Guided Static Analysis*

Denis Gopan[1] and Thomas Reps[1,2]

[1] University of Wisconsin
[2] GrammaTech, Inc.
{gopan,reps}@cs.wisc.edu

Abstract. In static analysis, the semantics of the program is expressed as a set of equations. The equations are solved iteratively over some abstract domain. If the abstract domain is distributive and satisfies the ascending-chain condition, an iterative technique yields the most precise solution for the equations. However, if the above properties are not satisfied, the solution obtained is typically imprecise. Moreover, due to the properties of widening operators, the precision loss is sensitive to the order in which the state-space is explored.

In this paper, we introduce *guided static analysis*, a framework for controlling the exploration of the state-space of a program. The framework guides the state-space exploration by applying standard static-analysis techniques to a sequence of modified versions of the analyzed program. As such, the framework does not require any modifications to existing analysis techniques, and thus can be easily integrated into existing static-analysis tools.

We present two instantiations of the framework, which improve the precision of widening in (i) loops with multiple phases and (ii) loops in which the transformation performed on each iteration is chosen non-deterministically.

1 Introduction

The goal of static analysis is, given a program and a set of initial states, to compute the set of states that arise during the execution of the program. Due to general undecidability of this problem, the sets of program states are typically over-approximated by families of sets that both are decidable and can be effectively manipulated by a computer. Such families are referred to as abstractions or abstract domains. In static analysis, the semantics of the program is cast as a set of equations, which are solved iteratively over a chosen abstract domain. If the abstract domain possesses certain algebraic properties, namely, if the abstract transformers for the domain are monotonic and distribute over join, and if the domain does not contain infinite strictly-increasing chains, then simple iterative techniques yield the least fix-point for the set of equations.

However, many useful existing abstract domains, especially those for modeling numeric properties, do not possess the above algebraic properties. As a result, standard iterative techniques (augmented with widening, to ensure analysis convergence) tend to lose precision. The precision is lost both due to overly-conservative invariant guesses made by widening, and due to joining together the sets of reachable states along multiple paths. In previous work [11], we showed that the loss of precision can sometimes be avoided by forcing the analysis to explore the state space of the program in a certain order. In particular, we showed that the precision of widening in loops with multiple phases can be improved if

* Supported by ONR under grant N00014-01-1-0796 and by NSF under grants CCF-0540955 and CCF-0524051.

the analysis has a chance to precisely characterize the behavior of each phase before having to account for the behavior of subsequent phases.

In this paper, we introduce *guided static analysis*, a general framework for guiding state-space exploration. The framework controls state-space exploration by applying standard static-analysis techniques to a sequence of *program restrictions*, which are modified versions of the analyzed program. The result of each analysis run is used to derive the next program restriction in the sequence, and also serves as an approximation of a set of initial states for the next analysis run. Note that existing static-analysis techniques are utilized "as is", making it easy to integrate the framework into existing tools. The framework is instantiated by specifying a procedure for deriving program restrictions.

We present two instantiations of the framework. The first instantiation improves the precision of widening in loops that have multiple phases. This instantiation generalizes the lookahead-widening technique [11]. It operates by generating program restrictions that incorporate individual loop phases. Also, it lifts the limitations of lookahead widening, such as the restrictions imposed on the iteration strategy and on the length of the descending-iteration sequence.

The second instantiation addresses the precision of widening in loops where the behavior of each iteration is chosen non-deterministically. Such loops naturally occur in the realm of synchronous systems [13, 10] and can occur in imperative programs if some condition within a loop is abstracted away. This instantiation derives a sequence of program restrictions, each of which enables a single iteration behavior and disables all of the others. At the end, to make the analysis sound, a program restriction with all behaviors enabled is analyzed. This strategy allows the analysis to characterize each behavior in isolation, thereby obtaining more precise results.

In non-distributive domains, the join operation loses precision. To keep the analysis precise, many techniques propagate sets of abstract values instead of individual values. Various heuristics are used to keep the cardinalities of propagated sets manageable. The main question that these heuristics address is which abstract elements should be joined and which must be kept separate. Guided static analysis is comprised of a sequence of phases, where each phase derives and analyzes a program restriction. The phase boundaries are natural points for separating abstract values: that is, within each phase the analysis may propagate a single abstract value; however, the results of different phases need not be joined together, but may be kept as a set, thus yielding a more precise overall result. In §5, we show how to extend the framework to take advantage of such disjunctive partitioning.

We implemented a prototype of guided static analysis with both of the instantiations, and applied them to a set of small programs that have appeared in recent literature on widening. The first instantiation and its disjunctive extension were used to analyze the benchmarks from [11]. The results were compared against those produced by lookahead widening. As expected, the results obtained by the instantiation were similar to the ones in [11]. However, the results obtained with the disjunctive extension were much more precise. The second instantiation was used to analyze the examples from [10]. The obtained results were similar to

the ones in [10]. However, we believe that our approach is conceptually simpler because it does not rely on acceleration techniques.

Contributions. In this paper, we make the following contributions:

- we introduce a general framework for guiding state-space exploration; the framework utilizes existing static-analysis techniques, which makes it easy to integrate into existing tools.
- we present two instantiations of the framework, which improve the precision of widening in (i) loops that have multiple phases; (ii) loops in which the transformations performed on each iteration are selected non-deterministically.
- we describe a disjunctive extension of the framework.
- we present an experimental evaluation of our techniques.

Paper organization. §2 defines the basic concepts used in the rest of the paper; §3 introduces the framework; §4 describes the two instantiations of the framework; §5 presents the disjunctive extension of the framework; §6 gives the experimental results; §7 reviews related work.

2 Preliminaries

We assume that a program is specified by a *control flow graph (CFG)* $G = (V, E)$, where V is a set of program locations, and $E \sqsubseteq V \times V$ is a set of edges that represent the flow of control. A *program state* assigns a value to every variable in the program. We will use Σ to denote the set of all possible program states. The function $\Pi_G : E \to (\Sigma \to \Sigma)$ assigns to each edge in the CFG the concrete semantics of the corresponding program statement. The semantics of individual statements is trivially extended to operate on sets of states, i.e., $\Pi_G(e)(S) = \{\Pi_G(e)(s) \mid s \in S\}$, where $e \in E$ and $S \subseteq \Sigma$.

Let $\Theta_0 : V \to \wp(\Sigma)$ denote a mapping from program locations to sets of states. The sets of program states that are *reachable* at each program location from the states in Θ_0 are given by the least map $\Theta_\star : V \to \wp(\Sigma)$ that satisfies the following set of equations:

$$\Theta_\star(v) \supseteq \Theta_0(v), \quad \text{and} \quad \Theta_\star(v) = \bigcup_{\langle u,v\rangle \in E} \Pi_G(\langle u, v\rangle)(\Theta_\star(u)), \text{ for all } v \in V$$

The problem of computing sets of reachable states is, in general, undecidable.

Static Analysis. Static analysis sidesteps undecidability by using abstraction: sets of program states are approximated by elements of some abstract domain $\mathbb{D} = \langle D, \alpha, \gamma, \sqsubseteq, \top, \bot, \sqcup\rangle$, where $\alpha : \wp(\Sigma) \to D$ constructs an approximation for a set of states, $\gamma : D \to \wp(\Sigma)$ gives meaning to domain elements, $\sqsubseteq$ is a partial order on D, $\top$ and $\bot$ are, respectively, the least and the greatest elements of D, and $\sqcup$ is the least upper bound operator. The function $\Pi^\sharp_G : E \to (D \to D)$ gives the abstract semantics of individual program statements.

To refer to abstract states at multiple program locations, we define *abstract-state maps* $\Theta^\sharp : V \to D$. The operations α, γ, $\sqsubseteq$, and $\sqcup$ for $\Theta^\sharp$ are point-wise extensions of the corresponding operations for $\mathbb{D}$.

A static analysis computes an approximation for the set of states that are reachable from an approximation of the set of initial states according to the abstract semantics of the program. In the rest of the paper, we view static analysis

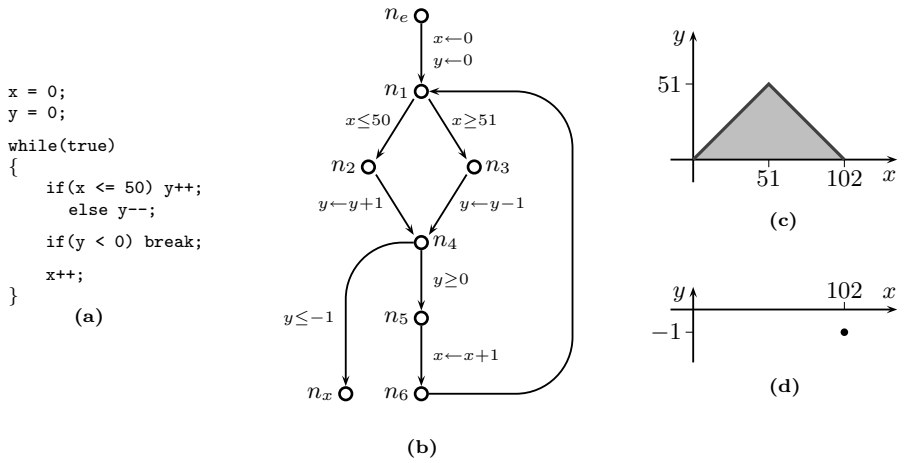


Fig. 1. Running example: (a) a loop with non-regular behavior; (b) control-flow graph for the program in (a); (c) the set of program states at n_1: the points with integer coordinates that lie on the dark upside-down "v" form the precise set of concrete states; the gray triangle gives the best approximation of that set in the polyhedral domain; (d) the single program state that reaches n_x.

as a black box, denoted by Ω, with the following interface: $\Theta^\sharp_\star = \Omega(\Pi^\sharp_G, \Theta^\sharp_0)$, where $\Theta^\sharp_0 = \alpha(\Theta_0)$ is the initial abstract-state map, and $\Theta^\sharp_\star$ is an abstract-state map that satisfies the following property:

$$\forall v \in V : \left[\Theta^\sharp_0(v) \sqcup \bigsqcup_{\langle u,v\rangle \in E} \Pi^\sharp_G(\langle u, v\rangle)(\Theta^\sharp_\star(u)) \right] \sqsubseteq \Theta^\sharp_\star(v).$$

3 Guided Static Analysis

A *guided static analysis* framework provides control over the exploration of the state space. Instead of constructing a new analysis by means of designing a new abstract domain or imposing restrictions on existing analyses (e.g., by fixing an iteration strategy), the framework relies on existing static analyses "as is". Instead, state-space exploration is guided by modifying the analyzed program to restrict some of its behaviors; multiple analysis runs are performed to explore all of the program's behaviors.

The framework is parametrized with a procedure for deriving such program restrictions. The analysis proceeds as follows: the initial abstract-state map, $\Theta^\sharp_0$, is used to derive the first program restriction; standard static analysis is applied to that program restriction to compute $\Theta^\sharp_1$, which approximates a set of program states reachable from $\Theta^\sharp_0$. Then, $\Theta^\sharp_1$ is used to derive the second program restriction, which is in turn analyzed by a standard analysis to compute $\Theta^\sharp_2$. This process is repeated until the i-th derived restriction is equivalent to the original program; the final answer is $\Theta^\sharp_i$.

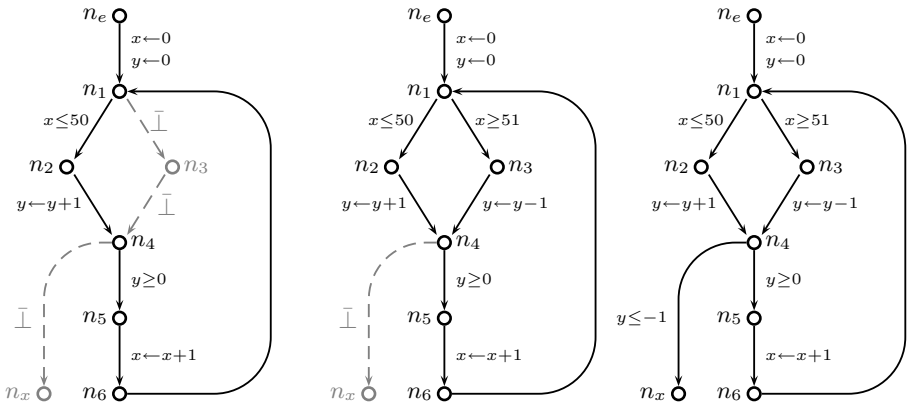


Fig. 2. Program restrictions for the program in Fig. 1: the unreachable portions of each CFG are shown in gray; (a) the first restriction corresponds to the first loop phase; (b) the second restriction consists of both loop phases, but not the loop-exit edge; (c) the third restriction incorporates the entire program.

We use the program in Fig. 1(a) to illustrate guided static analysis framework. The loop in the program has two explicit phases: during the first fifty iterations both variable x and variable y are incremented; during the next fifty iterations variable x is incremented and variable y is decremented. The loop exits when the value of the variable y falls below 0. This program is a challenge for standard widening/narrowing-based numeric analyses because the application of the widening operator over-approximates the behavior of the first phase and initiates the analysis of the second phase with overly-conservative initial assumptions. As a result, polyhedra-based standard numeric analysis concludes that at the program point n_1 the relationship between the values of x and y is $0 \leq y \leq x$, and at the program point n_x, $y = -1$ and $x \geq 50$. This is imprecise compared to the true sets of states at those program points (Figs. 1(c) and 1(d)).

Guided static analysis, when applied to the program in Fig. 1(a) consecutively derives three program restrictions shown in Fig. 2: (a) consists to the first phase of the program; (b) incorporates both phases, but excludes the edge that leads out of the loop; (c) includes the entire program. Each restriction is formed by substituting abstract transformers associated with certain edges in the control flow graph with more restrictive transformers (in this case, with $\bar{\perp}$, which is equivalent to removing the edge from the graph). We defer the description of the procedure for deriving these restrictions to §4.1.

Fig. 3(a) illustrates the operation of guided static analysis. $\Theta_0^\sharp$ approximates the set of initial states of the program. The standard numeric analysis, when applied to the first restriction (Fig. 2(a)), yields the abstract-state map $\Theta_1^\sharp$, i.e., $\Theta_1^\sharp = \Omega(\Pi_1^\sharp, \Theta_0^\sharp)$. Note, that the invariant for the first loop phase ($0 \leq x = y \leq 51$) is captured precisely. Similarly, $\Theta_2^\sharp$ is computed as $\Omega(\Pi_2^\sharp, \Theta_1^\sharp)$, and $\Theta_3^\sharp$ is

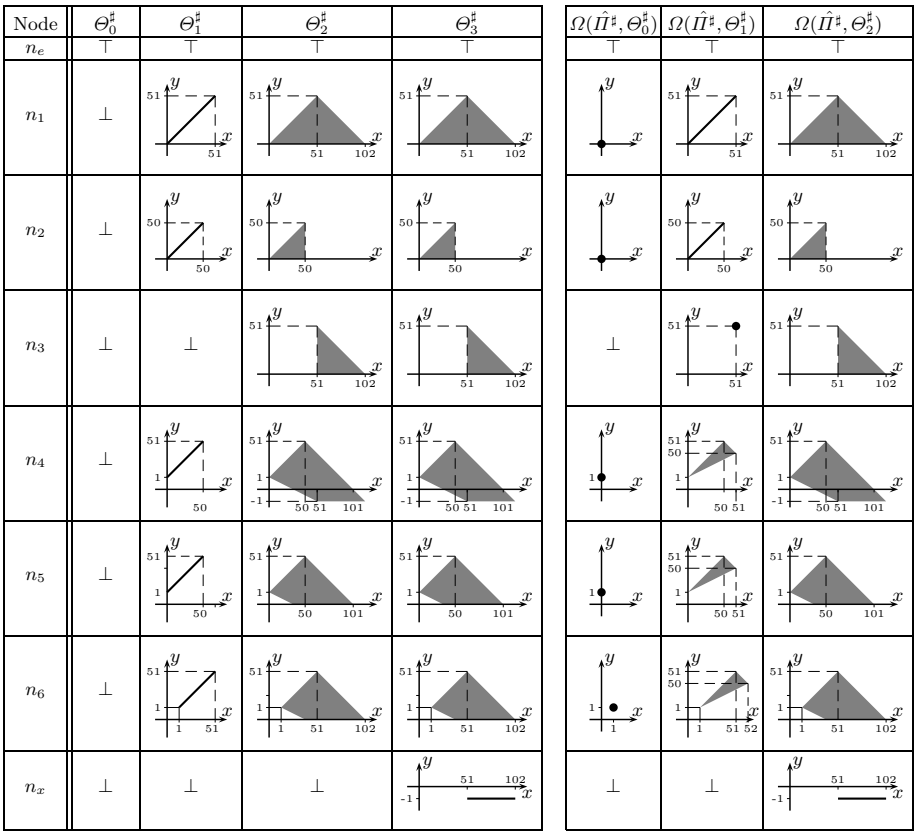


Fig. 3. Guided static analysis results for the program in Fig. 1(a); **(a)** the sequence of abstract states that are computed by analyzing the program restrictions shown in Fig. 2; $\Theta_3^\sharp$ is the overall result of the analysis; **(b)** the abstract states that are obtained by analyzing the acyclic version of the program, which are used to construct the program restrictions in Fig. 2 (see §4.1).

computed as $\Omega(\Pi_3^\sharp, \Theta_2^\sharp)$. Since the third restriction is equivalent to the program itself, the analysis stops, yielding $\Theta_3^\sharp$ as the overall result. Note that $\Theta_3^\sharp$ is more precise than the solution computed by the standard analysis: it precisely captures the loop invariant at program point n_1 and the upper bound for the value of x at node n_x. In fact, $\Theta_3^\sharp$ corresponds to the least fix-point for the program in Fig. 1(a) in the polyhedral domain.

3.1 Formal Description

We start by extending the partial order of the abstract domain to abstract transformers and to entire programs. The order is extended in a straightforward fashion.

Definition 1. *Let $f, g : D \to D$ be two abstract transformers, let $G = (V, E)$ be a control-flow graph, and let $\Pi_1^\sharp, \Pi_2^\sharp : E \to (D \to D)$ be two programs specified*

over G. Then we say that (i) $f \bar{\sqsubseteq} g$ iff $\forall d \in D : f(d) \sqsubseteq g(d)$; and (ii) $\Pi_1^\sharp \dot{\bar{\sqsubseteq}} \Pi_2^\sharp$ iff $\forall e \in E : \Pi_1^\sharp(e) \bar{\sqsubseteq} \Pi_2^\sharp(e)$.

A program restriction is a version of a program $\Pi^\sharp$ in which some abstract transformers under-approximate ($\bar{\sqsubseteq}$) those of Π. The aim is to make a standard analysis (applied to the restriction) explore only a subset of reachable states of the original program. Note, however, that, if widening is used by the analyzer, there are no guarantees that the explored state space would be smaller (because widening is, in general, not monotonic).

Definition 2 (Program Restriction). *Let $G = (V, E)$ be a control-flow graph, and $\Pi^\sharp : E \to (D \to D)$ be a program specified over G. We say that $\Pi_r^\sharp : E \to (D \to D)$ is a* restriction *of $\Pi^\sharp$ if $\Pi_r^\sharp \dot{\bar{\sqsubseteq}} \Pi^\sharp$*

To formalize guided static analysis, we need a notion of a *program transformer*: that is, a procedure Λ that, given a program and an abstract state, derives a corresponding program restriction. We allow a program transformer to maintain internal states, the set of which will be denoted $\mathbb{I}$. We assume that the set $\mathbb{I}$ is defined as part of Λ.

Definition 3 (Program transformer). *Let $\Pi^\sharp$ be a program, let $\Theta^\sharp : V \to D$ be an arbitrary abstract-state map, and let $I \in \mathbb{I}$ be an internal state of the program transformer. A* program transformer*, Λ, computes a restriction of $\Pi^\sharp$ with respect to $\Theta^\sharp$, and modifies its internal state, i.e.:*

$$\Lambda(\Pi^\sharp, I, \Theta^\sharp) = (\Pi_r^\sharp, I_r), \quad \textit{where } \Pi_r^\sharp \dot{\bar{\sqsubseteq}} \Pi^\sharp \textit{ and } I_r \in \mathbb{I}.$$

To ensure the soundness and the convergence of the analysis, we require that the program transformer possess the following property: the sequence of program restrictions generated by a non-decreasing chain of abstract states must converge to the original program in finitely many steps.

Definition 4 (Chain Property). *Let $(\Theta_i^\sharp)$ be a non-decreasing chain, s.t., $\Theta_0^\sharp \sqsubseteq \Theta_1^\sharp \sqsubseteq ... \sqsubseteq \Theta_k^\sharp \sqsubseteq ....$ Let $(\Pi_i^\sharp)$ be a sequence of program restrictions derived from $(\Theta_i^\sharp)$ as follows:*

$$(\Pi_{i+1}^\sharp, I_{i+1}) = \Lambda(\Pi^\sharp, I_i, \Theta_i^\sharp)$$

where I_0 is the initial internal state for Λ. We say that Λ satisfies the chain property if there exists a natural number n such that $\Pi_i^\sharp = \Pi^\sharp$, for all $i \geq n$.

The above property is not burdensome: any mechanism for generating program restrictions can be forced to satisfy the property by introducing a threshold and returning the original program after the threshold has been exceeded.

Definition 5 (Guided Static Analysis). *Let $\Pi^\sharp$ be a program, and let $\Theta_0^\sharp$ be an initial abstract-state map. Also, let I_0 be an initial internal state for the program transformer Λ. Guided static analysis performs the following sequence of iterations:*

$$\Theta_{i+1}^\sharp = \Omega(\Pi_{i+1}^\sharp, \Theta_i^\sharp), \textit{ where } (\Pi_{i+1}^\sharp, I_{i+1}) = \Lambda(\Pi^\sharp, I_i, \Theta_i^\sharp),$$

until $\Pi_{i+1}^\sharp = \Pi^\sharp$. The analysis result is $\Theta_\star^\sharp = \Theta_{i+1}^\sharp = \Omega(\Pi_{i+1}^\sharp, \Theta_i^\sharp) = \Omega(\Pi^\sharp, \Theta_i^\sharp)$.

Let us show that if the program transformer satisfies the chain property, the above analysis is sound and converges in a finite number of steps. Both arguments are trivial:

Soundness. Let $\Pi_a^\sharp$ be an arbitrary program and let $\Theta_a^\sharp$ be an arbitrary abstract-state map. Due to the soundness of Ω, the following holds: $\Theta_a^\sharp \sqsubseteq \Omega(\Pi_a^\sharp, \Theta_a^\sharp)$. Now, let $(\Pi_i^\sharp)$ be a sequence of programs and let $(\Theta_i^\sharp)$ be a sequence of abstract-state maps computed according to the procedure in Defn. 5. Since each $\Theta_i^\sharp$ is computed as $\Omega(\Pi_i^\sharp, \Theta_{i-1}^\sharp)$, clearly, the following relationship holds: $\Theta_0^\sharp \sqsubseteq \Theta_1^\sharp \sqsubseteq ... \sqsubseteq \Theta_k^\sharp \sqsubseteq ....$

Since Λ satisfies the chain property, there exists a number n such that $\Pi_i^\sharp = \Pi^\sharp$ for all $i \geq n$. The result of the analysis is computed as

$$\Theta_\star^\sharp = \Theta_n^\sharp = \Omega(\Pi_n^\sharp, \Theta_{n-1}^\sharp) = \Omega(\Pi^\sharp, \Theta_{n-1}^\sharp)$$

and, since $\Theta_0^\sharp \sqsubseteq \Theta_{n-1}^\sharp$ (i.e., the n-th iteration of the analysis computes a set of program states reachable from an over-approximation of the set of initial states, $\Theta_0^\sharp$), it follows that guided static analysis is sound.

Convergence. Convergence follows trivially from the above discussion: since $\Pi_n^\sharp = \Pi^\sharp$ for some finite number n, guided static analysis converges after n iterations.

4 Framework Instantiations

The framework of guided static analysis is instantiated by supplying a suitable program transformer, Λ. This section presents two instantiations that are aimed at recovering precision lost due to the use of widening.

4.1 Widening in loops with multiple phases

As was illustrated in §3, multiphase loops pose a challenge for standard analysis techniques. The problem is that standard techniques are not able to invoke narrowing after the completion of each phase to refine the analysis results for that phase. Instead, narrowing is invoked at the very end of the analysis when the accumulated precision loss is too great for precision to be recovered.

In previous work, we proposed a technique called *lookahead widening* that addressed this problem [11]. Lookahead widening propagated a pair of abstract values through the program: the first value was used to "lock" the analysis within the current loop phase; the second value computed the solution for the current phase and refined it with a narrowing sequence. When the second value converged, it was moved into the first value, thereby allowing the next loop phase to be considered. To make lookahead widening work in practice, certain restrictions were placed on the iteration strategy used by the analysis; also, the length of the descending-iteration sequence was limited to one. Furthermore, very short loop phases caused precision loss if the first value allowed the analysis to exit the current loop phase before the second value was able to converge.

In this section, we present an instantiation of the guided static analysis framework that generalizes lookahead widening and lifts the above restrictions and limitations. To instantiate the framework, we need to construct a program

transformer, Λ_{phase}, that derives program restrictions that isolate individual loop phases (as shown in Fig. 2). Intuitively, given an abstract-state map, we would like to include into the generated restriction the edges that are immediately exercised by that abstract state, and exclude the edges that require several loop iterations to become active.

To define the program transformer, we again rely on the application of a standard static analysis to a modified version of the program. Let $\hat{\Pi}^\sharp$ denote the version of $\Pi^\sharp$ from which all backedges have been removed. Note that the program $\hat{\Pi}^\sharp$ is acyclic and thus can be analyzed efficiently and precisely. The program transformer $\Lambda_{phase}(\Pi^\sharp, \Theta^\sharp)$ is defined as follows (no internal states are maintained, so we omit them for brevity):

$$\Pi^\sharp_r(\langle u, v\rangle) = \begin{cases} \Pi^\sharp(\langle u, v\rangle) & \text{if } \Pi^\sharp(\langle u, v\rangle)(\Omega(\hat{\Pi}^\sharp, \Theta^\sharp)(u)) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

In practice, we first analyze the acyclic version of the program: $\hat{\Theta}^\sharp = \Omega(\hat{\Pi}^\sharp, \Theta^\sharp)$. Then, for each edge $\langle u, v\rangle \in E$, we check whether that edge should be included in the program restriction: if the edge is active (that is, if $\Pi^\sharp(\langle u, v\rangle)(\hat{\Theta}^\sharp(u))$ yields a non-bottom value), then the edge is included in the restriction; otherwise, it is omitted.

Fig. 3(b) illustrates this process for the program in Fig. 1(a). $\hat{\Pi}^\sharp$ is constructed by removing the edge $\langle n_6, n_1\rangle$ from the program. The first column in Fig. 3(b) shows the result of analyzing $\hat{\Pi}^\sharp$ with $\Theta^\sharp_0$ used as the initial abstract-state map. The transformers associated with the edges $\langle n_1, n_3\rangle$, $\langle n_3, n_4\rangle$, and $\langle n_4, n_x\rangle$ yield $\bot$ when applied to the analysis results. Hence, these edges are excluded from the program restriction $\Pi^\sharp_1$ (see Fig. 2(a)). Similarly, the abstract-state map shown in the second column of Fig. 3(b) excludes the edge $\langle n_4, n_x\rangle$ from the restriction $\Pi^\sharp_2$. Finally, all of the edges are active with respect to the abstract-state map shown in the third column. Thus, the program restriction $\Pi^\sharp_3$ is equivalent to the original program.

Note that the program transformer Λ_{phase}, as defined above, does not satisfy the chain property from Defn. 4: arbitrary non-decreasing chains of abstract-state maps may not necessarily lead to the derivation of program restrictions that are equivalent to the original program. However, note that the process is bound to converge to some program restriction after a finite number of steps. To see this, note that each consecutive program restriction contains all of the edges included in the previously generated restrictions, and the overall number of edges in the program's CFG is finite. Thus, to satisfy the chain property, we make Λ_{phase} return $\Pi^\sharp$ after convergence is detected.

4.2 Widening in loops with non-deterministically chosen behavior

Another challenge for standard analysis techniques is posed by loops in which the behavior of each iteration is chosen non-deterministically. Such loops often arise when modeling and analyzing synchronous systems [13, 10], but they may also arise in the analysis of imperative programs when a condition of an if statement in the body of the loop is abstracted away (e.g., if variables used in the condition are

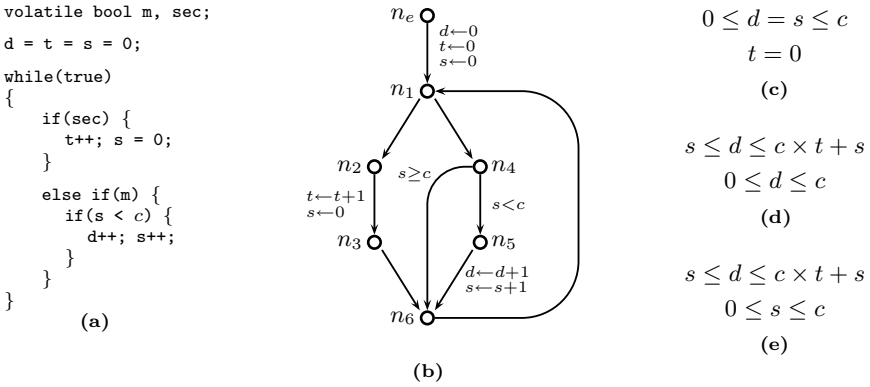


Fig. 4. A model of a speedometer with the assumption that maximum speed is c meters per second [10] (c is a positive constant): (a) a program; (b) control-flow graph for the program in (a); (c) abstract state at n_1 after $\Pi^\sharp_1$ (edge $\langle n_1, n_2\rangle$ disabled) is analyzed; (d) abstract state at n_1 after $\Pi^\sharp_2$ (edge $\langle n_1, n_4\rangle$ disabled) is analyzed; (e) abstract state at n_1 after $\Pi^\sharp_3 = \Pi^\sharp$ is analyzed.

not modeled by the analysis). These loops are problematic due to the following two reasons:

- the analysis may be forced to explore multiple iteration behaviors at the same time (e.g., simultaneously explore multiple arms of a non-deterministic conditional), making it hard for widening to predict the overall behavior of the loop accurately;
- narrowing is not effective in such loops: narrowing operates by filtering an over-approximation of loop behavior through the conditional statements in the body of the loop; in these loops, however, the relevant conditional statements are buried within the arms of a non-deterministic conditional, and the join operation at the point where the arms merge cancels the effect of such filtering.

Fig. 4(a) shows an example of such loop: the program models a speedometer with the assumption that the maximum speed is c meters per second ($c > 0$ is an arbitrary integer constant) [10]. Variables m and *sec* model signals raised by a time sensor and a distance sensor, respectively. Signal *sec* is raised every time a second elapses: in this case, the time variable t is incremented and the speed variable s is reset. Signal m is raised every time a distance of one meter is traveled: in this case, both the distance variable d and the speed variable s are incremented. Fig. 4(b) shows the CFG for the program: the environment (i.e., the signals issued by the sensors) is modeled non-deterministically (node n_1). The invariant that we desire to obtain at node n_1 is $d \leq c \times t + s$, i.e., the distance traveled is bound from above by the number of elapsed seconds times the maximum speed plus the distance traveled during the current second.

Standard polyhedral analysis, when applied to this example, simultaneously explores both arms of the non-deterministic conditional and yields the following sequence of abstract states at node n_1 during the first k iterations (we assume

that $k < c$):

$$\{\, 0 \le s \le d \le (k-1) \times t + s,\ t + d \le k \,\}$$

The application of widening extrapolates the above sequence to $\{\, 0 \le s \le d \,\}$ (i.e., by letting k go to ∞). Narrowing refines the result to $\{\, 0 \le s \le c,\ s \le d \,\}$. Thus, unless the widening delay is greater than c, the result obtained with standard analysis is imprecise.

To improve the analysis precision, we would like to analyze each of the loop's behaviors in isolation. That is, we would like to derive a sequence of program restrictions, each of which captures exactly one of the loop behaviors and suppresses the others. This can be achieved by making each program restriction enable a single outgoing edge outgoing from a node where the control is chosen non-deterministically and disable the others. After all single-behavior restrictions are processed, we can ensure that the analysis is sound by analyzing a program restriction where all of the outgoing edges are enabled.

For the program in Fig. 4(a), we construct three program restrictions: $\Pi_1^\sharp$ enables edge $\langle n_1, n_4 \rangle$ and disables $\langle n_1, n_2 \rangle$, $\Pi_2^\sharp$ enables edge $\langle n_1, n_2 \rangle$ and disables $\langle n_1, n_4 \rangle$, $\Pi_3^\sharp$ enables both edges. Figs. 4(c), 4(d), and 4(e) show the abstract states $\Theta_1^\sharp(n_1)$, $\Theta_2^\sharp(n_1)$, and $\Theta_3^\sharp(n_1)$ computed by guided static analysis instantiated with the above sequence of program restrictions. Note that the overall result of the analysis in Fig. 4(e) implies the desired invariant.

We formalize the above strategy as follows. Let $V_{nd} \subseteq V$ be a set of nodes at which loop behavior is chosen. An internal state of the program transformer keeps track of which outgoing edge is to be enabled next for each node in V_{nd}. One particular scheme for achieving this is to make an internal state I map each node $v \in V_{nd}$ to a non-negative integer: if $I(v)$ is less then the out-degree of v, then $I(v)$-th outgoing edge is to be enabled; otherwise, all outgoing edges are to be enabled. The initial state I_0 maps all nodes in V_{nd} to zero.

If iteration behavior can be chosen at multiple points (e.g., the body of the loop contains a chain of non-deterministic conditionals), the following problem arises: an attempt to isolate all possible loop behaviors may generate exponentially many program restrictions. In the prototype implementation, we resort to the following heuristic: simultaneously advance the internal states for *all* reachable nodes in V_{nd}. This strategy ensures that the number of generated program restrictions is linear in $|V_{nd}|$; however, some loop behaviors will not be isolated.

Let $deg_{out}(v)$ denote the out-degree of node v; also, let $edge_{out}(v, i)$ denote the i-th edge outgoing from v, where $0 \le i < deg_{out}(v)$. The program transformer $\Lambda_{nd}(\Pi^\sharp, I, \Theta^\sharp)$ is defined as follows:

$$\Pi_r^\sharp(\langle u, v \rangle) = \begin{cases} \bar{\bot} & \text{if } \left[\begin{array}{c} u \in V_{nd},\ \Theta^\sharp(u) \neq \bot,\ I(u) < deg_{out}(u) \\ \text{and } \langle u, v \rangle \neq edge_{out}(u, I(u)) \end{array}\right] \\ \Pi^\sharp(\langle u, v \rangle) & \text{otherwise} \end{cases}$$

The internal state of Λ_{nd} is updated as follows: for all $v \in V_{nd}$ such that $\Theta^\sharp(v) \neq \bot$, $I_r(v) = I(v) + 1$; for the remaining nodes, $I_r(v) = I(v)$.

As with the first instantiation, the program transformer defined above does not satisfy the chain property. However, the sequence of program restrictions

generated according to Defn. 4 is bound to stabilize in a finite number of steps. To see this, note that once node $v \in V_{nd}$ becomes reachable, at most $deg_{out}(v)+1$ program restrictions can be generated before exhausting all of the choices for node v. Thus, we can enforce the chain property by making Λ_{nd} return $\Pi^\sharp$ once the sequence of program restrictions stabilizes.

5 Disjunctive Extension

A single iteration of guided static analysis extends the current approximation for the entire set of reachable program states (represented with a single abstract-domain element) with the states that are reachable via the new program behaviors introduced on that iteration. However, if the abstract domain is not distributive, using a single abstract-domain element to represent the entire set of reachable program states may degrade the precision of the analysis. A more precise solution can potentially be obtained if, instead of joining together the contributions of individual iterations, the analysis represents the contribution of each iteration with a separate abstract-domain element.

In this section, we extend guided static analysis to perform such disjunctive partitioning. To isolate a contribution of a single analysis iteration, we add an extra step to the analysis. That step takes the current approximation for the set of reachable program states and constructs an approximation for the set of states that immediately exercise the new program behaviors introduced on that iteration. The resulting approximation is used as a starting point for the standard analysis run performed on that iteration. That is, an iteration of the analysis now consists of three steps: the algorithm (i) derives the (next) program restriction $\Pi^\sharp_r$; (ii) constructs an abstract-state map $\Theta^\sharp_r$ that forces a fix-point computation to explore only the new behaviors introduced in $\Pi^\sharp_r$; and (iii) performs a fix-point computation to analyze $\Pi^\sharp_r$, using $\Theta^\sharp_r$ as the initial abstract-state map.

We start by defining the *analysis history* H_k, a sequence of abstract-state maps obtained by the first $k \geq 0$ iterations of guided static analysis. H_k maps an integer $i \in [0, k]$ to the result of the i-th iteration of the analysis. H_k approximates the set of program states reached by the first k analysis iterations: $\gamma(H_k) = \bigcup_{i=0}^{k} \gamma(H_k(i))$.

The introduction of the analysis history necessitates a change in the definition of a program transformer Λ (Defn. 3): instead of a single abstract domain element, a program transformer must accept an analysis history as input. We leave it in the hands of the user to supply a suitable program transformer Λ_{dj}. In our implementation, we used a simple, albeit conservative way to construct such a program transformer from Λ:

$$\Lambda_{dj}(\Pi^\sharp, I, H_k) = \Lambda(\Pi^\sharp, I, \bigsqcup_{i=1}^{k} H_k(i)).$$

For the program in Fig. 1, Λ_{dj} derives the same program restrictions as the ones derived by plain guided static analysis (see Fig. 2).

Let $\Pi^\sharp_k$ be the program restriction derived on the k-th iteration of the analysis, where $k \geq 1$. The set of *frontier edges* for the k-th iteration consists of the

edges whose associated transformers are changed in $\Pi^\sharp_k$ from $\Pi^\sharp_{k-1}$ (for convenience, we define $\Pi^\sharp_0$ to map all edges to $\bar{\bot}$): $F_k = \left\{ e \in E \mid \Pi^\sharp_k(e) \neq \Pi^\sharp_{k-1}(e) \right\}$. For the program in Fig. 1, the sets of frontier edges on the second and third iterations are $F_2 = \{\langle n_1, n_3\rangle, \langle n_3, n_4\rangle\}$ and $F_3 = \{\langle n_4, n_x\rangle\}$.

The *local analysis frontier* for the k-th iteration of the analysis is an abstract-state map that approximates the set of states that are immediately reachable via the edges in F_k:

$$LF_k(v) = \bigsqcup_{\langle u,v\rangle \in F_k} \left[\bigsqcup_{i=0}^{k-1} \Pi^\sharp_k(\langle u, v\rangle)(H_{k-1}(i)(u)) \right].$$

For the program in Fig. 1, the local analysis frontier on the second iteration contains a single program state: $LF_2(n_3) = \{x = y = 51\}$, which is obtained by applying the transformer associated with the edge $\langle n_1, n_3\rangle$ to the abstract state $H_1(1)(n_1) = \{0 \leq x = y \leq 51\}$.

Some program states in the local analysis frontier may have already been explored on previous iterations. The *global analysis frontier* refines the local frontier by taking the analysis history into consideration. Ideally, we would like to compute

$$GF_k(v) = \alpha(\gamma(LF_k(v)) - \bigcup_{i=0}^{k-1} \gamma(H_{k-1}(i)(v))),$$

where "$-$" denotes set difference. However, this is hard to compute in practice. In our implementation, we take a simplistic approach and compute:

$$GF_k(v) = \begin{cases} \bot & \text{if } LF_k(v) \in \{H_{k-1}(i)(v) \mid 0 \leq i \leq k-1\} \\ LF_k(v) & \text{otherwise} \end{cases}$$

For the program in Fig. 1, $GF_2 = LF_2$.

Definition 6 (Disjunctive Extension). *Let $\Pi^\sharp$ be a program, and let $\Theta^\sharp_0$ be an abstract state that approximates the initial configuration of the program. Also, let I_0 be an initial internal state for the program transformer, Λ_{dj}. The disjunctive extension of guided static analysis computes the set of reachable states by performing the following iteration,*

$$H_0 = \left[0 \mapsto \Theta^\sharp_0\right] \quad \textit{and} \quad H_{i+1} = H_i \cup \left[(i+1) \mapsto \Omega(\Pi^\sharp_{i+1}, GF_{i+1})\right],$$

$$\textit{where}\,(\Pi^\sharp_{i+1}, I_{i+1}) = \Lambda_{dj}(\Pi^\sharp, I_i, H_i),$$

until $\Pi^\sharp_{i+1} = \Pi^\sharp$. The result of the analysis is given by H_{i+1}.

Fig. 5 illustrates the application of the disjunctive extension to the program in Fig. 1(a). The analysis precisely captures the behavior of both loop phases. Also, the abstract value computed for program point n_x exactly identifies the set of program states reachable at n_x. Overall, the results are significantly more precise than the ones obtained with plain guided static analysis (see Fig. 3).

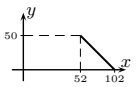
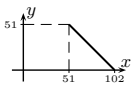
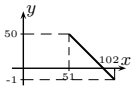
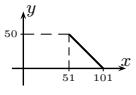
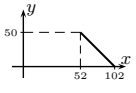
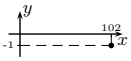
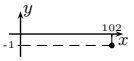
Node	$GF_1 = \Theta_0^\sharp$	$\Omega(\Pi_1^\sharp, GF_1)$	GF_2	$\Omega(\Pi_2^\sharp, GF_2)$	GF_3	$\Omega(\Pi_3^\sharp, GF_3)$
n_e	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$
n_1	$\bot$		$\bot$		$\bot$	$\bot$
n_2	$\bot$		$\bot$	$\bot$	$\bot$	$\bot$
n_3	$\bot$	$\bot$			$\bot$	$\bot$
n_4	$\bot$		$\bot$		$\bot$	$\bot$
n_5	$\bot$		$\bot$		$\bot$	$\bot$
n_6	$\bot$		$\bot$		$\bot$	$\bot$
n_x	$\bot$	$\bot$	$\bot$	$\bot$		

Fig. 5. Disjunctive extension of guided static analysis: the analysis trace for the program in Fig. 1(a); for each analysis phase, the global frontier and the resulting abstract state are shown. Note that the set of abstract values computed for program point n_x describes the true set of states reachable at n_x (see Fig. 1(d)).

6 Experimental Results

We implemented a prototype of guided static analysis. The prototype uses a polyhedra-based numeric analysis built on top of a weighted pushdown system library, wpds++ [15], as the *base* static analysis. It relies on the Parma Polyhedral Library [2] to manipulate polyhedral abstractions. A widening delay of 4 was used in all of the experiments. The performance of each analysis run is measured in *steps*: each step corresponds to a single abstract-transformer application. Speedups (overheads) are reported as the percent of extra steps performed by the baseline analysis (evaluated analysis), respectively.

We applied the instantiation from §4.1 to the set of benchmarks that were used to evaluate policy-iteration techniques [5] and lookahead widening [11]. Tab. 1 shows the results we obtained. With the exception of "test6", the results from GSA and lookahead widening are comparable: the precision is the same, and the difference in running times can be attributed to implementation choices. This is something we expected, because GSA is a generalization of the lookahead-widening technique. However, GSA yields much better results

	LA	GSA				Disjunctive GSA			
	steps	phases	steps	prec.	speedup(%)	phases	steps	prec.	speedup(%)
test1	58	2	54	-	**7.9**	2	42	-	**22.2**
test2	56	2	56	-	-	2	42	-	**25.0**
test3	58	1	44	-	**24.1**	1	42	-	**4.5**
test4	210	6	212	-	-1.0	6	154	-	**27.4**
test5	372	3	368	-	**1.1**	3	406	1/3	-10.3
test6	402	3	224	3/3	**44.3**	3	118	2/3	**47.3**
test7	236	3	224	-	**3.4**	3	154	4/4	**31.3**
test8	106	4	146	-	-37.7	3	114	-	**21.9**
test9	430	4	444	-	-3.3	4	488	4/4	-9.9
test10	418	4	420	-	-0.5	4	246	5/5	**41.4**

Table 1. Experimental results: loops with multiple phases (§4.1): GSA is compared against lookahead widening (LA); Disjunctive GSA is compared against GSA. *steps* is the total number of steps performed by each of the analyses; *phases* is the number of GSA phases; *prec* reports precision improvement: "-" indicates no improvement, k/m indicates that sharper invariants are obtained at k out of m "interesting" points (interesting points include loop heads and exit nodes);

Program	Vars	Nodes	ND	Lookahead		GSA				Overhead
				steps	inv.	runs	phases	steps	inv.	(%)
astree	1	7	1(2)	104	no	2	3	107	**yes**	2.9
speedometer	3	8	1(2)	114	no	2	3	207	**yes**	81.6
gas burner	3	8	2(2)	164	no	4	3.5	182.5	3/4	11.3
gas burner II	4	5	1(3)	184	no	6	4	162	4/6	**-12.0**

Table 2. Experimental results: loops with non-deterministic behavior (§4.2): *ND* $k(m)$ gives the amount of non-determinism: $k = |V_{nd}|$ and m is the out-degree for nodes in V_{nd}; *runs* is the number of GSA runs, each run isolates iteration behaviors in different order; *steps* is the total number of analysis steps (for GSA it is the average accross all runs); *phases* is the average number of GSA phases; *inv.* indicates whether the desired invariant is obtained (for GSA, k/m indicates that the invariant is obtained on k out of m runs).

for "test6": in "test6", the loop behavior changes when the induction variable is equal to certain values. The changes in behavior constitute short loop phases, which cause problems for lookahead widening. Also, GSA stabilizes in a fewer number of steps because simpler polyhedra arise in the course of the analysis.

Tab. 1 also compares the disjunctive extension to plain GSA. Because the analysis performed in each phase of the disjunctive extension does not have to reestablish the invariants obtained on previous phases, the disjunctive extension requires fewer analysis steps for most of the benchmarks. To compare the precision of the two analyses, we joined the analysis history obtained by the disjunctive extension for each program location into a single abstract value: for half of the benchmarks, the resulting abstract values are still significantly more precise than the ones obtained by plain GSA. Most notably, the two loop invariants in "test6" are further sharpened by the disjunctive extension, and the number of analysis steps is further reduced.

The instantiation in §4.2 is applied to a set of examples from [3, 10]: "astree" is the (second) example that motivates the use of threshold widening in [3], "speedometer" is the example used in §4.2; the two other benchmarks are the models of a leaking gas burner from [10]. The results are shown in Tab. 2: guided static analysis was able to establish the desired invariants for all of the examples. We enumerated all possible orders in which iteration behaviors can be enabled for these examples. Interestingly, the precision of the analysis on the gas-burner benchmarks does depend on the order in which the behaviors are enabled. In the future, we plan to address the issue of finding optimal behavior orders.

7 Related Work

Controlled state-space exploration. Bourdoncle discusses the effect of an iteration strategy on the overall efficiency of analysis [4]. *Lazy abstraction* [14] guides the state-space exploration in a way that avoids performing joins: the CFG of a program is unfolded as a tree and stabilization is checked by a special *covering* relation. The *directed automated random testing (DART)* technique [9] restricts the analysis to the part of the program that is exercised by a particular test input; the result of the analysis is used to generate inputs that exercise program paths not yet explored. The analysis is carried out dynamically by an instrumented version of the program. Grumberg et al. construct and analyze a sequence of under-approximated models by gradually introducing process interleavings in an effort to speed up the verification of concurrent processes [12]. We believe that the GSA framework is more general than the above approaches. Furthermore, the GSA instantiations presented in this paper address the precision of widening, which is not addressed by any of the above techniques.

Widening precision. *Threshold widening* [3] and *widening up-to* [13] rely on external invariant guesses supplied by the user or obtained from the program code with the use of some heuristics or by running a separate analysis. In contrast, our instantiations are self-contained: that is, they do not rely on external invariant guesses. The *new control-path heuristic* [13] detects the introduction of new behaviors and delays widening until the introduced behavior is sufficiently explored. However, it lacks the ability to refine the solution for already-explored behaviors before the new behavior is introduced. Policy-iteration techniques [5, 8] derive a series of program simplifications by changing the semantics of the meet operator: each simplification is analyzed with a dedicated analysis. We believe that our approach is easier to adopt because it relies on existing and well-understood analysis techniques. Furthermore, policy-iteration techniques are not yet able to operate on fully-relational abstract domains (e.g., polyhedra). The instantiation in §4.1 is the generalization of *lookahead widening* [11]: it lifts some of the restrictions imposed by lookahead widening. Gonnord et al. combine polyhedral analysis with acceleration techniques [10]: complex loop nests are simplified by "accelerating" some of the loops. The instantiation in §4.2 attempts to achieve the same effect, but does not rely on explicit acceleration techniques.

Powerset extensions. *Disjunctive completion* [6] improves the precision of the analysis by propagating sets of abstract-domain elements. However, to allow its use in numeric program analysis, widening operators must be lifted to

operate on sets of elements [1]. Sankaranarayanan et al. [18] circumvent this problem by propagating single abstract-domain elements through an elaboration of a control-flow graph (constructed on the fly). *ESP* [7], TVLA [16], and the *trace-partitioning framework* [17] structure abstract states as functions from a specially-constructed finite set (e.g., set of FSM states [7], or set of valuations of nullary predicates [16]) into the set of abstract-domain elements: at merge points, only the elements that correspond to the same member of the set are joined. The disjunctive extension in §5 differs from these techniques in two aspects: (i) the policy for separating abstract-domain elements is imposed implicitly by the program transformer; (ii) the base-level static analysis, invoked on each iteration of GSA, always propagates single abstract-domain elements.

References

1. R. Bagnara, P. Hill, and E. Zaffanella. Widening operators for powerset domains. In *VMCAI*, 2004.
2. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *SAS.*, 2002.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation*, pages 85–108. 2002.
4. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, pages 128–141, 1993.
5. A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *CAV*, 2005.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
7. M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.
8. S. Gaubert, E. Goubault, A. Taly, and S. Zennou. Static analysis by policy interation on relational domains. In *ESOP*, 2007.
9. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.
10. L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *SAS.*, pages 144–160, 2006.
11. D. Gopan and T. Reps. Lookahead widening. In *CAV*, pages 452–466, 2006.
12. O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL*, 2005.
13. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *FMSD*, 11(2):157–185, 1997.
14. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
15. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2004. http://www.cs.wisc.edu/wpis/wpds++/.
16. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS.*, pages 280–301, 2000.
17. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, pages 5–20, 2005.
18. S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *SAS.*, pages 3–17, 2006.