A Framework for Combining Analysis and Verification

Nevin Heintze
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
nch@research.bell-labs.com

Joxan Jaffar **Răzvan Voicu**
School of Computing
National University of Singapore
Republic of Singapore 117543
{joxan,razvan}@comp.nus.edu.sg

Abstract

We present a general framework for combining program verification and program analysis. This framework enhances program analysis because it takes advantage of user assertions, and it enhances program verification because assertions can be refined using automatic program analysis. Both enhancements in general produce a better way of reasoning about programs than using verification techniques alone or analysis techniques alone. More importantly, the combination is better than simply running the verification and analysis in isolation and then combining the results at the last step. In other words, our framework explores synergistic interaction between verification and analysis.

In this paper, we start with a representation of a program, user assertions, and a given analyzer for the program. The framework we describe induces an algorithm which exploits the assertions and the analyzer to produce a generally more accurate analysis. Further, it has some important features:

- it is flexible: any number of assertions can be used anywhere;
- it is open: it can employ an arbitrary analyzer;
- it is modular: we reason with conditional correctness of assertions;
- it is incremental: it can be tuned for the accuracy/efficiency tradeoff.

Keywords: *program verification, program analysis, abstract interpretation*

1 Introduction

We present a general framework for combining program verification and program analysis. By program analysis, we mean an automated procedure for extracting information about a program's runtime behavior, such as an analysis component of an optimizing compiler. For example, the analysis might compute information about the ranges of program values, aliasing between program variables, or else provide analysis support for a common sub-expression elimination transformation. Such analyses are necessarily approximate, and are expected to be efficiently computable. Our focus is on analyses that can be formulated using abstract interpretation [3], and viewed as finite fixed-point computation.

By verification, we mean a procedure for reasoning exactly about a program behavior, such as some form of Hoare-logic proof-checker. We assume that the procedure takes the following form: each statement of the program comes with a program assertion and the procedure verifies these assertions by checking whether they are locally consistent. This is a very primitive notion of verification. Most real verification systems provide considerable support for a user to interactively prove program assertions.

One way to combine verification and analysis is at a system level: we view both the analysis and verification as black boxes and provide some external common language for them to communicate. Examples of this are mentioned in the subsection on related work below.

In contrast, our framework goes beyond system level integration of analysis and verification and attempts a conceptual integration using fixed-point operators on a representation of the program's collecting semantics. The result is a much tighter two-way interaction between analysis and verification. In the end, the framework enhances program analysis because it takes advantage of user assertions, and it enhances program verification because assertions can be refined using automatic program analysis. Both enhancements in general produce a better way of reasoning about programs than using verification techniques alone or analysis techniques alone. More importantly, the combination is better than simply running the verification and analysis in isolation and then combining the results at the last step. In other words, our framework explores synergistic interaction between verification and analysis.

Consider the following example program, and suppose we want to infer that $x = y$ just after the while loop. The annotation $\{x + y = x0 + y0\}$ indicates an assertion supplied by the programmer at the end of the while-loop body.

```
x = x0 = 0
y = y0 = 4
while (x < y) do
      x = x + 1
      y = y - 1
      x0 = x0 - 2
      y0 = y0 + 2
      {x + y = x0 + y0}
endwhile
```

Hoare-style verification (see eg [7]) of this loop with the invariant indicated would yield the property $\{x + y = x0 + y0 \wedge x \geq y\}$ at the end of the loop, which does not imply $x = y$. The verification component of our algorithm, which propagates invariants from program points to program points using strongest post-condition style reasoning, yields a slightly stronger property at the end of the loop: $\{x + y = x0 + y0 \wedge x - 1 < y + 1 \wedge x \geq y\}$. However this still does not imply $x = y$. The problem is that the invariant is not strong enough. Now, suppose that we combine the strongest post-condition reasoning with an abstract interpreter equipped with the classic domain $\{\perp, \top, odd, even\}$ for reasoning about odd and even numbers (see eg. [13]). Abstract interpretation using this domain infers that `x0, y0` are even at the end of the loop, although it cannot infer anything about `x, y`. Combining the results from the Hoare-style reasoner with this abstract interpretation results gives $\{x+y = x0+y0 \wedge x-1 < y+1 \wedge x \geq y \wedge even(x0) \wedge even(y0)\}$, which now does imply $x = y$.

This example illustrates the potential of cooperation between a verification engine and an analysis engine. However, it only shows how running verification and then analysis can give better results than either on its own. In fact the verification and analysis process are not independent – results from one can be used to improve the results of the other and vice-versa. This interaction suggests a process consisting of a sequence of alternating analysis and verification phases.

Related Work

In the literature, there is no systematic nor formal framework, in a general setting, where program verification and program analysis work co-operatively together. Cousot & Cousot [4, 6] provided basic understanding about some relationships between assertions and abstract values. They showed that assertions can be viewed, in a straightforward manner, as abstract values. Thus they establish a common ground between program verification and abstract interpretation, which we build upon in this paper.

A number of works explore ways to combine verification and analysis at the system level. For example, [17, 18] show how to build an analysis system that automatically infers useful program invariants (the propagation techniques used in [18] are similar to the propagation used in the main part of our algorithm). This analysis is then used as a black box by the PVS theorem prover [15]. In a similar spirit, [16] use the theorem prover PVS to guess, refine and verify desired invariants. We note that this work addresses transition systems, as opposed to a general programming language. A related example involves hooking up model checkers to theorem provers [8]. The interaction in these works is typically just one-way, ie: the theorem prover calls the model checker or analyzer and then utilizes the results.

There are also works which combine different inference procedures so that the resulting inference procedure is more than a disjoint sum of the component procedures. For example, Nelson and Oppen [14] showed that propagating certain equality information between decision procedures for two different theories results in a decision procedure for the combined theory. Aiken's BANE system [1], on the other hand, is an example of how to provide low level interaction between two different analysis algorithms – specifically, they show how to combine term and set constraints.

What sets our work apart is that it provides for low level interaction between an analyzer and a verifier. A verifier determines if given assertions are correct, that is, the assertions contain the real runtime values. An analyzer, on the other hand, assumes nothing is given; it serves to produce correct assertions. Thus a verifier *checks* against given assertions but does not infer, while an analyzer infers but does not *check*. This is the essential reason why combining a verifier and an analyzer is not a straightforward matter of combining two inference procedures.

The primary effort in this paper is an inference procedure which *contains* a program verifier. Given any amount of assertions, it is able to infer other assertions, and is able to determine if assertions are correct *conditionally* on the given assertions, or unconditionally, or any combination of this. Consequently, it is combinable with another inference procedure such as abstract interpretation. The main benefit is, once again, that since these two components have different inference powers, the result of alternating them provides an inference procedure which is in general better than the disjoint sum of the power of each one employed alone.

Overview of the Paper

In this paper, we start with a representation of a program, user assertions, and a given analyzer for the program. The framework we describe induces an algorithm that exploits the assertions and the analyzer to produce a generally more accurate analysis. Its main elements are: constraints, equational representation of operational semantics, a notion of exact propagation, and the use of a standard analyzer which is a parameter to the framework.

In summary, our algorithm satisfies the following important criteria:

- It is possible to annotate the program with assertions. The underlying philosophy is that certain conclusions can only be obtained with user help. Verifiers have this property; analyzers do not.
- Any number of assertions is acceptable, and they can be placed anywhere. This is not true for verifiers, and not applicable to analyzers.
- The framework is built upon the notion of *conditional analysis*, that is, the correctness of the analysis is dependent on the correctness of some or all of the user assertions. This provides the ability to reason about a module whose context is (yet) unknown.
- There exists a facility for exact reasoning about propagating assertions throughout certain parts of the program. This is crucial for the practical task of creating correct user assertions, because the usefulness of a user assertion is intimately related to how it can be used by the analyzer. Verifiers employ weakest precondition

and/or strongest postcondition operations to facilitate such exact reasoning; in analyzers, this is generally not available.

- The analysis process is incremental. It is possible to define different criteria for termination. This criteria can help to limit the amount of work needed by the analyzer. In fact, our algorithm can be stopped at any time and still yield correct information.

- There are many important analysis algorithms. Our algorithm embeds an arbitrary analyzer as a procedure. In other words, our framework is an *open* system.

2 An Equational Representation of Programs

Consider a flow-graph style representation of the collecting semantics of a first-order programs over a lattice $\mathcal{L}$ [4]. A flow-graph program P consists of a collection of *program points* $1, 2, \ldots, N$, and a directed graph G_P whose nodes are $P's$ program points and whose edges (i, j) are labeled with a "transfer" function $T_{(i,j)} : \mathcal{L} \to \mathcal{L}$ that describes the flow of information from point i to point j. Each $T_{(i,j)}$ is monotonic and continuous.

Now, let $pred(i)$ be the predecessor nodes of i, and define the following equations, $i = 1, N$:

$$C_i = \bigcup_{j \in pred(i)} T_{(j,i)}(C_j)$$

The least solution of these equations defines the *collecting semantics* $CS(P)$ of the program P.

Our algorithm is based on this standard setting, but the above definitions are too abstract. The rest of this section refines these definitions with the necessary details.

Consider a first-order language made up from *program variables* V, a *signature* Σ of function and relation symbols (including equality) and their arities. (Σ, V) terms and constraints are defined from Σ and V in the usual way. We also assume that the language is equipped with a domain of discourse $\mathcal{D}$ and an pre-interpretation of Σ, which gives a meaning to the symbols in Σ. This gives a meaning to ground[1] terms and closed constraints.

In addition to V, we shall assume that there is a corresponding set of *annotated variables* $V\!\!\downarrow$, where each $x \in V$ has a corresponding annotated version $x\!\!\downarrow$ in $V\!\!\downarrow$. In what follows, we shall consider expressions constructed from Σ and both V and $V\!\!\downarrow$. Where e is an expression involving variables only from V, we write $e\!\!\downarrow$ to denote the expression obtained from e by replacing each occurrence of a variable in e by its annotated counterpart.

A *constraint transformer* (c, i) consists of a $(\Sigma, V \cup V\!\!\downarrow)$ constraint c and program point i, called the *dependency* of c. We include a special program point "*" to abstractly denote the entry point of the program. Intuitively, transformers such as $(c, *)$ represent entry conditions[2].

Definition 1 An *equational representation* $\mathcal{E}(P)$ of a program P is a collection of equations, exactly one for each program point, each of the form:

$$C_i = t_1 \cup \cdots \cup t_{m_i}$$

where i ranges over the program points, and each t_j, $1 \leq j \leq m_i$ is a constraint transformer. □

Figure 1 contains an example. Note the constraint $20 \leq x \leq 50$ associated with the first program point; this indicates the input condition for the program segment. We shall omit consideration of how to obtain equational representations in general; instead we refer to [10, 11].

For illustration purposes, we use in our examples, a basic imperative language made up using program variables V, (Σ, V) terms and constraints, assignment, conditional, jump and loop statements, all defined in the usual manner.

Note that Definition 1 is in fact a special case of the more general formulation in [10, 11]. The equations there allowed to contain multiple constraint transformers in place of each t_i, and this accommodates more complicated operational models such as those of logic programs. See [9] for general methods for obtaining representations for functional programs and (top-down and bottom-up) constraint logic programs.

Next consider interpretations of equational representations. Let θ denote a mapping from V into $\mathcal{D}$; call such mappings *environments*. Let Θ denote sets of such mappings. The meaning $[\![c]\!]$ of a constraint c, which is induced by the pre-interpretation of Σ, is defined in the obvious way: $\{\theta : \theta(c) \text{ is } \mathit{true}\}$. We extend this notation as follows: if $\mathcal{C}$ is a set of constraints, then $[\![\mathcal{C}]\!] = \{\theta \in [\![c]\!] : c \in \mathcal{C}\}$. Similarly, given a sequence of such sets $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, $[\![(\mathcal{C}_1, \ldots, \mathcal{C}_n)]\!] = ([\![\mathcal{C}_1]\!], \ldots, [\![\mathcal{C}_n]\!])$. The notation $\tilde{\mathcal{C}}[i]$ projects on the i^{th} component of $\tilde{\mathcal{C}}$.

For such sequences $\tilde{\mathcal{C}} = (\mathcal{C}_1, \ldots, \mathcal{C}_n)$ and $\tilde{\mathcal{C}}' = (\mathcal{C}'_1, \ldots, \mathcal{C}'_n)$, we write $\tilde{\mathcal{C}} \subseteq \tilde{\mathcal{C}}'$ to mean $\mathcal{C}_i \subseteq \mathcal{C}'_i$ for all $1 \leq i \leq n$. Similarly, for sequences $\tilde{\Theta}$ of sets of environments. Finally, for constraint sets $\mathcal{C}$ and $\mathcal{C}'$, $\mathcal{C} \bowtie \mathcal{C}'$ is the constraint set containing $c \wedge c'$ for all $c \in \mathcal{C}$ and $c' \in \mathcal{C}'$. Similarly, $(\mathcal{C}_1, \ldots, \mathcal{C}_n) \bowtie (\mathcal{C}'_1, \ldots, \mathcal{C}'_n)$ denotes $(\mathcal{C}_1 \bowtie \mathcal{C}'_1, \ldots, \mathcal{C}_n \bowtie \mathcal{C}'_n)$.

Now, an interpretation $\mathcal{I}$ of each variable C_i in the equations is a set of environments. The meaning of constraint transformer (c, i) is defined as follows:

$$\mathcal{I}((c, i)) = \{(\theta\!\!\downarrow)(c) : \theta \in \mathcal{I}(C_i)\}$$

where $\theta\!\!\downarrow$ is like θ, except that it maps $V\!\!\downarrow$ instead of V i.e. $(\theta\!\!\downarrow)(X\!\!\downarrow) = \theta(X)$, and $\theta\!\!\downarrow$ is the identity elsewhere. In case i is the special program point "*", then $\mathcal{I}(C_i)$ is defined to be the set of all environments. Finally, $\mathcal{I}$ is a *solution* of an equation $C_i = t_1 \cup \cdots \cup t_{m_i}$ if

$$\mathcal{I}(C_i) = \mathcal{I}(t_1) \cup \cdots \cup \mathcal{I}(t_{m_i}).$$

We define that the collecting semantics $CS(P)$ of a program P is least solution of the equations $\mathcal{E}(P)$. Using our definition of interpretation, we can view $\mathcal{E}(P)$ as a fixpoint operator $\mathcal{F}_P$ on interpretations, in the standard way: if $C_i = t_1 \cup \cdots \cup t_{m_i}$ is the equation for C_i in $\mathcal{E}(P)$ then

$$\mathcal{F}_P(\mathcal{I})(C_i) \stackrel{\text{def}}{=} \mathcal{I}(t_1) \cup \cdots \cup \mathcal{I}(t_{m_i}).$$

The operator $\mathcal{F}_P$ is monotonic and continuous and hence it has a least fixpoint $\mathit{lfp}(\mathcal{F}_P)$. It follows from definitions that $\mathit{lfp}(\mathcal{F}_P) = CS(P)$.

Note that we have presented a "ground semantics" for the constraints in programs. This need not be so. We could define a semantics for reasoning about intensional properties

[1] Variable free.

[2] In general there are several of these.

```
while x<100 do
      x := x+1
endwhile
```

$$
\begin{aligned}
C_1 &= (20 \le x \le 50, *)\\
C_2 &= (x < 100 \wedge x = x^{\downarrow}, 1) \cup (x < 100 \wedge x = x^{\downarrow}, 3)\\
C_3 &= (x = x^{\downarrow} + 1, 2)\\
C_4 &= (x \ge 100 \wedge x = x^{\downarrow}, 1) \cup (x \ge 100 \wedge x = x^{\downarrow}, 3)
\end{aligned}
$$

Figure 1: Example program and its equational representation.

of constraints. For example, the constraints could be term equations to be unified, and the intensional properties are the variable bindings entailed by the unification process. For the rest of this paper we maintain the ground semantics for simplicity of presentation.

Consider the equations, using a signature of $\{+, 0, 1, 2, \cdots\}$ with the usual meaning:

$$C_1 = (x = 0, *), \quad C_2 = (x = x^{\downarrow}, 1) \cup (x = x^{\downarrow} + 2, 2)$$

Then the following interpretation $\mathcal{I}$ is a solution (and in fact the least one) for the equations.

$$\mathcal{I}(C_1) = \{x \mapsto 0\}, \quad \mathcal{I}(C_2) = \{x \mapsto 0, x \mapsto 2, x \mapsto 4, \cdots\} \ \Box$$

We conclude this section by showing that $\mathcal{F}_P$ has more than one fixpoint in general. Consider the program below. The indicated constraint sets at program points a, b and c give rise to a fixpoint of $\mathcal{F}_P$. But the least fixpoint of $\mathcal{F}_P$ has $\{x = 0\}$, $\{x = 1\}$ and $\{x = 1\}$ at those points respectively.

```
x = 0
while (x ≠ 1) do
      a : { x = 0 } ∪ { x = 2 }
      x = x + 1
      b : { x = 1 } ∪ { x = 3 }
      if (x = 3) then x = x - 1
      c : { x = 1 } ∪ { x = 2 }
endwhile
```

3 An Operator for Exact Propagation

In order to define an analysis algorithm, we need appropriate representations of the underlying semantics. In this paper, we use constraints for representations; the formal definitions are inherited directly from the definitions of the programming language (possibly extended with existential quantification). The object of primary interest is a notion of *program descriptor*, which is a mapping from program points to sets of constraints. This provides uniform representations for (a) input to our algorithm (and these correspond to programmer specified assertions, as in program verification), (b) the internal data-structure used in our algorithm, and (c) the output of algorithm.

Formally, given a program P with N program points, we define:

Definition 2 Let P be a program with N program points. A *program descriptor* has the form $(\mathcal{C}_1, \cdots, \mathcal{C}_N)$ where each $\mathcal{C}_i$ is a set of constraints[3]. A program descriptor is *correct* at program point i if $[\![\mathcal{C}_i]\!] \supseteq CS[i]$, where $CS[i]$ is the set of environments associated to program point i by the collecting semantics CS of P. A program descriptor is *correct* if it is correct for all program points. $\Box$

We now present an operator that formalizes the core computational step of our algorithm. First define a family of functions $\mathcal{T}[i]$, $1 \le i \le N$, associated to a program P with N program points, as follows:

$$\mathcal{T}[i]((\mathcal{C}_1, \cdots, \mathcal{C}_N)) \stackrel{\text{def}}{=}$$

$$\left\{ c : \begin{array}{l} \text{the } i^{th} \text{ equation of } \mathcal{E}(P) \text{ is} \\ C_i = (c_1, d_1) \cup \cdots \cup (c_n, d_n), \text{AND} \\ \exists c' \in \mathcal{C}_{d_j}, 1 \le j \le n, \text{AND} \\ c \text{ is } \exists_{\bar{V}_j} (c_j \wedge c'\!\downarrow) \end{array} \right\}$$

where V_j is the set of program variables in c_j.

Note that (a) the set $\mathcal{C}_*$, covering the case above where d_j is $*$, is defined to be the singleton $\{true\}$, and (b) the notation $\exists_{\bar{V}_j} c$ denotes existential closure over variables in c that do not appear in V_j. We now define the operator $\mathcal{T}$.

Definition 3 $\mathcal{T}(\tilde{\mathcal{C}}) \stackrel{\text{def}}{=} (\mathcal{T}[1](\tilde{\mathcal{C}}), \ldots, \mathcal{T}[N](\tilde{\mathcal{C}}))\ \Box$

Note that this fixpoint operator is structurally similar to $\mathcal{F}_P$ defined in Section 2, and we formalize this connection below[4]. It is monotonic and continuous, although we shall not need these properties here. Figure 4 contains an example of a program and its associated operator $\mathcal{T}$, and descriptors $\tilde{\mathcal{C}}$ and $\mathcal{T}(\tilde{\mathcal{C}})$.

Intuitively, executing $\mathcal{T}[i]$ corresponds to computing information for program point i from the program points on which i directly depends. However the definition of $\mathcal{T}[i]$ is based on the syntax of constraints, and not on their meaning. For example, if we had just one program point and the equation $C_1 = (x = x^{\downarrow}, 1)$, then $T[1](\{x = 2\})$ is $\{\exists x^{\downarrow}(x = x^{\downarrow} \wedge x^{\downarrow} = 2)\}$, and $T[1](T[1](\{x = 2\}))$ is $\{\exists x^{\downarrow}(x = x^{\downarrow} \wedge \exists x^{\downarrow}(x = x^{\downarrow} \wedge x^{\downarrow} = 2))\}$. Hence $\{x = 2\}$ is not a fixpoint of $\mathcal{T}$, although it is intuitively one because it is equivalent to both of the above constraints.

At one extreme, we could interpret sets of constraints using $[\![\cdot]\!]$. That is, we could define an ordering on constraints $\mathcal{C}$ by $\mathcal{C} \le \mathcal{C}'$ if $[\![\mathcal{C}]\!] \subseteq [\![\mathcal{C}']\!]$, and consider constraints $\mathcal{C}$ and $\mathcal{C}'$ to be equal if $[\![\mathcal{C}]\!] = [\![\mathcal{C}']\!]$. In general, this ordering will not be computable even on finite sets of constraints. In practice, it suffices to implement a partial decision procedure. More formally, consider an approximate implication

[3] In our algorithm, these sets will always be finite.

[4] It is also remarkably similar to the T_P operator in CLP, used in the context of non-ground answer constraints [12].

operation, call it $\Longrightarrow_{approx}$ such that if $\mathcal{C} \Longrightarrow_{approx} \mathcal{C}'$ then $[\![\mathcal{C}]\!] \subseteq [\![\mathcal{C}']\!]$. This can be used to define a variant of $\mathcal{T}$ that operates over equivalence classes of constraints. However, such a $\mathcal{T}$ will not in general be monotonic[5]. Fortunately, we do not need such properties. For the correctness of our algorithm, all that is needed is the property described by the following proposition. In the following proposition and lemma, we shall freely interchange objects of the form $[\![\tilde{\mathcal{C}}]\!]$ and interpretations (defined in Section 2): both can be viewed as mappings from program points to collections of environments.

Proposition 1 For all program descriptors $\tilde{\mathcal{C}}$, $[\![\mathcal{T}(\tilde{\mathcal{C}})]\!][i] = \mathcal{F}_P([\![\tilde{\mathcal{C}}]\!])(C_i)$. $\square$

Using this we can prove the correctness of the following property which is the cornerstone of the verification component of our algorithm.

Lemma 1 If $\tilde{\mathcal{C}} = (\mathcal{C}_1, \ldots, \mathcal{C}_N)$ is a program descriptor such that, for each i, $1 \leq i \leq N$, either:

(a) $\mathcal{T}(\tilde{\mathcal{C}})[i] \Longrightarrow_{approx} \mathcal{C}_i$ (i.e. the analysis "proves" $\mathcal{C}_i$), or

(b) $CS(P)[i] \subseteq [\![\mathcal{C}_i]\!]$ (i.e. $\mathcal{C}_i$ is correct),

then $\tilde{\mathcal{C}}$ is a correct program descriptor.

Proof: The proof is essentially an application of Bekic's Lemma on simultaneous fixpoints. Let A be the set of program points such that case (a) of the conditions of the lemma holds. Now, if $i \in A$, then $\mathcal{T}(\tilde{\mathcal{C}})[i] \Longrightarrow_{approx} \mathcal{C}_i$ and so by definition of approximate implication, $[\![\mathcal{T}(\tilde{\mathcal{C}})[i]]\!] \subseteq [\![\mathcal{C}_i]\!]$. By Proposition 1, $\mathcal{F}_P([\![\tilde{\mathcal{C}}]\!])(C_i) \subseteq [\![\mathcal{C}_i]\!]$.

Define an interpretation $\mathcal{I}$ (see Section 2) by

$$\mathcal{I}(C_i) \stackrel{\text{def}}{=} \begin{cases} [\![\mathcal{C}_i]\!], \text{ if } i \in A \\ CS(P)(C_i), \text{ otherwise} \end{cases}$$

Clearly $\mathcal{I} \subseteq [\![\tilde{\mathcal{C}}]\!]$, and so $\mathcal{F}_P(\mathcal{I})(C_i) \subseteq \mathcal{F}_P([\![\tilde{\mathcal{C}}]\!])(C_i) \subseteq [\![\mathcal{C}_i]\!]$. Next, define an operator $\mathcal{F}'$ that maps from and into interpretations $\mathcal{I}$ as follows:

$$\mathcal{F}'(\mathcal{I})(C_i) \stackrel{\text{def}}{=} \begin{cases} \mathcal{F}_P(\mathcal{I})(C_i), \text{ if } i \in A \\ CS(P)(C_i), \text{ otherwise} \end{cases}$$

We can show $\mathcal{F}'(\mathcal{I}) \subseteq \mathcal{I}$, and so $lfp(\mathcal{F}') \subseteq \mathcal{I}$. We can also show, for all n, $\mathcal{F}_P \uparrow n \subseteq \mathcal{F}' \uparrow n$, where $\mathcal{F} \uparrow n$ for an operator $\mathcal{F}$ on interpretations is defined in the standard way: $(\mathcal{F} \uparrow 0) = (\{\}, \ldots, \{\})$, and $\mathcal{F}_P \uparrow (n+1) = \mathcal{F}_P(\mathcal{F}_P \uparrow n)$. Since $\mathcal{F}_P$ and $\mathcal{F}'$ are monotone and continuous, it follows from standard fixpoint results that $lfp(\mathcal{F}_P) \subseteq lfp(\mathcal{F}')$. Hence $CS(P) \subseteq lfp(\mathcal{F}')$. Combining the various statement proved so far, we have $CS(P) \subseteq lfp(\mathcal{F}') \subseteq \mathcal{I} \subseteq [\![\tilde{\mathcal{C}}]\!]$, and so $\tilde{\mathcal{C}}$ is a correct program descriptor. $\square$

[5]Consider for example an implementation of approximate implication with a timeout mechanism to bound runtime. The $\Longrightarrow_{approx}$ relation defined by this algorithm with not satisfy the necessary monotonicity properties to ensure that $\mathcal{T}$ is monotonic wrt $\Longrightarrow_{approx}$.

4 The Framework

The main algorithm takes as input a program and a (not necessarily correct) descriptor, and it outputs a descriptor $\tilde{\mathcal{C}}$ which is *conditionally correct* on S, where S is a subset of program points. This means that the descriptor is correct *provided* $\tilde{\mathcal{C}}[i]$, $i \in S$ are correct.

Our main algorithm consists of two main phases, one that concerns verification and the other analysis. In phase I, we define a notion of "one-step exact propagation", constituting the basic step in an iterative process. The basic step serves to determine, given a current descriptor, the consequences local to one program point. Intuitively, the function $\mathcal{T}$ can be computed by applying this step simultaneously over all program points. Phase I then iterates this basic step to an arbitrary extent. Phase II is concerned with analysis. No specific analysis algorithm is considered; instead an analysis algorithm is a parameter to our algorithm. The idea, but not a requirement, is that the given analyzer can be adapted to take advantage of a given descriptor to obtain better results. We argue below that this is not intrusive. There is an additional phase which attempts to determine if any input descriptor (representing the user assertions) is in fact wrong.

The main algorithm then iterates the phases to an arbitrary extent. In what follows, note that at any stage in the main algorithm, a constraint solver may be applied to simplify constraints.

4.1 Phase I (One-Step Exact Propagation)

This is the critical phase of our algorithm. It serves to compute, given a program descriptor, a new descriptor by using exact propagation as realized by the operator $\mathcal{T}[i]$. In what follows, note that the real operation performed on the descriptor is not just $\mathcal{T}[i]$, but also a joining operation. The purpose here is to ensure that successive applications of $\mathcal{T}[i]$ can only lead to information that is at least as good as what is already in hand.

Phase I can now be simply defined as:

```
obtain an initial descriptor C̃_init from the user
let C̃_current = C̃_init
do
      choose any program point i
      C̃_current[i] = T[i](C̃_current) ⋈ C̃_current[i]
until (timeout)
```

where `timeout` is some arbitrary termination criterion. At one extreme, `timeout` happens when we reach a fixpoint of the function $(\mathcal{T} \bowtie Id)$, where Id is the identity function[6]. This is in general undecidable and so this condition cannot be systematically applied. A more practical implementation is to `timeout` depending on a static property of the program, e.g. terminate after $f(N)$ steps, for some simple function f. We discuss this further below.

When the above iteration terminates, of interest are those program points i such that

$$[\![\mathcal{T}[i](\tilde{\mathcal{C}}_{current})]\!] \not\subseteq [\![\tilde{\mathcal{C}}_{current}[i]]\!]$$

Call such program points *non-conforming* wrt $\tilde{\mathcal{C}}_{current}$.

[6]$(\mathcal{T} \bowtie Id)(\tilde{\mathcal{C}}) \stackrel{\text{def}}{=} \mathcal{T}(\tilde{\mathcal{C}}) \bowtie \tilde{\mathcal{C}}$

The approximate implication operator $\Longrightarrow_{approx}$ will be used to test if a given program point is conforming.

Lemma 2 Let $\tilde{\mathcal{C}}$ be the descriptor produced after phase I, and let S be its non-conforming program points. Then $\tilde{\mathcal{C}}$ is conditionally correct wrt to S. $\square$

The idea is, of course, to have as few non-conforming points as possible. In case there are none, then the program is in fact verified wrt the current descriptor (which in fact is saying more than the program being verified wrt the the initial descriptor).

We conclude this subsection with the definition of a special termination condition which intuitively makes phase I "straight-line accurate". We show that this provides a minimal amount of propagation: constraints are accurately propagated throughout all straight lines[7] of the program.

Definition 4 Let P be a program with N program points. Phase I is *straight-line accurate* on P if its termination condition is such that, for every input descriptor $\tilde{\mathcal{C}}_{init}$, the resulting descriptor $\tilde{\mathcal{C}}_{current}$ satisfies[8]

$$[\![\tilde{\mathcal{C}}_{current}]\!] \subseteq [\![T^N(\tilde{\mathcal{C}}_{init})]\!]. \quad \square$$

We now informally state that a straight-line accurate phase I will, in some sense, do at least as well as a Hoare-style program verifier. More specifically, we can show that phase I performs at least the lemma generation phase of such a verifier which serves to obtain the verification conditions. Our inbuilt constraint handling mechanism then automatically performs the required proofs.

Lemma 3 Let P be a *while*-program and suppose there is a Hoare proof of P using assertions $\mathcal{C}_1, \ldots, \mathcal{C}_k$ at program points $i_1, \ldots, i_k$ corresponding to the input and output assertions, as well the loop invariants to be used in the Hoare proof. Consider a corresponding descriptor $\tilde{\mathcal{C}}_{init}$ for P such that $\tilde{\mathcal{C}}_{init}[i_j] = \mathcal{C}_j$ for $1 \leq j \leq k$, and $\tilde{\mathcal{C}}_{init}[.] = \{true\}$ everywhere else. Then a straight-line accurate phase I, with C_{init} as input, will return a correct (unconditional) descriptor. $\square$

This lemma is, strictly speaking, dependent on the completeness of the approximate implication operator $\Longrightarrow_{approx}$.

4.2 Phase II (Approximate Propagation)

A program analyzer serves to compute a conservative approximation of $lfp([\![\mathcal{T}]\!])$, that is it returns a descriptor $\tilde{\mathcal{C}}$ such that $[\![\tilde{\mathcal{C}}]\!] \supseteq lfp([\![\mathcal{T}]\!])$. (In practice only some parts of the descriptor are of interest.) Typically, it takes as input a representation (such as $\mathcal{E}(P)$) of the program, considers a fixpoint operator associated with this representation (such as $\mathcal{T}$) and performs fixpoint iteration toward the least fixpoint. Typically, it does not take any other input.

In this phase, we define our analysis procedure. We assume that we are given some existing analyzer, say $\mathcal{Z}$, and also, we are given some information provided from the previous phase, namely $\tilde{\mathcal{C}}_{current}$. A minimal requirement of the resulting analysis is that it is at least as good as $\mathcal{Z}$ *and* $\tilde{\mathcal{C}}_{current}$. A trivial way to obtain this is to use $\mathcal{Z}$ itself, obtain the descriptor, and then join it with $\tilde{\mathcal{C}}_{current}$. In this simple setting, however, successive applications of this analysis phase would not bring improvement. We therefore seek to do better.

We now consider using the information $\tilde{\mathcal{C}}_{current}$ *within* the algorithm $\mathcal{Z}$. Not having the formal definition of $\mathcal{Z}$ at hand, we cannot do this directly. There are however at least two general approaches, which we describe informally.

- *Iterative joining:*
  One is to directly modify certain key steps in $\mathcal{Z}$. Assuming that $\mathcal{Z}$ is an iterative process, and there is a data structure for the current abstract values updated during iteration, we can embed a step which bounds these values. Here, we can improve, in each iteration, the current computed abstract value with one that best approximates the information contained in the joining of the old value and $\tilde{\mathcal{C}}_{current}$.
- *Program transformation*
  Given a program P and $\tilde{\mathcal{C}}_{current}$, consider modifying P (or its representation) to take the latter into account. The challenge is clearly how best to obtain P'. One simple way is place at selected program points i, a statement such as

  `if` $(\neg\tilde{\mathcal{C}}_{current}[i])$ `then exit;`

  The big advantage of this approach is that the given analyzer $\mathcal{Z}$ can be applied *without modification.*

We now formalize phase II as follows. We assume that the analyzer $\mathcal{Z}$ inputs an operator such as $\mathcal{T}$ and computes an approximation of $lfp(\mathcal{T})$. We seek another operator $\mathcal{A}$ which is defined from both $\mathcal{T}$ and $\tilde{\mathcal{C}}_{current}$. This can be done as follows:

$$A(.) \stackrel{\text{def}}{=} \mathcal{T}(.) \bowtie \tilde{\mathcal{X}}$$

where $\tilde{\mathcal{X}}$ is any descriptor such that $\tilde{\mathcal{X}}[i] \supseteq \tilde{\mathcal{C}}_{current}[i]$ for all program points i. At one extreme, $\tilde{\mathcal{X}}[i]$, for all i, contains the constraint *true*, in which case our algorithm $\mathcal{A}$ reduces to $\mathcal{Z}$. This corresponds to not being able to use the information in $\tilde{\mathcal{C}}_{current}$. At the other extreme, $\tilde{\mathcal{X}}[i] = \tilde{\mathcal{C}}_{current}[i]$ for all i, in which case we have captured the information provided by $\tilde{\mathcal{C}}_{current}$ in the *best* way.

Phase II of our algorithm can now be defined as follows.

`input a descriptor` $\tilde{\mathcal{C}}_{current}$ `from phase I`
`construct any descriptor` $\tilde{\mathcal{X}}$ `such that`
$$[\![\tilde{\mathcal{X}}]\!] \supseteq [\![\tilde{\mathcal{C}}_{current}]\!]$$
`construct the operator` $\mathcal{A}(\tilde{\mathcal{C}}) = \mathcal{T}(\tilde{\mathcal{C}}) \bowtie \tilde{\mathcal{X}}$
`apply the analyzer` $\mathcal{Z}$ `to` $\mathcal{A}$`, and obtain` $\tilde{\mathcal{C}}_{new}$
`recompute` $\tilde{\mathcal{C}}_{current} = \tilde{\mathcal{C}}_{current} \bowtie \tilde{\mathcal{C}}_{new}$

4.3 Counterexamples

This phase, III, considers how to determine if in fact the given descriptor is wrong. We can do this only in the presence of a *counterexample.* Up to now, all information contained in program descriptors is conservative. In this phase, we compute some definite information about environments so

[7] Cycle free paths in the flow graph.

[8] In practice, there are more efficient ways to compute this by using the structure of the control flow graph.

Prog. point	$\tilde{\mathcal{C}}_{init}$	$\mathcal{T}(\tilde{\mathcal{C}}_{init})$	$\mathcal{T}^2(\tilde{\mathcal{C}}_{init}$	$\mathcal{T}^3(\tilde{\mathcal{C}}_{init})$
1	$20 \leq x \leq 50$	$20 \leq x \leq 50$	$20 \leq x \leq 50$	$20 \leq x \leq 50$
2	*true*	$x < 100 \wedge 20 \leq x \leq 50$	$20 \leq x \leq 50$	$20 \leq x \leq 51$
3	*true*	*true*	$21 \leq x \leq 51$	$21 \leq x \leq 51$
4	*true*	$true \cup (x \geq 100 \wedge 20 \leq x \leq 50))$	*true*	*true*

Figure 2: Example showing exact propagation

that we can discover if and when a descriptor omits this information, in which case the entire analysis process fails.

Phase III is simply as follows. Note the use of the variable $\tilde{\mathcal{C}}_{current}$ below, indicating that this phase is to be used in conjunction with the other phases, as detailed in the following section. The condition `timeout` is again an arbitrary one.

let $\tilde{\mathcal{C}}_{definite} = \{\}$
do
$\quad \tilde{\mathcal{C}}_{definite}[i]$ = $\mathcal{T}[i](\tilde{\mathcal{C}}_{definite})$
until (timeout)
if ($[\![\tilde{\mathcal{C}}_{definite}]\!] \not\subseteq [\![\tilde{\mathcal{C}}_{current}]\!]$) exit

Note that the *exit* test will in general be incomplete, since we cannot implement it exactly in all cases.

4.4 Putting It All Together

Our main algorithm can now be described as an amalgamation of the above phases: it is presented as a loop in which any one of the phases can be executed in each iteration, thus facilitating flexibility. It outputs a descriptor and the conforming programming points.

Consider an example in which two applications of phase II are useful, in figure 4. Here we assume that the language of constraints can directly specify the abstract descriptions (eg. $x = \perp, y = \top$) of the analyzer used. We seek to infer that $x = odd$ and $y = even$ at the end of the program.

Apply the main algorithm as follows. We start with $\tilde{\mathcal{C}}_{init}$ being the trivial descriptor containing a sequence of *true* constraints. Then we apply exact propagation at all program points simultaneously (see column 2 in the table in figure 4), and we repeat this for a total of 11 times (column 3). The purpose here is to have exact propagation of the descriptors along all straight-lines of the program (whose length is 11). Consider for example the program point 7 in between the two loops, and the constraint $x = 3$ in $\mathcal{T}^{11}[7](\tilde{\mathcal{C}}_{init})$. This is obtained as the conjunction of $x \geq 3$ in $\mathcal{T}[7](\tilde{\mathcal{C}}_{init})$, for this is the negation of the first loop's condition, and the propagation of $x \leq 3$ obtained from program point 6.

Next we apply the analyzer that we encountered in section 1 (column 4). If we applied it directly, we would infer $x = \top$ at program point 7. In our algorithm, this analyzer would use the information $x = 3$ (in column 3) that was obtained by exact propagation. Thus the abstract value produced is then refined to become $x = odd$. Now we terminate this analysis phase, obtaining $\tilde{\mathcal{C}}_{analyzed}$. Note that

- in this descriptor, we have the constraint $y = even$ at program point 6;
- Further note that this information cannot be propagated outside the loop to program point 7 because (a) the assignment $y = 1$ before the loop would contribute the abstract condition $y = odd$, and (b) the analyzer cannot reason about the loop condition $x < 3$ exactly, and it would in fact propagate this condition $y = odd$ to program point 7;
- in program point 7, the analysis obtains $x = odd$, rather than the exact information $x = 3$;
- the condition $y = even$ at program point 6 will be used as a loop invariant for the subsequent verification phase.

We next proceed to use the exact propagation phase in order to compute $\mathcal{T}^{11}(\tilde{\mathcal{C}}_{analyzed})$ (column 5). The abovementioned invariant, $y = even$ at program point 6, can (only) now be propagated outside the loop. This, and the fact that we can recover $x = 3$ at program point 7, gives us $x = 3 \wedge even(y)$. Note that at program point 11 that $x = -1$ (column 5) was obtained from the conjunction of $-1 \leq x \leq 0$ (column 3) and $x = odd$ (column 4). Now this phase is finished.

Finally, we apply the analysis again (column 6). Now we are able to reason that $y = even$ throughout the second loop. Hence the result.

Theorem 1 Let P be a program and let C_{init} be a descriptor of P. Suppose that an algorithm of the framework above is run on P and C_{init}, and suppose $\tilde{\mathcal{C}}$ is the descriptor finally obtained. Then if the algorithm returns

- *true:* $\tilde{\mathcal{C}}$ is correct for P.
- *maybe:* $\tilde{\mathcal{C}}$ is conditionally correct for P wrt its non-conforming program points;
- *false:* the initial descriptor C_{init} of P is not correct. □

5 An Extended Example

Here we consider a simplified variant of the "available expressions" (AV) analyzer described in Appel's book [2]. The idea here is to determine if multiple computations of the same expression can be optimized by using a value for the expression saved within an existing program variable (as opposed to more general approaches which create new variables for caching purposes). Intuitively, AV-analysis maintains a constraint set for each program point, and each constraint is of the form

$$lval = rval$$

where *lval* is a variable or array element, and *rval* an expression. Such a constraint indicates that the value for *rval* is (at this program point) available in *lval*.

Obtain a program P and an initial descriptor $\tilde{\mathcal{C}}_{init}$ of P
let $\tilde{\mathcal{C}}_{current}$ = $\tilde{\mathcal{C}}_{init}$ and $\tilde{\mathcal{C}}_{definite} = \{\}$
do
    choose one of the phases I, II, III below
        Phase I:
            choose any program point i
            $\tilde{\mathcal{C}}_{current}[i]$ = $\mathcal{T}[i](\tilde{\mathcal{C}}_{current}) \bowtie \tilde{\mathcal{C}}_{current}[i]$
        Phase II:
            construct any descriptor $\tilde{\mathcal{X}}$ such that $[\![\tilde{\mathcal{X}}]\!] \supseteq [\![\tilde{\mathcal{C}}_{current}]\!]$
            construct the operator $\mathcal{A}(\tilde{\mathcal{C}}) = \mathcal{T}(\tilde{\mathcal{C}}) \bowtie \tilde{\mathcal{X}}$
            apply the analyzer $\mathcal{Z}$ to $\mathcal{A}$, and obtain $\tilde{\mathcal{C}}_{new}$
            recompute $\tilde{\mathcal{C}}_{current} = \tilde{\mathcal{C}}_{current} \bowtie \tilde{\mathcal{C}}_{new}$
        Phase III:
            choose any program point i
            recompute $\tilde{\mathcal{C}}_{definite}[i]$ = $\mathcal{T}[i](\tilde{\mathcal{C}}_{definite})$
            if ($[\![\tilde{\mathcal{C}}_{definite}]\!] \not\subseteq [\![\tilde{\mathcal{C}}_{current}]\!]$) return *false*[a]
until (timeout)
compute $S = \{i : \mathcal{T}[i](\tilde{\mathcal{C}}_{current}) \Longrightarrow_{approx} \tilde{\mathcal{C}}_{current}[i]\}$
output $\tilde{\mathcal{C}}_{current}$ and S
if (S contains all program points) return *true*
else return *maybe*

[a] This step need not be performed in every execution of Phase III.

Figure 3: Main Algorithm

```
x = 0
y = 1
while x<3 do
      y = y * 2
      x = x + 1
endwhile
while x>0 do
      y = y + 2
      x = x - 2
endwhile
```

	$\mathcal{T}(\tilde{\mathcal{C}}_{init})$	$\mathcal{T}^{11}(\tilde{\mathcal{C}}_{init})$	$\tilde{\mathcal{C}}_{analyzed}$	$\mathcal{T}^{11}(\tilde{\mathcal{C}}_{analyzed})$	
1	*true*	*true*	$x, y = \top, \top$	true	$x, y = \top, \top$
2	$x = 0$	$x = 0$	$x, y = even, \top$	$x = 0$	$x, y = even, \top$
3	$y = 1$	$x, y = 0, 1$	$x, y = even, odd$	$x, y = 0, 1$	$x, y = even, odd$
4	$x < 3$	$x < 3$	$x, y = \top, \top$	$x = 0 \wedge y = 1, x < 3 \wedge y = even$	$x, y = \top, \top$
5	*true*	$x < 3$	$x, y = \top, even$	$x = 0 \wedge y = 0, x < 3 \wedge y = even$	$x, y = \top, even$
6	*true*	$x \leq 3$	$x, y = \top, even$	$x = 1 \wedge y = 1, x \leq 3 \wedge y = even$	$x, y = \top, even$
7	$x \geq 3$	$x = 3$	$x, y = odd, \top$	$x, y = 3, even$	$x, y = odd, even$
8	$x > 0$	$x > 0$	$x, y = odd, \top$	$x = odd$	$x, y = odd, even$
9	*true*	$x > 0$	$x, y = odd, \top$	$x = odd$	$x, y = odd, even$
10	*true*	$x > -2$	$x, y = odd, \top$	$x = odd$	$x, y = odd, even$
11	$x \leq 0$	$-1 \leq x \leq 0$	$x, y = odd, \top$	$x = -1$	$x, y = odd, even$

Figure 4: Example showing two analyses phases

$$C_1 = (true, *)$$

```
i, x, y = 10, x0, y0
```

$$C_2 = (i = 10 \wedge x = x0 \wedge y = y0 \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 1)$$

```
a[i+1] = x ⊕ y
```

$$C_3 = (i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge \\ a[i+1] = x \oplus y \wedge (\forall j)\, j \neq i \rightarrow a[j] = a^{\downarrow}[j], 2)$$

```
while i>=0 do
```

$$C_4 = (i \geq 0 \wedge i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 3) \cup \\ (i \geq 0 \wedge i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 11)$$

```
      a[i] = x ⊕ y
```

$$C_5 = (i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge \\ a[i] = x \oplus y \wedge (\forall j)\, j \neq i \rightarrow a[j] = a^{\downarrow}[j], 4)$$

```
      k = i
```

$$C_6 = (i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = i \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 5)$$

```
      while k<20 do
```

$$C_7 = (k \leq 20 \wedge i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 6) \cup \\ (k \leq 20 \wedge i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 9)$$

```
            a[k+2]=a[k+1]+a[k]
```

$$C_8 = (i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge \\ a[k+2] = a[k+1] + a[k] \wedge (\forall j)\, j \neq k+2 \rightarrow a[j] = a^{\downarrow}[j], 7)$$

```
            k = k+1
```

$$C_9 = (i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} - 1 \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 8)$$

```
      endwhile
```

$$C_{10} = (i < 0 \wedge i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 9) \cup \\ (i < 0 \wedge i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 10)$$

```
      i = i-1
```

$$C_{11} = (i = i^{\downarrow} - 1 \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 10)$$

```
endwhile
```

$$C_{12} = (i < 0 \wedge i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 11) \cup \\ (i < 0 \wedge i = i^{\downarrow} \wedge x = x^{\downarrow} \wedge y = y^{\downarrow} \wedge k = k^{\downarrow} \wedge (\forall j)\, a[j] = a^{\downarrow}[j], 14)$$

Figure 5: Example Program and Equational Representation

$$\frac{C_p \bullet x = rval \bullet C_{p+1}}{C_{p+1} = \{x = rval\} \cup \{c : c \in C_p \wedge \neg contains(c, x)\}} \quad \neg contains(rval, x)$$

$$\frac{C_p \bullet x = rval \bullet C_{p+1}}{C_{p+1} = \{c : c \in C_p \wedge \neg contains(c, x)\}} \quad contains(rval, x)$$

$$\frac{C_p \bullet a[x] = rval \bullet C_{p+1}}{C_{p+1} = \{a[x] = rval\} \cup \{c : c \in C_p \wedge \neg contains(c, a)\}}$$

$$\frac{C_p \bullet a[rval1] = rval2 \bullet CE_{p+1}}{C_{p+1} = \{a[x] = rval\} \cup \{c : c \in C_p \wedge \neg contains(c, a)\}} \quad (x = rval1) \in C_p \wedge \neg contains(rval1, a)$$

$$\frac{C_p \bullet a[rval1] = rval2 \bullet C_{p+1}}{C_{p+1} = \{c : c \in C_p \wedge \neg contains(c, a)\}} \quad (x = rval1) \notin C_p \vee contains(rval1, a)$$

$$\frac{C_p \bullet if\ (cond)\ then \bullet C_{p+1} \bullet seq \bullet C_r \bullet endif \bullet C_{r+1}}{C_{p+1} = C_p \wedge C_{r+1} = C_r \cap C_p}$$

$$\frac{C_p \bullet if\ (cond)\ then \bullet C_{p+1} \bullet seq_1 \bullet C_r \bullet else \bullet C_{r+1} \bullet seq_2 \bullet C_s \bullet endif \bullet C_{s+1}}{C_{p+1} = C_p \wedge C_{r+1} = C_p \wedge C_{s+1} = C_r \cap C_s}$$

$$\frac{C_p \bullet while\ (cond)\ do \bullet C_{p+1} \bullet seq \bullet C_r \bullet endwhile \bullet C_{r+1}}{C_{p+1} = C_p \cap C_r \wedge C_{r+1} = C_r \cap C_p}$$

Figure 6: AV Analysis described using Constraint Sets

We next describe, in figure 6, the application of a set of propagation rules for the purpose of propagating constraints across program fragments. In that figure, we consider a program fragment with program points $1, \ldots, n$ and associated constraint sets $C_1, \ldots, C_n$ respectively. Each rule then states the basic relationship that is established between the constraint sets. The symbols x and a denote a variable and an array respectively. The notation $\bullet$ is just punctuation to indicate the location of the program points. The predicate *contains*(e_1, e_2) is true when expression e_1 appears in the expression e_2. These rules are a straightforward translation of those in [2].

In what follows, we shall be mixing the AV-analysis constraints with others that allow the statement of assertions. These new constraints are the basic equalities and inequalities involving variables and array elements and integers, eg. $i \leq k$ and $a[i+1] = x \oplus y$, where $\oplus$ is some (complicated) integer operation. Though these are formally different from the AV-analysis constraints, we assume the use of a constraint solver that can do elementary reasoning about mixed constraints, in particular, that it can consider AV-analysis constraints as integer equalities.

Now consider the program in figure 5. The result of applying AV-analysis on it is shown in figure 7. (Note that the punctuation "," there is not meant necessarily to mean logical conjunction; it simply separates the various syntactic objects we call constraints.) Of particular interest is program point 4 where the question is whether the expression $x \oplus y$ is available. As shown in figure 5, AV-analysis could not infer any availability for $x \oplus y$ at program point 4. The main reason for this is the assignment to the array in the inner loop which prevents the analysis from guaranteeing that $a[i]$ is unchanged in the main loop. (See the third rule in figure 6.) In reality, at program point 4, $x \oplus y$ is available in $a[i+1]$ for all i (which is $a[i]$ from the previous iteration). We now show how to determine this.

Consider injecting the assertion

$$i \leq k \wedge a[i] = x \oplus y$$

into the program at point 8, and consider applying the main algorithm, figure 3, as follows: run Phase II once (AV analysis), and then Phase I four times, and stop. The result of phase II has already been described in figure 7. (We now assume the analyzer ignored the given assertion.) The subsequent result of running Phase I four times is shown in figure 8.

The following are key observations.

- The given assertion in the inner loop ($i \leq k \wedge a[i] = x \oplus y$) is combined with the results of AV-analysis ($x = x0 \wedge y = y0$, which essentially say that x and y are unchanged within the loop) to produce a new constraint. Both these two components of the new constraint, call it C', are needed in what follows.

- After four iterations of exact propagation (Phase I), what is obtained at point 8 is a merger of C_4 (hence the need for four iterations to reach point 8), and the result of propagating C' around the loop (twice). Call this resulting constraint C''. The program point 8 with the associated constraints C'', is in fact a *conforming* point. That is, the C'' above is a *proven* invariant. Further iterations will refine this constraint, but it always remains proven.

j	C_j
1	$\{\}$
2	$\{i = 10, x = x0, y = y0\}$
3	$\{a[i] = x \oplus y, i = 10, x = x0, y = y0\}$
4	$\{x = x0, y = y0\}$
5	$\{x = x0, y = y0\}$
6	$\{x = x0, y = y0, k = i\}$
7	$\{x = x0, y = y0\}$
8	$\{x = x0, y = y0, a[k+2] = a[k+1] + a[k]\}$
9	$\{x = x0, y = y0\}$
10	$\{x = x0, y = y0\}$
11	$\{x = x0, y = y0\}$
12	$\{x = x0, y = y0\}$

Figure 7: AV Analysis for Program in Figure 5

- Another result of these four iterations is that C_8 is propagated to program point 4. This produces the constraint at point 4 as shown in the right column of figure 8. This final constraint now contains the desired information:

  $a[i+1] = x \oplus y$

- At the end, the program points, with the associated constraint sets shown in the right column of figure 2, are conforming. That is, all these constraints are true *unconditionally.*

Acknowledgements

We thank Doug Howe for discussions and references on theorem proving and model checking.

References

[1] Alexander Aiken and Manuel Fähndrich, "Program Analysis Using Mixed Term and Set Constraints", Fourth International Static Analyses Symposium, Paris, France, September 1997.

[2] A. W. Appel, "Modern compiler implementation in ML", Chapter 17.3, Cambridge University Press, 1998.

[3] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", *Proc. 4^{th} ACM Symp. on Principles of Programming Languages*, Los Angeles, 238–252, January 1977.

[4] P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks", *Proc. 6^{th} ACM Symp. on Principles of Programming Languages*, San Antonio, 269–282, January 1979.

	$\tilde{\mathcal{C}}_{current}$ after the analysis	$\tilde{\mathcal{C}}_{current}$ after 4 applications of phase I
1	$true$	$true$
2	$\{i = 10 \wedge x = x0 \wedge y = y0\}$	$\{i = 10 \wedge x = x0 \wedge y = y0\}$
3	$\{a[i] = x \oplus y \wedge i = 10 \wedge$ $x = x0 \wedge y = y0\}$	$\{a[i] = x \oplus y \wedge i = 10 \wedge$ $x = x0 \wedge y = y0\}$
4	$\{x = x0 \wedge y = y0\}$	$\{i = 10 \wedge x = x0 \wedge$ $y = y0 \wedge a[i+1] = x \oplus y\} \cup$ $\{i \geq 0 \wedge x = x0 \wedge y = y0 \wedge$ $a[i+1] = x \oplus y \wedge k \geq 20k > i\}$
5	$\{x = x0 \wedge y = y0\}$	$\{a[i] = x \oplus y \wedge i = 10 \wedge x = x0 \wedge$ $y = y0 \wedge a[i+1] = x \oplus y\} \cup$ $\{a[i] = x \oplus y \wedge i \geq 0 \wedge x = x0 \wedge$ $y = y0 \wedge k \geq 20\}$
6	$\{x = x0 \wedge y = y0 \wedge k = i\}$	$\{a[i] = x \oplus y \wedge x = x0 \wedge y = y0 \wedge$ $k = i \wedge i \geq 0 \wedge a[i+1] = x \oplus y\}$
7	$\{x = x0 \wedge y = y0\}$	$\{x = x0 \wedge y = y0 \wedge i \geq 0 \wedge$ $a[i] = x \oplus y \wedge k = i \wedge k < 20\} \cup$ $\{x = x0 \wedge y = y0 \wedge i \geq 0 \wedge$ $a[i] = x \oplus y \wedge k - 1 < 20 \wedge$ $a[k+1] = a[k] + a[k-1]\}$
8	$\{x = x0 \wedge y = y0 \ \wedge$ $a[k+2] = a[k+1] + a[k] \ \wedge$ $k \geq i \wedge a[i] = x \oplus y\}$	$\{x = x0 \wedge y = y0 \ \wedge a[i] = x \oplus y \wedge k < 20 \wedge$ $a[k+2] = a[k+1] + a[k] \ \wedge k = i\} \cup$ $\{x = x0 \wedge y = y0 \ \wedge a[i] = x \oplus y \wedge k < 20 \ \wedge$ $a[k+2] = a[k+1] + a[k] \ \wedge k - 1 \geq i \ \wedge$ $a[k+1] = a[k] + a[k-1]\}$

Figure 8: Exact propagation for Program in Figure 5

	$\tilde{\mathcal{C}}_{current}$ after the analysis	$\tilde{\mathcal{C}}_{current}$ after 4 applications of phase I
9	$\{x = x0 \wedge y = y0\}$	$\{x = x0 \wedge y = y0 \ \wedge a[i] = x \oplus y \wedge k - 1 < 20 \ \wedge$ $a[k+1] = a[k] + a[k-1] \ \wedge k - 1 = i\} \cup$ $\{x = x0 \wedge y = y0 \ \wedge a[i] = x \oplus y \ \wedge$ $a[k+1] = a[k] + a[k-1] \ \wedge k < 20 \ \wedge$ $a[k] = a[k-1] + a[k-2]\}$
10	$\{x = x0 \wedge y = y0\}$	$\{a[i] = x \oplus y \wedge x = x0 \wedge y = y0 \wedge$ $k = i \wedge k \geq 20 \wedge i \geq 0\} \cup$ $\{k = 20 \wedge x = x0 \wedge y = y0 \ \wedge a[i] = x \oplus y \ \wedge$ $a[k+1] = a[k] + a[k-1] \ \wedge k - 1 \geq i\}$
11	$\{x = x0 \wedge y = y0\}$	$\{a[i+1] = x \oplus y \wedge x = x0 \wedge y = y0 \wedge$ $k = i + 1 \wedge k \geq 20\} \cup$ $\{k \geq 20 \wedge x = x0 \wedge y = y0 \ \wedge a[i+1] = x \oplus y \ \wedge$ $a[k+1] = a[k] + a[k-1]\}$
12	$\{x = x0 \wedge y = y0\}$	$\{i < 0 \wedge x = x0 \wedge y = y0 \ \wedge k \geq 20\}$

Figure 9: Exact propagation for Program in Figure 5 (cont'd)

[5] P. Cousot. Methods and Logics for Proving Programs. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, Elsevier, 843–993, 1990.

[6] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks" *Journal of Logic and Computation*, 2(4), 511–547, August 1992.

[7] D. Gries, "Science of Programming", Springer-Verlag, 1991.

[8] K. Havelund and N. Shankar, "Experiments in Theorem Proving and Model Checking for Protocol Verification", Proceedings of Formal Methods in Europe, 1996.

[9] N. Heintze, "Set Based Program Analysis", Ph.D. thesis, School of Computer Science, Carnegie Mellon University, October 1992.

[10] N. Heintze and J. Jaffar, "A Generic Algorithm for CLP Analysis", 12^{th} *International Conference on Logic Programming*, 49-63, 1995.

[11] N. Heintze and J. Jaffar, "An Engine for Logic Program Analysis", *Proc.* 7^{th} *IEEE Symp. on Logic in Computer Science*, pp 318–328, 1992.

[12] J. Jaffar and J.-L. Lassez, "Constraint Logic Programming", *Proc.* 14^{th} *ACM Symp. on Principles of Programming Languages*, 1987, pp 111–119.

[13] N. Jones and S. Muchnick, "A Flexible Approach to Interprocedural Dataflow Analysis and Programs with Recursive Data Structures", *Proc.* 9^{th} *ACM Symp. on Principles of Programming Languages*, pp. 244–256, January 1982.

[14] Greg Nelson and Derek C. Oppen, "Simplification by cooperating decision procedures", ACM Transactions on Programming Languages and Systems, 1(2), pp 245–257, October 1979.

[15] Sam Owre, John M. Rushby, and Natarajan Shankar, "PVS: A Prototype Verification System", in Deepak Kapur, editor, Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, Springer-Verlag, 748– 752, June 1992.

[16] V. Rusu and E. Singerman, "On Proving Safety Properties by Integrating Static Analysis, Theorem Proving and Abstraction", *5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, Springer-Verlag, March 1999.

[17] S. Graf and H. Saidi, "Verifying Invariants Using Theorem Proving" *Proc.* 8^{th} *Int. Conf. on Computer Aided Verification*, 196–207, 1996.

[18] S. Bensalem, Y. Lakhnech, S. Saidi, "Powerful Techniques for the Automatic Generation of Invariants" *Proc.* 8^{th} *Int. Conf. on Computer Aided Verification*, 323–335, 1996.