilde{x}\cup\tilde{x}',\phi_1\wedge\Psi_1,\cdots,\phi_m\wedge\Psi_m))$

Note that GENERALIZE() is used for efficiency, to deal with the exponential explosion caused by branches in expanding symbolic traces of a bounded program. In contrast, COMMON() is used to represent the *result* of analysis of a bounded program, and this is only used in case this bounded program is embedded in a larger program needs to be analyzed.

The following theorem states that the above algorithm is exact.

THEOREM 1. *Let P be a bounded program, p its CTS representation, and $\mathcal{G}$ the initial goal. The algorithm run on $\mathcal{G}$ returns $(\mathcal{A},\Psi)$ if and only if all derivations from $\mathcal{A}(\mathcal{G})$ in p are safe. Further, Ψ is a postcondition of $\mathcal{A}(\mathcal{G})$.* □

4.2 The Algorithm for Two Consecutive Bounded Programs

Let the CTS p and the CTS q represent two programs P and Q respectively. Let $\mathcal{G}$ denote an initial goal of p, and let 0 denote the initial program point of q.

Suppose the bounded algorithm on $\mathcal{G}$ returns (A,Ψ). Let $\tilde{x}$ denote fresh primary variables for the atom $q(0,\tilde{x})$, and θ rename the final variables in Ψ into $\tilde{x}$. Now run the bounded algorithm again, this time on $\Psi\theta,q(0,\tilde{x})$, and say the return value is (A_2,Ψ_2). The return value of the program "$P;Q$" may now be returned as $(\mathcal{A},\Psi\wedge\Psi_2)$.

4.3 The Single-Loop Algorithm

A loop is defined by (a) a CTS p representing its body program fragment, and (b) an exit condition *exit*(). We now extend the algorithm above to deal with a single "repeat-until" loop.

A key function needed here is INVARIANT($\mathcal{G},\Psi$). Let the goal $\mathcal{G}(\tilde{x})$ represent the entry to the loop body, and let it have a derivation sequence giving rise to constraints $\Psi(\tilde{z})$ at the end of the loop body. Intuitively, INVARIANT() generalized the constraints in $\mathcal{G}$ so that what remains are constraints which are "invariant" through the loop. More formally, INVARIANT($\mathcal{G}(\tilde{x}),\Psi(\tilde{z})$) returns a generalization $\mathcal{G}'$ of $\mathcal{G}$ that is, $\mathcal{G}'(\tilde{x})$ subsumes
$\mathcal{G}'(\tilde{z}),\Psi,\neg exit(\tilde{z})$.

As with REDUNDANT(), there is a straightforward implementation of INVARIANT(G,Ψ): remove each individual constraint that is not invariant through the loop. More precisely, let θ rename $\tilde{x}$ into $\tilde{z}$. We say that an individual constraint c is invariant through Ψ if $p(k,\tilde{z}),\Psi,\neg exit(\tilde{z})\models c\theta$.

ALGORITHM 2. *Input initial goal $\mathcal{G}\equiv p(0,\tilde{x}),\Psi_0$ and exit condition $exit(\tilde{x})$*

1. *Let $\mathcal{A}'$ be $\bot$.*
2. *Perform bounded program analysis on $\mathcal{G}$ obtaining $(\mathcal{A},\Psi(\tilde{z}))$.*
3. *Let $\mathcal{G}'$ be the goal $p(0,\tilde{z}),\Psi_0,\Psi(\tilde{z})$.*
   If $\mathcal{G}$ subsumes $\mathcal{G}'$, Return $(\mathcal{A}',\Psi(\tilde{z})\wedge exit(\tilde{z}))$
4. *Let $\mathcal{A}'$ be* INVARIANT($\mathcal{G},\Psi$).
   Restart step 1 with $\mathcal{A}'(\mathcal{G})$ in place of $\mathcal{G}$.

A key property of this algorithm is that all invariant constraints in the calling context Ψ are *automatically propagated* through the loop. We exemplify this below.

4.4 The General Algorithm

We now combine the above three algorithms for the general case. It suffices to describe how to embed, in a bounded program, a single loop.

Suppose we want to embed a loop in between program points k and $k+1$ in a CTS p for a bounded program. We now introduce a special rule called a *loop rule*, of the form:

$$p(k,\tilde{x}):-loop(q,exit(\tilde{x}),\tilde{z}),p(k+1,\tilde{z}).$$

where the CTS q represents the loop-body, the predicate q is not defined in terms of the predicate p, the term $exit(\tilde{x})$ denotes the exit condition of the loop, and finally, the variables $\tilde{z}$ are fresh (and intended to represent the final variables of the analysis of q.

We now present our main algorithm as a generalization of the unbounded algorithm. In this generalization, we shall be calling the single-loop algorithm when we encounter a goal from a loop rule. We assume, by recursive reasoning so that we can deal with arbitrarily nested loops, that the single-loop algorithm will in turn call this main algorithm (as opposed to the bounded algorithm).

ALGORITHM 3. *Input $\mathcal{G}$.*

If $\mathcal{G}$ is not a loop goal: proceed as in the bounded algorithm. Otherwise, let $\mathcal{G}$ be of the form

$$\Psi, loop(q, exit(\tilde{x}), \tilde{z}), p(k+1, \tilde{z})$$

Run the single loop algm on $\Psi, q(0, \tilde{x})$ with the exit condition $exit(\tilde{x})$. Obtain the pair $(\mathcal{A}, \Psi')$, and let the final variables in Ψ' be $\tilde{z}$. Now (recursively) analyze using the (general) algorithm on the new goal:

$$\mathcal{A}(\Psi), \Psi', p(k+1, \tilde{z})$$

and obtain the new result $(\mathcal{A}', \Psi'')$.
Return $(\mathcal{A}, \Psi' \wedge \Psi'')$.

We finally state what is obviously needed.

THEOREM 2. *Let P be a program, p its CTS representation, $\mathcal{G}$ the the initial goal. The general algorithm on $\mathcal{G}$ terminates, and if it does not return FAIL, the program P is safe. Further, the return value $(\mathcal{A}, \Psi)$ is such that $\mathcal{A}(G)$ has a postcondition Ψ.* □

We finally mention that the user may apply an arbirtrary abstraction function at any point. To so on a goal, say to apply $\mathcal{A}$ on $\Psi, p(k, \tilde{x})$, simply means the following. Let $\Psi = \phi_1, \cdots, \phi_n$. (a) remove each constraint ϕ_i where $\phi_i \models \mathcal{A}(\Psi)$ from Ψ, and (b) continuing the analysis process with $\Psi, \mathcal{A}(\Psi), p(k, \tilde{x})$ where this Ψ is what results from step (a).

Consider a skeleton of the bubblesort program:

```
⟨0⟩ repeat
⟨1⟩      j = 0
⟨2⟩      repeat
⟨3⟩          t++; j++
⟨4⟩      until (j = i)
⟨5⟩      i++
⟨6⟩ until (i = n) ⟨7⟩
```

Consider a top-level input constraint of $t \geq i$, and we prove automatically that $t \geq i$ at the end. First consider the single-loop algorithm applied to the outer loop at point $\langle 0 \rangle$. We now analyze its body at $\langle 1 \rangle$.

Consider the bounded loop algorithm on this body of three statements. The first, at $\langle 1 \rangle$, leads us to analyze the second statement $\langle 2 \rangle$ with the new context $t \geq i, j = 0$. We now analyze the inner loop body at $\langle 3 \rangle$.

We easily get the summarization $(\mathcal{A}, t' = t+1, j' = j+1)$; we ignore $\mathcal{A}$ henceforth. Using step 3 of the algorithm, we check if $p(3, i, j, t)$ subsumes $p(3, i, j', t'), t \geq i, j = 0, t' = t+1, j' = j+1, j' \neq i$. It does not, and we apply INVARIANT() which now deletes $j = 0$ so that step 3 of the single loop algorithm succeeds. We have now determined the summarization $(\mathcal{A}_2, t' > i, t' = t+1, j' = j+1, j' = i)$, where $\mathcal{A}_2$ simply deletes $j = 0$, for the inner loop.

Returning now to the third statement of the outer loop, we propagate the above summarize formula through point $\langle 5 \rangle$ to obtain $t \geq i, t' = t+1, j' = j+1, j' \neq i, i' = i+1$, which can be simplified into $t' \geq i'$ by projecting onto t' and i'.

We have now completed the inner loop body with this formula as a summarization. We are now back at step 3 of the single loop algorithm, and because the subsumption test passes (ie: $p(1, i, j, t), t \geq i$ subsumes p(1, i', j', t'), $t' \geq i', i' = n$), we are done.

Next consider another example:

```
⟨0⟩ repeat
⟨1⟩  i++
⟨2⟩  t = t + i
⟨3⟩ until (i = n) ⟨4⟩
```

We now demonstrate the use of a user-given invariant $t = 0.5i^2$ at point $\langle 1 \rangle$. We assume a new top-level constraint $0 = i = t < n$ and we prove automatically that $t = 0.5n^2$ at the end. We start with the single loop algorithm, and analyzing the inner loop body at $\langle 1 \rangle$. This point is marked with a user-defined abstraction $t = 0.5i^2$ and so we delete the contraints here which imply this; that is, we delete $0 = i = t$. We now proceed with the analysis of point $\langle 2 \rangle$, now with the constraint $t = 0.5i^2$. We eventually get to the bottom of the loop with the constraints $t = 0.5i^2, i' = i+1, t' = t+i'$. The rest of the proof, that this constraint augmented with $i \neq n$ at (t', i') implies the user-defined invariant at (t, i), is straightforward.

5. Tuning the Algorithm

A key strength of the algorithm is that it is easily *tunable*. The key technical concepts permitting this is the use of summarizations, which in turn provides for compositional reasoning. That is, program fragments may be analyzed with different specializations of the technique, and pieced together using summarizations. For example, while the basic algorithm is exact on a loop-free program, a judicious partition of the program into two fragments can signifcantly improve performance while preserving a proof. One could choose the partition point where the preceding program fragment terminates after culminating a complex computation into a simple set of constraints which can appeart in the summarization of the fragment. Similarly, for loops, one could use different notions of obtaining an invariant so that the loop need not be unrolled.

Now, for a single unbounded program fragment, the essential challenge is that the analysis of a bounded program generally requires consideration of an exponential number of symbolic traces. The standard approach is to use a static abstraction function and perform search by considering abstractly similar states or traces as one. The problem is, of course, that the choice of the abstraction function is not necessarily designed for the program at hand, and hence the function may not give rise to an efficient proof. As mentioned above in section 1.1, dynamic abstract interpretation methods apart from CEGAR are not well developed, and they are not compositional. Our technique of dynamic summarizations is an effective optimization of this exponential problem.

A key component in computing the summarization is the generalization function REDUNDANT() which keeps a goal *false*. A key component of the single-loop algorithm is also a generalization function INVARIANT(), but this time it computes a loop invariant. In both these cases, there are straightforward and efficient algorithms based upon constraint deletion. Such a standard algorithm may not always be appropriate. We then could resolve to choosing from a family of abstract functions, for example, a family specified by predicate abstraction [5]. The important point is that one could design this family of functions with the problem domain at hand. We should mention that designing the abstraction family is also applicable in standard approaches using abstraction interpretation.

For loops, our use of a loop invariant is in fact less restrictive than is standard. In particular, note that constraints which are invariant will be propagated through the loop, regardless of abstraction. For example, if we were to analyze the program fragment `while (i < n) do i++ ; j++2 end` with a context $i = 0, i < j$, then the (standard) abstraction function would delete the constraint $i = 0$. It will not, however, delete the second constraint $i < j$; thus this

constraint is available for subsequent use. Note that while this constraint can be (manually or automatically) proven invariant by some generalization method, the point here is that we need not know whether or not it *in advance*.

Finally we address the postcondition computed in a summarization by means of the COMMON() function. This essentially involves the abstraction of two constraint sequences into one. Here the idea to generalize as little as possible while using a compact representation. Again, this is tunable if one has an idea about what is essential about the output of a program fragment. An important class of examples is that only (the abstract values of) certain variables are essential. In this case, the computation of COMMON() would operate on the projection of the input constraints onto these variables. After projection, one then needs a compact representation of the disjunction of the resultant constraints, something that abstraction functions are often designed to do.

6. Experimental Evaluation

In this section, we evaluate only bounded programs to demonstrate the search reduction capabilities of dynamic summarizations. This is the primary area of concern for efficiency. Loops are analysed by composition of bounded programs, and so their performance is secondary.

We implement example programs by direct translation into the CLP($\mathcal{R}$) programming language [8]. Two critical features of CLP($\mathcal{R}$) are its meta-level facilities [6], and its projection algorithm [7]. The latter needs to deal with the basic problem in CTS systems of dynamic variable creation. For example, one derivation resulted in tens of thousands of constraints over a few hundred thousand auxiliary variables, but CLP($\mathcal{R}$) ran in approximately 0.2 seconds. We also augmented the CLP($\mathcal{R}$) system with a memoization mechanism, storing the summarization result of each encountered goal, in order to perform subsumption checking. Finally, we implemented our basic algorithm presented in Section 4.1 to perform the experiments. The REDUNDANT function which we implemented uses constraint deletion at every *false* node.

Our first example, analyzes the value of t in the bubblesort program in Figure 3, shows idealized behavior. The answer, as is well known, is given by that fact that there are at most $\alpha(\alpha+1)/2$ swaps. The symbolic search space (which is already considerably smaller than the concrete space) is exponential in this number and hence unmanageable. Using dynamic summarizations, the search space we obtain is in fact linear in the array size α.

Now, for this trivial example, there may be other algorithms which can use the simple facts that each iteration increments t by at most 2 and that the longest path has length $\alpha(\alpha+1)/2$. However, in general there would be ad-hoc input constraints, For example, suppose the array contained only binary elements. Then a simple algorithm would be inaccurate: for example, the answer is 9 instead of 15 for an array of size 6. We are unaware of any algorithm which could solve this kind of problem efficiently.

To further demonstrate the handling of ad-hoc contexts we ran bubblesort again, this time modelling the underlying microarchitecture in order to produce ad-hoc but realistic contexts. We choose a "data cache" microarchitecture as follows. At any time, the data cache stores array elements $a[3i]$, $a[3i+1]$, and $a[3i+2]$ for some i. When there is a need to load a new set of array elements into the cache, we give an additional cost of 1 to array element comparisons.

Results are shown in Table 1 with time in seconds, obtained using Linux 2.4.22 OS on a Pentium 4 2.8GHz processor with 512Mb RAM. Note that the number of nodes is linear in the square of array size (which in turn is linear in the maximal path length), in both versions[3].

Next we ran a few random programs such as the ADPCM encoder (41 lines), and decoder (27 lines) [13] (which are similar to a large class of embedded streaming applications). Here we simply assumed that each C statement consumes 1 time unit. In both programs, we also modelled an instruction cache which can store 8 statements, and a cache miss would consume 10 time units. For these examples, we modified our algorithm so that not just proves but discovers the exact bound. Finally we ran an iterative square root algorithm [15] and janne_complex [4, 10] examples. The results are shown in Table 2.

In all these examples, dynamic summarization produced significant improvements.

7. Concluding Discussion

We presented an analysis technique for structured sequential programs based upon abstract reasoning over symbolic traces. Its first component deals with bounded programs, and the novelty here was the use of dynamic summarizations to succinctly describe arbitrary program fragments as relations. A summarization serves two purposes. First, by means of abstraction on demand, its context of use can be more general than in the original invocation of the program fragment. This is the key idea that permits a reduction of the search space. Second, a summarization permits the postcondition of the program fragment, which is in general a large disjunction of constraints, to be compactly representation. These two purposes allow our technique the crucial feature of compositionality.

We then extend the bounded program technique to the general case. The important contribution here concerns how a loop invariant is obtained. We presented a method where only certain parts of a Hoare-style is required to be given, either manually, or via the family of abstraction functions considered. Our technique will automatically preserve invariant constraints in the sense that these dynamically encountered constraints need not be accomodated by the abstraction function in order to be propagated through the loop.

We demonstrated a standard implementation of the three core functions as follows. For generalization, we simply removed constraints one at a time as long as some condition holds (REDUNDANT()), or until some condition holds (INVARIANT()). For summarization of postconditions, we simply collected the common constraints in the intersection of two constraints. Even with these trivial implementations, we obtained an automatic analyzer which was useful.

In the end, we implemented a technique based on the philosophy of performing analysis with as much precision as possible, until efficiency is at stake, in other words, "abstraction on demand". Being compositional and having isolated its core functions of generalization and succint representation of postconditions, we believe the technique is easily tunable.

[3] Standard benchmarks [15, 10] usually fix the array values and consider analysis only on a single execution path.

Problem	Array Size	No Summarization **Nodes**	(Time)	W/ Summarization **Nodes**	(Time)
Normal	5	**1606**	(9.98)	**58**	(0.06)
	10	∞		**218**	(1.41)
	15	∞		**478**	(10.71)
	20	∞		**838**	(42.45)
	25	∞		**1298**	(134.48)
	30	∞		**1858**	(349.73)
	35	∞		**2518**	(824.72)
Cached	5	**2233**	(20.46)	**88**	(0.20)
	10	∞		**336**	(5.03)
	15	∞		**798**	(45.02)
	20	∞		**1410**	(216.60)

Table 1. Bubble Sort

Problem	No Summarization **Nodes**	(Time)	W/ Summarization **Nodes**	(Time)
Encoder	**494**	(1.22)	**266**	(0.69)
Decoder	**344**	(0.46)	**164**	(0.30)
Encoder (Cached)	**494**	(1.63)	**266**	(0.95)
Decoder (Cached)	**344**	(0.56)	**164**	(0.39)
Square Root	**923**	(5.96)	**253**	(1.91)
Janne_complex	**1517**	(24.25)	**683**	(5.8)

Table 2. Some Random 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] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, September 2003.

[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] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *3rd Euro-Par*, volume 1300 of *LNCS*, pages 1298–1307. Springer, 1997.

[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. 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.

[8] 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.

[9] J. Jaffar, A. E. Santosa, and R. Voicu. A CLP method for compositional and intermittent predicate abstraction. In E. A. Emerson and K. S. Namjoshi, editors, *7th VMCAI*, volume 3855 of *LNCS*, pages 17–32. Springer, 2006.

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

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

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

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

[14] 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.

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