SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement.

Gogul Balakrishnan[1], Sriram Sankaranarayanan[1], Franjo Ivančić[1], Ou Wei[2], and Aarti Gupta[1]

[1] NEC Laboratories America, Princeton, NJ, USA.
[2] University of Toronto
{bgogul,srirams,ivancic,agupta}@nec-labs.com, owei@cs.toronto.edu

Abstract. We present a technique for detecting semantically infeasible paths in programs using abstract interpretation. Our technique uses a sequence of path-insensitive forward and backward runs of an abstract interpreter to infer paths in the control flow graph that cannot be exercised in concrete executions of the program.

We then present a syntactic language refinement (SLR) technique that automatically excludes semantically infeasible paths from a program during static analysis. SLR allows us to iteratively prove more properties. Specifically, our technique simulates the effect of a path-sensitive analysis by performing syntactic language refinement over an underlying path-insensitive static analyzer. Finally, we present experimental results to quantify the impact of our technique on an abstract interpreter for C programs.

1 Introduction

Static analysis techniques compute over-approximations of the reachable states for a given program. The theory of abstract interpretation is used to compute such an over-approximation as a fixpoint in a suitably chosen abstract domain [5, 6]. The degree of approximation as well as the scalability can be traded-off through a judicious choice of an abstract domain. However, the presence of approximations can cause the analysis to report false positives. Such false positives can be avoided, in part, using techniques such as path-sensitive analysis, disjunctive completion, and various forms of refinements [1, 7, 8, 11, 13, 17, 22].

In practice, many syntactic paths in the control-flow graph (CFG) representation of the program are semantically infeasible, i.e, they may not be traversed by any execution of the program. Reasoning about the infeasibility of such paths is a *key factor* in performing accurate static analyses for checking properties such as correct API usage, absence of null-pointer dereferences and uninitialized use of variables, memory leaks, and so on. Our previous experience with building path-sensitive abstract interpreters [22] also indicates that the benefit of added path sensitivity to static analysis seems to lie mostly in the identification and elimination of semantically infeasible paths.

```
0: if (x > 0)
1:     f := 1;
2: else
3:     f := 0;
4: y := x;
5: if (f > 0)
6:     ASSERT(y >= 0);
```

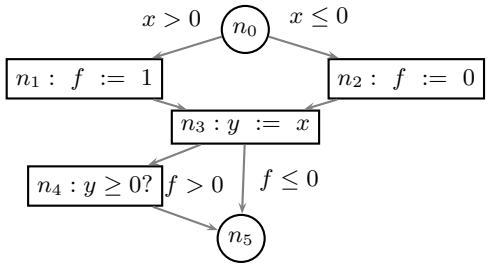

Fig. 1. An example program (left) along with its CFG representation (right). (Node numbers do not correspond to line numbers.)

This paper presents two main contributions. We present a technique based on path-insensitive abstract interpretation to infer semantically infeasible paths in the CFG. Secondly, we use a *syntactic language refinement* scheme for path-sensitive analysis by iteratively removing infeasible paths from the CFG.

Our technique for inferring semantically infeasible paths performs a sequence of many forward and backward runs using a path-insensitive abstract interpreter. We first present infeasibility theorems that use the results of forward/backward fixpoints obtained starting from different initial conditions to characterize paths in the CFG that are semantically infeasible. We then present techniques that enumerate infeasible paths using propositional SAT solvers without repeating previously enumerated paths.

The infeasible paths detected by our technique are excluded from the CFG using syntactic language refinement. Starting with the *syntactic language* defined by the set of all syntactically valid CFG paths, we remove semantically infeasible paths from the syntactic language to obtain *refinement* of the original language. Using a path-insensitive analysis over the refined syntactic language effectively incorporates partial path-sensitivity into the analysis, enabling us to prove properties that are beyond the reach of path-insensitive analyses.

Example 1. The program in Fig. 1 depicts a commonly occurring situation in static analysis. Abstract interpretation using the polyhedral abstract domain is unable to prove the property since it loses the correlation between f and x by computing a join at node n_3. On the other hand, our techniques allow us to prove using path-insensitive analysis that *any semantically feasible path from node n_0 to node n_4 cannot pass through node n_2*. Syntactic language refinement removes this path from the CFG, and performs a new path-insensitive analysis on the remaining paths in the CFG. Such an analysis maintains the correlation between x and f at node n_3 and successfully proves the property.

The F-Soft tool checks C programs for invalid pointer accesses, buffer overflows, memory leaks, incorrect usage of APIs, and arbitrary safety properties specified by a user [14]. We use the techniques described in the paper to improve the path-insensitive analysis used inside the F-Soft tool to obtain the effects

of path-sensitive analysis. The resulting analyzer proves more properties with a reasonable resource overhead.

Related Work. Path-sensitive analyses help minimize the impact of the *join* operation at the merge points in the program. A completely path-sensitive analysis is forbiddingly expensive. Many heuristic schemes achieve partial path-sensitive solutions that selectively join or separate the contributions due to different paths using logical disjunctions [8–10, 13, 17, 22]. While path-sensitive analysis techniques modify the analysis algorithm, it is possible to achieve path-sensitivity by modifying the abstract domain using powerset extensions [1, 16]. Finally, refinement-based techniques can modify the analysis algorithm or the domain itself on demand, based on the failure to prove a property. Gulavani and Rajamani iteratively refine the analysis algorithm by modifying analysis parameters such as widening delays and using weakest preconditions inside a powerset domain [11]. Cousot, Ganty, and Raskin present a fixpoint-based refinement technique that refines an initial Moore-closed abstract domain by adding predicates based on preconditions computed in the concrete domain [7].

Our technique for detecting infeasible paths relies on repeated forward and backward fixpoints. Similar ideas on using path-insensitive analyses to reason about specific program paths have appeared in the context of *abstract debugging* and *semantic slicing* [4, 20], which help to interactively zero-in on bugs in code by generating invariants, intermediate assertions, and weakest preconditions.

The use of infeasible paths to refine data-flow analysis has been considered previously by Bodik et al. [3]. However, our approach is more general in many ways: (a) we use abstract interpretation in a systematic manner to handle loops, conditions, procedures, and so on without sacrificing soundness, (b) the underlying analysis used to detect infeasible paths in our approach is itself path-insensitive, which makes it possible to apply our approach on a whole-program basis without requiring much overhead or depth cutoffs.

Ngo and Tan [19] use simple syntactic heuristics to detect infeasible paths in programs that are used in test generation. Surprisingly, their approach seems to detect many infeasible paths using relatively simplistic methods. Such lightweight approaches can also be used as a starting point for syntactic language refinement.

2 Preliminaries

Throughout this paper, we consider single-procedure (while) programs over integer variables. Our results also extend to programs with many procedures and complex datatypes. We use control-flow graphs (CFG) to represent programs. A CFG is a tuple $\langle N, E, V, \mu, n_0, \varphi_0 \rangle$, where N is a set of nodes, $E \subseteq N \times N$ is a set of edges, $n_0 \in N$ is an initial location, V is a set of integer-valued program variables, and φ_0 specifies an initial condition over V that holds at the start of the execution. Each edge $e \in E$ is labeled by a condition or an update $\mu(e)$.

A *state* of the program is a map $s : V \to Z$, specifying the value of each variable. Let Σ be the universe of all valuations to variables in V. A program is assumed to start from the initial location n_0 and a state $s \in [\![\varphi_0]\!]$.

The semantics of an edge $e \in E$ is given by the (concrete) strongest post-condition $\mathsf{post}(e, S)$ and the (concrete) weakest precondition (backward post-condition) $\mathsf{pre}(e, S)$ for sets $S \subseteq \Sigma$. The operator $\mathsf{post} : E \times 2^\Sigma \to 2^\Sigma$ yields the *smallest* set of states reachable upon executing an edge from a given set of states S, while the $\mathsf{pre} : E \times 2^\Sigma \to 2^\Sigma$ yields the *largest* set T such that $t \in T$ iff $\mathsf{post}(e, \{t\}) \subseteq S$. The pre operator is also the post for the "reverse" semantics for the edge e.

A *flow-sensitive* concrete map $\eta : N \to 2^\Sigma$ associates a set of states with each node in the CFG. We denote $\eta_1 \subseteq \eta_2$ iff $\forall n \in N, \eta_1(n) \subseteq \eta_2(n)$.

Forward Propagation.[3] The forward-propagation operator $\mathfrak{F}$ takes a concrete map $\eta : N \to 2^\Sigma$ and returns a new concrete map $\eta' : N \to 2^\Sigma$ such that

$$\eta'(m) = \begin{cases} \bigcup_{\ell \to m \in E} \mathsf{post}(\ell \to m, \eta(\ell)) & \text{if } m \neq n_0 \\ \eta(m) \cup \bigcup_{\ell \to m \in E} \mathsf{post}(\ell \to m, \eta(\ell)) & m = n_0 \end{cases}$$

A map η is *inductive* iff (a) $\eta(n_0) \supseteq [\![\varphi_0]\!]$ and (b) $\eta \supseteq \mathfrak{F}(\eta)$. In other words, if η is inductive, it is also a *post fixpoint* of $\mathfrak{F}$. Since $\mathfrak{F}$ is a monotone operator, the least fixpoint always exists due to Tarski's theorem.

Given a CFG, a *property* consists of a pair $\langle n, \varphi \rangle$ where $n \in N$ is a node, and φ is a first-order assertion representing a set of states. Associated with each property $\langle n, \varphi \rangle$, the CFG has a node $n_E \in N$ and an edge $(n \to n_E) \in E$ with condition φ_E, where φ_E is $\neg\varphi$. The pair $\langle n_E, \varphi_E \rangle$ is referred to as the *error configuration* for property $\langle n, \varphi \rangle$. A property $\langle n, \varphi \rangle$ is verified if the error configuration $\langle n_E, \varphi_E \rangle$ is unreachable, i.e., if $\eta(n_E) = \emptyset$ for an inductive map η.

The least fixpoint of a monotone operator $\mathfrak{F}$ can be computed using *Tarski* iteration. Starting from the initial map η^0, such that $\eta^0(n_0) = [\![\varphi_0]\!]$ and $\eta^0(m) = \emptyset$ for all $m \neq n_0$, we iteratively compute $\eta^{i+1} = \mathfrak{F}(\eta^i)$ until a fixpoint is reached. Unfortunately, this process is computationally infeasible, especially if the program is *infinite state*.

To analyze programs tractably, *abstract interpretation* is used to compute post fixpoints efficiently. An *abstract domain* consists of a lattice $\langle L, \sqsubseteq, \sqcup, \sqcap \rangle$, along with the abstraction map $\alpha : 2^\Sigma \to L$ and the concretization map $\gamma : L \to 2^\Sigma$. Each abstract object $a \in L$ is associated with a set of states $\gamma(a) \subseteq \Sigma$. The maps α and γ together provide a *Galois Connection* between the concrete lattice 2^Σ and the abstract lattice L. The abstract counterparts for the union ($\cup$) and intersection ($\cap$) are the lattice join ($\sqcup$) and lattice meet ($\sqcap$) operators, respectively. Finally, the concrete post-conditions and concrete preconditions have the abstract counterparts post_L and pre_L in the abstract lattice L. A flow-sensitive abstract map $\eta^\sharp : N \to L$ associates each node $n \in N$ to an abstract object $\eta^\sharp(n) \in L$. As before, $\eta_1^\sharp \sqsubseteq \eta_2^\sharp$ iff $\forall n \in N, \eta_1^\sharp(n) \sqsubseteq \eta_2^\sharp(n)$.

The forward propagation operator $\mathfrak{F}$ can be generalized to a monotone operator $\mathfrak{F}_L : \eta^\sharp \mapsto \eta^{\sharp\prime}$ in the lattice L such that:

$$\eta^{\sharp\prime}(m) = \begin{cases} \bigsqcup_{\ell \to m \in E} \mathsf{post}_L(\ell \to m, \eta^\sharp(\ell)) & \text{if } m \neq n_0 \\ \eta^\sharp(m) \sqcup \bigsqcup_{\ell \to m \in E} \mathsf{post}_L(\ell \to m, \eta^\sharp(\ell)) & m = n_0 \end{cases}$$

[3] Our presentation assumes that the initial node n_0 may have predecessors.

For a given program, abstract interpretation starts with the initial map $\eta_0^\sharp$, where $\eta_0^\sharp(n_0) = \alpha([\![\varphi_0]\!])$ and $\eta_0^\sharp(m) = \bot$ for all $m \neq n_0$. The process converges to a fixpoint $\eta_F^\sharp$ in L if $\eta_{i+1}^\sharp \sqsubseteq \eta_i^\sharp$. Furthermore, its concretization $\gamma \circ \eta_F^\sharp$ is inductive (post fixpoint) on the concrete lattice. In practice, widening heuristics enforce convergence of the iteration in lattices that do not satisfy the ascending chain condition.

Backward Propagation. An alternative to verifying a property with error configuration $\langle n_E, \varphi_E \rangle$ is *backward propagation* using the backward propagation operator $\mathcal{B}$, which takes a map $\eta : N \to 2^\Sigma$ and returns a new map $\eta' : N \to 2^\Sigma$:

$$\eta'(\ell) = \begin{cases} \eta(\ell) \bigcup_{\ell \to m \in E} \mathsf{pre}(\ell \to m, \eta(m)) & \mathsf{if}\ \ell = n_E \\ \bigcup_{\ell \to m \in E} \mathsf{pre}(\ell \to m, \eta(m)) & \mathsf{otherwise} \end{cases}$$

For an error configuration $\langle n_E, \varphi_E \rangle$, we compute the least fixpoint of $\mathcal{B}$ starting with the initial map η^0 such that $\eta^0(n_E) = [\![\varphi_E]\!]$ and $\eta^0(m) = \emptyset$ for all $m \neq n_E$. A map η is a post fixpoint of the operator $\mathcal{B}$, if $\mathcal{B}(\eta) \subseteq \eta$. The property can be verified if $\eta(n_0) \cap [\![\varphi_0]\!] = \emptyset$, which establishes that it is not possible to reach an error state in $[\![\varphi_E]\!]$ at node n_E from a state in $[\![\varphi_0]\!]$ at the initial node n_0.

Analogous to forward propagation, one may compute a backward (post) fixpoint map $\eta_B^\sharp$ in an abstract domain L by extending the operator $\mathcal{B}$ to the lattice L using the precondition map pre_L to yield $\mathcal{B}_L$. The fixpoint map $\eta_B^\sharp$ computed using $\mathcal{B}_L$ can be used to verify properties.

3 Infeasible-Path Detection

We now characterize infeasible paths in the program using abstract interpretation. Rather than focus on individual paths (of which there may be infinitely many), our results characterize sets of infeasible paths, succinctly. For the remainder of the section, we assume a given abstract domain $\langle L, \sqsubseteq, \sqcup, \sqcap \rangle$ (or even a combination of many abstract domains) that defines the forward operator $\mathfrak{F}_L$ and the backward operator $\mathcal{B}_L$. These operators transform initial maps $\eta_{F_0}^\sharp$ ($\eta_{B_0}^\sharp$) into post fixpoints $\eta_F^\sharp$ ($\eta_B^\sharp$) using abstract forward (backward) propagation. The fixpoint maps $\eta_F^\sharp$ and $\eta_B^\sharp$ are concretized to yield maps $\eta_F = \gamma \circ \eta_F^\sharp$ and $\eta_B = \gamma \circ \eta_B^\sharp$, respectively. Therefore, we present our results in the concrete domain based on concretized fixpoint maps η_F and η_B.

Consider a node $n \in N$ and a set $[\![\varphi]\!]$. We define a basic primitive called *state-set* projection that projects $\langle n, \varphi \rangle$ onto another node $m \in N$ in the CFG as follows: (a) We compute the forward fixpoint map η_F and the backward fixpoint map η_B starting from the following initial map:

$$\eta_0^{\langle n,\varphi \rangle}(\ell) = \begin{cases} [\![\varphi]\!], & \ell = n \\ \bot, & \mathsf{otherwise} \end{cases}$$

(b) The set $\eta_F(m)$ is a *forward projection* and $\eta_B(m)$ is a *backward projection* of $\langle n, \varphi \rangle$ onto m.

Definition 1 (State-set Projection). *A* forward projection *of the pair* $\langle n, \varphi \rangle$ *onto a node* m*, denoted* $(\langle n, \varphi \rangle \overset{L}{\hookrightarrow} m)$ *is the set* $\eta_F(m)$*, where* η_F *is a forward (post) fixpoint map starting from the initial map* $\eta_0^{\langle n,\varphi \rangle}$*.*

Similarly, a backward projection *of the pair* $\langle n, \varphi \rangle$ *back onto* m*, denoted* $(m \overset{L}{\hookleftarrow} \langle n, \varphi \rangle)$ *is the set* $\eta_B(m)$*, where* η_B *is the backward fixpoint starting from the initial map* $\eta_0^{\langle n,\varphi \rangle}$*.*

Forward and backward state-set projections are not unique. They vary, depending on the specific abstract interpretation scheme used to compute them. The projection of a node n onto itself yields the assertion *true*.

Lemma 1. *Let* $\varphi_F : \langle n, \varphi \rangle \overset{L}{\hookrightarrow} m$ *and* $\varphi_B : m \overset{L}{\hookleftarrow} \langle n, \varphi \rangle$ *denote the forward and backward projections, respectively, of the pair* $\langle n, \varphi \rangle$ *onto* m*. The following hold for state-set projections:*

(1) If an execution starting from a state $s \in [\![\varphi]\!]$ *at node* n *reaches node* m *with state* t*, then* $t \in [\![\varphi_F]\!]$*.*
(2) If an execution starting from node m *with state* t *reaches node* n *with state* $s \in [\![\varphi]\!]$ *then* $t \in [\![\varphi_B]\!]$*.* ⊠

3.1 Infeasibility-Type Theorems

The state-set projections computed using forward and backward propagation can be used to detect semantically infeasible paths in a CFG. Let $n_1, \ldots, n_k$ be a subset of nodes in the CFG, n_0 be the initial node and n_{k+1} be some target node of interest. We wish to find if an execution may reach n_{k+1} starting from n_0, while passing through each of the nodes $n_1, \ldots, n_k$, possibly more than once and in an arbitrary order. Let $\Pi(n_0, \ldots, n_{k+1})$ denote the set of all such syntactically valid paths in the CFG.

Let $\varphi_i : \langle n_i, \mathit{true} \rangle \overset{L}{\hookrightarrow} n_{k+1}$, $i \in [0, k+1]$, denote the forward state-set projections from $\langle n_i, \mathit{true} \rangle$ onto the final node n_{k+1}. Similarly, let $\psi_i : n_0 \overset{L}{\hookleftarrow} \langle n_i, \mathit{true} \rangle$, $i \in [0, k+1]$, denote the backward projections from $\langle n_i, \mathit{true} \rangle$ onto node n_0.

Theorem 1 (Infeasibility-type theorem). *The paths in* $\Pi(n_0, \ldots, n_{k+1})$ *are all semantically infeasible if either*

1. $\varphi_0 \wedge \varphi_1 \wedge \cdots \wedge \varphi_k \wedge \varphi_{k+1} \equiv \emptyset$*, or*
2. $\psi_0 \wedge \psi_1 \wedge \cdots \wedge \psi_k \wedge \psi_{k+1} \equiv \emptyset$*.*

Proof. The proof uses facts about the forward and backward state-set projections. From Lem. 1, we conclude that any path that reaches n_{k+1} starting from $\langle n_i, \mathit{true} \rangle$ must do so with a state that satisfies φ_i. As a result, consider a path that traverses all of $n_0, \ldots, n_k$ to reach n_{k+1}. The state at node n_{k+1} must simultaneously satisfy $\varphi_0, \varphi_1, \ldots, \varphi_k$. Since the conjunction of these assertions is empty, we conclude that no such path may exist.

A similar reasoning applies to backward projection from node n_i to node n_0 using the assertions $\psi_0, \ldots, \psi_{k+1}$. ⊠

It is also possible to formulate other infeasibility-type theorems that are similar to Thm. 1 using state-set projection. Let η_F be the forward fixpoint map computed starting from $\langle n_0, \varphi_0 \rangle$. Let $\psi_i : \ n_0 \overset{L}{\hookleftarrow} \langle n_i, \eta_F(n_i) \rangle$ be the state-set projection of the set $\eta_F(n_i)$ from node n_i onto node n_0.

Lemma 2. *If* $\psi_1 \wedge \ \ldots \wedge \psi_k \ \wedge \ \psi_{k+1} \equiv \emptyset$ *then there is no semantically valid path from node* n_0 *to node* n_{k+1} *that passes through all of* $n_1, \ldots, n_k$. ⊠

A similar result can be stated for a pair of nodes using the forward and the backward fixpoint maps. Let $\langle n_E, \varphi_E \rangle$ be an error configuration of interest, η_F be the forward (post) fixpoint map computed starting from $\langle n_0, \varphi_0 \rangle$ and η_B be the backward (post) fixpoint map computed starting from $\langle n_E, \varphi_E \rangle$.

Lemma 3. *Any error trace that leads to a state in the error configuration* $\langle n_E, \varphi_E \rangle$ *cannot visit node* n' *if* $\eta_F(n') \wedge \eta_B(n') \equiv \emptyset$. ⊠

Example 2. Consider the example program shown in Fig. 1. We wish to prove the infeasibility of any path that simultaneously visits n_0, n_2 and n_4. Using state-set projections, we obtain

$$\psi_2 : \ n_0 \overset{L}{\hookleftarrow} \langle n_2, \mathit{true} \rangle = \{x \mid x \leq 0\}$$
$$\psi_4 : \ n_0 \overset{L}{\hookleftarrow} \langle n_4, \mathit{true} \rangle = \{x \mid x > 0\}$$

Since ψ_2 and ψ_4 are disjoint, it is not possible for an execution of the CFG to visit simultaneously the nodes n_0, n_2 and n_4. Likewise, a semantically valid path cannot visit n_2 and n_4 simultaneously, since $\varphi_2 : \ \langle n_2, \mathit{true} \rangle \overset{L}{\hookrightarrow} n_4 \equiv \emptyset$. ⊠

3.2 Infeasible-Path Enumeration

Thus far, we can detect if all the program paths in $\Pi(n_0, \ldots, n_{k+1})$ are semantically infeasible for a given set $n_1, \ldots, n_k, n_{k+1}$. We now consider the problem of *enumerating* such sets using the results derived in the previous sections. Let $N = \{n_0, n_1, \ldots, n_m\}$ denote the set of all nodes in the CFG. In order to apply infeasibility-type theorems such as Thm. 1 and Lem. 2, we compute $m+1$ state-set projections $\psi_0, \ldots, \psi_m$ corresponding to the nodes $n_0, \ldots, n_m$, respectively. Furthermore, to test the subset $\{n_{i_1}, \ldots, n_{i_k}\} \subseteq N$, we test the conjunction $\psi_{i_1} \wedge \cdots \wedge \psi_{i_k}$ for satisfiability. Therefore, to *enumerate* all such subsets, we need to enumerate all index sets $I \subseteq \{1, \ldots, m\}$ such that $\bigwedge_{i \in I} \psi_i \equiv \mathit{false}$. For each such set I, the corresponding subset of N characterizes the infeasible paths.

Definition 2 (Infeasible & Saturated Index Set). *Given assertions* φ_1, $\ldots$, φ_m, *an index set* $I \subseteq \{1, \ldots, m\}$ *is said to be* infeasible *iff* $\bigwedge_{j \in I} \varphi_j \equiv \mathit{false}$. *Likewise, an infeasible index set* I *is said to be* saturated *iff no proper subset is itself infeasible.*

Note that each infeasible set is an *unsatisfiable core* of the assertion $\varphi_1 \wedge \cdots \wedge \varphi_m$. Each saturated infeasible set is a *minimal* unsatisfiable core w.r.t

set inclusion. Given assertions $\varphi_1, \ldots, \varphi_m$, we wish to enumerate all saturated infeasible index sets. To solve this problem, we provide a generic method using a SAT solver to aid in the enumeration. This method may be improved by encoding the graph structure as a part of the SAT problem to enumerate *continuous* segments in the CFG. We also present domain-specific techniques for directly enumerating all the cores without using SAT solvers.

Generic Enumeration Technique. Assume an oracle O to determine if a conjunctive theory formula $\psi_I : \bigwedge_{i \in I} \psi_i$ is satisfiable. Given O, if ψ_I is unsatisfiable we may extract a minimal unsatisfiable core set $J \subseteq I$ as follows: (1) set $J = I$, (2) for some $j \in J$, if $\psi_{J-\{j\}}$ is unsatisfiable, $J := J - \{j\}$, and (3) repeat step 2 until no more conjuncts need to be considered. Alternatively, O may itself be able to provide a minimal core.

Our procedure maintains a family of subsets $\mathfrak{I} \subseteq 2^{\{1,\ldots,m\}}$ that have not (yet) been checked for feasibility. Starting from $\mathfrak{I} = 2^{\{1,\ldots,m\}}$, we carry out steps (a) and (b), below, until $\mathfrak{I} = \emptyset$:

(a) Pick an untested subset $J \in \mathfrak{I}$.
(b) Check the satisfiability of $\psi_J : \bigwedge_{j \in J} \varphi_j$.
If ψ_J is *satisfiable*, then remove all subsets of J from $\mathfrak{I}$: $\mathfrak{I}' = \mathfrak{I} - \{K \mid K \subseteq J\}$. If ψ_J is *unsatisfiable*, compute a minimal core set $C \subseteq J$. Remove all supersets of C: $\mathfrak{I}' = \mathfrak{I} - \{I | I \supseteq C\}$. Also, *output* C as an infeasible set.

Symbolic enumeration using SAT. In practice, the set $\mathfrak{I}$ may be too large to maintain explicitly. It is therefore convenient to encode it succinctly in a SAT formula. We introduce Boolean *selector variables* $y_1, \ldots, y_m$, where y_i denotes the presence of the assertion φ_i in the theory formula. The set $\mathfrak{I}$ is represented succinctly by a Boolean formula $\mathfrak{F}$ over the selector variables. The initial formula $\mathfrak{F}_0$ is set to *true*. At each step, we may eliminate all supersets of a set J by adding the new clause $\bigvee_{j \in J} \neg y_j$. Fig. 2 shows the procedure to enumerate all infeasible indices using SAT solvers and elimination of unsatisfiable cores.

Graph-based enumeration using SAT. To further improve the enumeration procedure, we note that many subsets $I \subseteq N$ may not lead to any syntactically valid paths through the CFG that visit all nodes in I. Such subsets need not be considered. Nodes n_i and n_j are said to *conflict* if there is no syntactic path starting from n_0 that visits both n_i and n_j. Let $C \subseteq N \times N$ denote the set of all conflicting node pairs. We exclude conflicting nodes or their supersets from the enumeration process by adding the clause $\neg y_i \vee \neg y_j$ for each conflict pair $(n_i, n_j) \in C$. However, in spite of adding the conflict information, syntactically meaningless subsets may still be enumerated by our technique.

Example 3. Consider the CFG skeleton in Fig. 1, disregarding the actual operations in its nodes and edges. We suppose that all paths between nodes n_0 and n_5 are found to be infeasible: i.e., $\{0, 5\}$ is an infeasible index set. Clearly, due to the structure of the CFG, there is now no need to check the satisfiability of the index set $\{0, 3\}$, since all paths to node n_5 have to pass through node n_3.

```
1: proc GenericSATEnumerateCore(φ_1, ..., φ_m)
2:    𝔉 := true.
3:    Add all syntactic constraints to 𝔉.
4:    while (𝔉 is satisfiable ) do
5:       ⟨y_1, ..., y_m⟩ := satisfying assignment to 𝔉,
6:       Let ψ ≡ ⋀_{y_i:true} φ_i.
7:       if ψ is theory satisfiable then
8:          𝔉 := 𝔉 ∧ ⋁_{y_i:false} y_i.
9:       else
10:         Let J be the unsat core set for ψ.
11:         Output J as an infeasible set.
12:         𝔉 := 𝔉 ∧ ⋁_{j∈J} ¬y_j.
13:      end if
14:   end while
15: end proc
```

Fig. 2. SAT-based enumeration modulo theory to enumerate infeasible index sets.

However, this information is not available to the SAT solver, which generates the candidate index set $\{0, 3\}$. ⊠.

To avoid such paths, we restrict the SAT solver to enumerate *continuous segments* in the CFG.

Definition 3 (Continuous Segment). *A subset $I \subseteq N$ is a* continuous segment *iff for each $n_i \in I$, some successor of n_i (if n_i has any successors) and some predecessor (if n_i has any predecessors) belong to I.*

An infeasible index set I is *syntactically meaningful* iff there exists a continuous segment C such that $I \subseteq C$. Therefore, it suffices to restrict our SAT solver to enumerate all feasible continuous segments C in the CFG. The unsatisfiable core set for any such segment is also a syntactically meaningful set. Secondly, if a continuous segment C is shown to be semantically infeasible by a subset I, we are not interested in other infeasible subsets I' that show C to be infeasible.

Let $p_1, \ldots, p_m$ be the indices of predecessors of a node n_i ($m \geq 1$), and $s_1, \ldots, s_r$ denote the successors indices ($r \geq 1$). We encode continuous segments by adding the following constraints, corresponding to each node n_i in the CFG:

- Forward: If $m > 0$, add $\neg y_i \vee y_{p_1} \vee y_{p_2} \vee \ldots \vee y_{p_m}$.
- Backward: If $r > 0$, add $\neg y_i \vee y_{s_1} \vee y_{s_2} \vee \ldots \vee y_{s_r}$.

The total size of these constraints is linear in the size of the CFG (number of nodes, number of edges). Fig. 2 is modified to add the continuous segment constraints to $\mathfrak{F}$, in addition to the conflicts.

Example 4. Again considering the CFG skeleton from Fig. 1, if the index set $\{0, 5\}$ is found for the unsatisfiable core of the continuous segment $C : \{0, \ldots, 5\}$, the additional clause $\neg y_0 \vee \neg y_5$ will prevent any future consideration of this segment. The index set $\{0, 3\}$ will not be considered, because, n_3 is also part of the continuous segment C which has already been shown to be infeasible.

Utilizing SMT and MAX-SAT Techniques. In principle, a tighter integration of the propositional and theory part can be obtained by enumerating all minimal unsatisfiable cores of an SMT (Satisfiability Modulo Theories) formula $\Psi_m : \wedge_{i=0}^{m}((\neg y_i \vee \varphi_i) \wedge (y_i \vee \neg\varphi_i))$, along with other propositional clauses over $y_1, y_2, \ldots, y_m$ arising from conflict pairs and syntactic graph-based constraints discussed earlier.

The problem of finding all minimal unsatisfiable cores is related to the problem of finding all maximal satisfiable solutions[4] [2, 15]. This duality has been exploited to use procedures for finding maximal satisfiable solutions (MAX-SAT) for generating all minimal unsatisfiable cores. This could lead to improved performance, since checking satisfiability is usually considered easier in practice than checking unsatisfiability.

Enumerating Unsatisfiable Cores Directly. In some domains, it is possible to directly enumerate all the unsatisfiable cores of the conjunction $\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m$. Each unsatisfiable core directly yields infeasible index sets. The advantage of this enumeration method is that it avoids considering index sets for which the corresponding conjunctions is *theory satisfiable*. Secondly, the properties of the underlying abstract domain can be exploited for efficient enumeration. The disadvantage is that the same infeasible index sets may be repeatedly obtained for different unsatisfiable cores.

As a case in point, we consider the interval domain. The concretizations of interval domain objects are conjunctions of the form $x_i \in [l_i, u_i]$. Let $\varphi_1, \ldots, \varphi_m$ be the result of the state projections carried out using interval analysis. We assume that each φ_i is satisfiable. Let φ_i be the assertion $\bigwedge_j x_j \in [l_{ij}, u_{ij}]$, wherein each $l_{ij} \leq u_{ij}$. The lack of relational information in the interval domain restricts each unsatisfiable core to be of size at most 2:

Lemma 4. *Any unsatisfiable core in $\bigwedge_i \varphi_i$ involves exactly two conjuncts: $l_{ij} \leq x_j \leq u_{ij}$ in φ_i and $l_{kj} \leq x_j \leq u_{kj}$ such that $[l_{ij}, u_{ij}] \cap [l_{kj}, u_{kj}] = \emptyset$.* ⊠

As a result, it is more efficient to enumerate infeasible paths using domain-specific reasoning for the interval domain. Direct enumeration of unsatisfiable cores is possible in other domains also. For instance, we may enumerate all the negative cycles for the octagon domain, or all dual polyhedral vertices in the polyhedral domain.

4 Path-Sensitive Analysis

In this section, we describe how information about infeasible paths discovered in §3 can be used to improve the accuracy of a path-insensitive analysis.

Example 5. Consider the program shown in Fig. 3. Clearly, the assertion at line 9 in the program is never violated. However, neither a forward propagation nor

[4] Specifically, the set of all minimal unsatisfiable cores, also called MUS-es, is equivalent to the set of all irreducible hitting sets of all MCS-es, where each MCS is the complement of a maximal satisfiable solution.

```
1: if (x < 0)
2:     x := 1;
3:     y := 1;
4: else
5:     x := 2;
6:     y := 3;
7: if(y = 2)
8:     x := x - 2;
9: ASSERT(x >= 0);
```

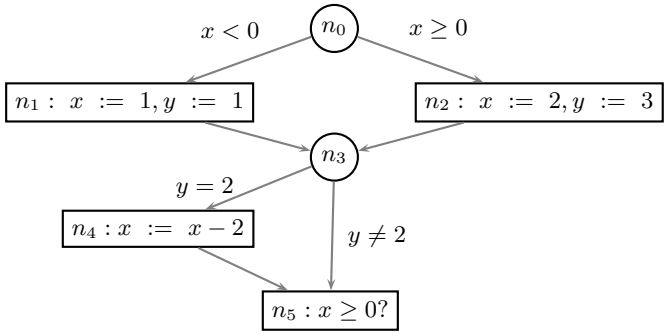

Fig. 3. An example program (left) along with its CFG representation (right). (Node numbers do not correspond to line numbers.)

a backward propagation using the interval domain is able to prove the assertion in node n_5. A SAT-based infeasible path enumeration using the interval domain enumerates the sets $\{n_2, n_4\}$ and $\{n_1, n_4\}$ as infeasible. ⊠

Syntactic Language Refinement (SLR). Let $\Pi : \langle N, E, \mu, n_0, \varphi_0 \rangle$ be a CFG with a property $\langle n, \mathit{false} \rangle$ to be established. The *syntactic language* of Π consists of all the edge sequences $e_1, \ldots, e_m$ that may be visited along some walk through the CFG starting from n_0. In effect, we treat Π as an automaton over the alphabet E, wherein each edge e_i accepts the alphabet symbol e_i. Let L_Π denote the language of all edge sequences accepted by CFG Π, represented as a deterministic finite automaton.

The results of the previous section allow us to infer sets $I : \{n_1, \ldots, n_k\} \subseteq N$ such that no semantically valid path from node n_0 to node n may pass through all the nodes of the set I. For each set I, we remove all the (semantically invalid) paths in the set $\Pi(I \cup \{n_0, n\})$ from the syntactic language of the CFG:

$$L'_\Pi = L_\Pi - \underbrace{\{\pi | \pi \text{ is a path from } n_0 \text{ to } n, \text{ passing through all nodes in } I\}}_{L_I}$$

Since the sets L_Π and L_I are regular, $L_{\Pi'} = L_\Pi - L_I$ is also regular. The refinement procedure continues to hold even if L_Π is context free, which happens in the presence of function calls and returns in the CFG.

Let Π' be the automaton recognizing $L_{\Pi'}$. The automaton Π' can be viewed as a CFG once more by associating actions (conditions and assignments) with its edges. Each edge labeled with the alphabet e_i is provided the action $\mu(e_i)$ from the original CFG Π. Since $L_{\Pi'} \subseteq L_\Pi$, Π' encodes a smaller set of syntactically valid paths. Therefore, abstract interpretation on Π' may result in a more precise fixpoint.

Example 6. Consider the CFG skeleton Π in Fig. 4(a). Suppose nodes $\{n_4, n_8\}$ were found to be infeasible. The refined CFG Π' is shown in Fig. 4(b). The

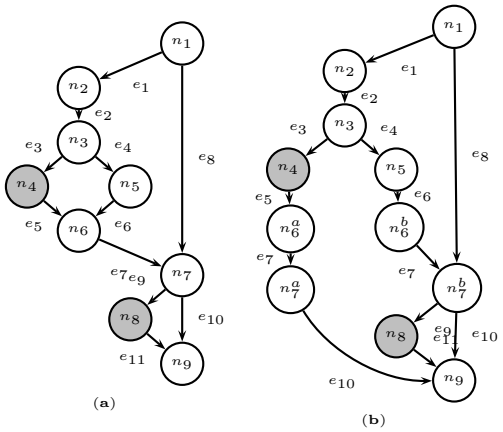

Fig. 4. Path-sensitivity can be simulated by syntactic language refinement: (a) original CFG, (b) paths visiting nodes $\{n_4, n_8\}$ are found infeasible and removed.

edges and the nodes are labeled to show the correspondences between the two CFGs. Notice that it is no longer possible to visit the shaded nodes in the same syntactic path.

A path sensitive static analysis *might* be able to obtain the same effect by not using join at node n_6, and partially merging two disjuncts at node n_7. Borrowing the terminology from our previous work the CFG Π' is an *elaboration* of the CFG Π [22], or from the terminology of Mauborgne et al., a trace partitioning [13, 17], that keeps the trace visiting node n_4 separate from the other traces. However, our prior knowledge of the infeasibility of $\{n_4, n_8\}$ enables us to automatically rule out the edge $n_7^a \rightarrow n_8$. ⊠

From the discussion above, it is evident that syntactic language refinement can be cast in the framework of related schemes such as *elaborations* or *trace-partitions*. However, the key difference is that our scheme always uses the infeasible CFG paths as a partitioning heuristic. The heuristic used in the elaboration or trace-partitioning may be unable to guess the right partitions required to detect the infeasible paths in practice.

Example 7. Returning to the example in Fig. 3, we find that the paths from $n_0 \rightsquigarrow n_5$, traversing the nodes $I_0 : \{n_1, n_4\}$ and $I_1 : \{n_2, n_4\}$ are semantically infeasible. Therefore, we may remove such paths from the CFG using syntactic language refinement. The resulting CFG Π' is simply the original CFG with the node n_4 removed. A *path-insensitive* analysis over Π' proves the property. ⊠

In theory, it is possible to first remove infeasible path segments using abstract interpretation, perform a language refinement, and subsequently, analyze the refined CFG. In practice, however, we observe that most infeasible paths involve no more than two intermediate nodes. Furthermore, the size of the refined CFG after a set I has been removed can be a factor $2^{|I|}$ larger.

```
1: proc VerifyProperty(⟨n, φ⟩)
2:    I := N
3:    Let ⟨n_E, φ_E⟩ be the error configuration for property ⟨n, φ⟩
4:    while ( I ≠ ∅ ) do
5:       Let η_F be the forward fixpoint computed starting from ⟨n_0, φ_0⟩
6:       Let η_B be the backward fixpoint computed starting from ⟨n_E, φ_E⟩
7:       I := ∅
8:       for all conditional branches ℓ → m ∈ E do
9:          if ( η_F(m) ⊓ η_B(m) ≡ false) then
10:            Let I := I ∪ {n ∈ N | n is control dependent on ℓ → m}.
11:         end if
12:      end for
13:      Remove the nodes in I from the CFG
14:      if η_F(n) ≡ false then
15:         return PROVED.
16:      end if
17:   end while
18:   return NOT PROVED.
19: end proc
```

Fig. 5. Using infeasible-path detection to improve path-insensitive analysis.

Application. We now present a simple version of the SLR scheme using Lem. 3 that removes at most one intermediate node w.r.t a given property. Secondly, we repeatedly refine the CFG at each stage by using the improved abstract interpretation result on the original CFG. Finally, thanks to the form of Lem. 3, each application of the scheme requires just two fixpoint computations, one in the forward direction and the other in the backward.

Fig. 5 shows an iterative syntactic language refinement scheme. Each step involves a forward fixpoint from the initial node and a backward fixpoint computed from the property node. First, infeasible pairs of nodes are then determined using Lem. 3, and the paths involving such pairs are pruned from the CFG. Since paths are removed from the CFG, subsequent iterations can produce stronger fixpoints, and therefore, detect more infeasible intermediate nodes. The language refinement is repeated until the property is verified, or no new nodes are detected as infeasible in consecutive iterations.

Example 8. VerifyProperty proves the assertion in the example shown in Fig. 3. During the first iteration, the condition at line 8 of *VerifyProperty* holds for the edge $n_0 \rightarrow n_2$. Consequently, node n_2 will be removed before the next forward fixpoint computation. Hence, interval analysis will be able to determine that edge $n_3 \rightarrow n_4$ is infeasible, and therefore, the property $\langle n_5, x \geq 0 \rangle$ is verified. ⊠

5 Experiments

We have implemented the SLR technique inside the F-Soft C program verifier [14] to check array, pointer, and C string usage. The analyzer is *context*

Table 1. Number of saturated infeasible sets from SAT-based enumeration.

Prog.	LOC	#Vars	#Branches	Time (s)		#Inf.Sets
				FixPoint	Enum	
ex1	35	7	13	0.07	0.01	3
ex2	40	6	15	0.08	0.02	10
ex3	79	9	36	1.46	0.12	6
ex4	85	71	41	2160.67	6.12	0
ex5	94	12	40	2.85	3.60	38
ex6	102	38	27	51.76	0.15	2
ex7	115	2	31	0.06	0.02	28

sensitive, by using call strings to track contexts. Our abstract interpreter supports a combination of different numerical domains, including constant folding, interval, octagon [18], polyhedron [12] and the symbolic range domains [21]. The experiments were run on a Linux machine with two Xeon 2.8GHz processors and 4GB of memory.

Infeasible-Path Enumeration. We implemented the algorithm in Fig. 2 using the octagon abstract domain as a proof-of-concept. Tab. 1 shows the performance of the infeasible-path enumerator over a set of small but complex functions written in C [23]. As an optimization, we modified the algorithm to enumerate infeasible sets solely involving conditional branches. The running time of the algorithm is a function of the number of conditional branches and the number of variables in the program. Surprisingly, computing fixpoints accounts for the majority of the running time. Not surprisingly, almost all saturated infeasible sets involved exactly two edges. With the additional optimizations described here and a variable-packing heuristic, we hope to scale this technique to larger functions.

Syntactic Language Refinement. We implemented the *VerifyProperty* algorithm on top of our existing abstract interpreter. Given a program, we first run a series of path-insensitive analyses. Properties thus proved are sliced away from the CFG. The resulting simplified CFG is fed into our analysis. Our analysis is run twice: first, using the interval domain, and then using the octagon domain on the sliced CFG from the interval analysis instantiation. Tab. 2 shows the performance of our tool chain on a collection of industrial as well as open source projects. For each program, we show the number of proofs obtained and the time taken by the base analysis as well as the *additional proofs* along with the overhead of the SLR technique using the intervals and the octagon domains successively. For our set of examples, the SLR technique obtains 15% more proofs over and above the base analysis. However, it involves a significant time overhead of roughly 15% for the interval domain and 75% for the octagon domain. A preliminary comparison with our previous work suggests that this overhead is quite competitive with related techniques for performing path-sensitive analysis [22].

Tab. 3 shows a direct comparison of our implementation with a partially path-sensitive analysis implemented using CFG elaborations [22]. The compari-

Table 2. Performance of the tool flow using SLR. (H_i: mobile software application modules, L_i: Linux device drivers, M_i: a network protocol implementation modules.)

		Base analysis		*Additional* Proofs with SLR			
	Simplified			Intervals		Octagons	
Code	LOC	Proofs	Time (s)	Proofs	Time (s)	Proofs	Time (s)
H_1	2282	7/14	28.08	0/7	0.5	0/7	8.77
H_2	3319	33/49	355.85	0/16	7.04	0/16	102.24
H_3	2668	23/37	52.49	0/14	1.25	0/14	14.17
L_4	4626	12/31	10.21	0/19	5.07	0/19	9.1
M_5	5095	67/265	69.62	35/198	84.99	28/163	217.44
L_6	5346	6/20	3.62	0/14	1.53	0/14	7.59
L_7	5599	5/146	681.48	0/141	198.26	0/141	239.39
M_8	6142	109/314	86.85	1/205	173.74	98/204	2008.07
L_9	6326	119/135	70.88	0/16	2.74	0/16	8.18
M_{10}	6645	93/385	244.57	25/292	390.15	70/267	990.11
M_{11}	7541	162/442	262.94	77/280	361.08	49/203	1069.1
M_{12}	10206	285/745	1479.29	30/460	1584.45	154/430	6681.36
M_{13}	11803	325/786	1089.86	121/461	859.36	99/340	5710.56
L_{14}	13162	38/114	289.9	34/76	130.98	0/42	263.86
M_{15}	14665	313/606	117.96	163/293	112.19	10/130	204.16
M_{17}	26758	1904/1918	2208.85	0/14	0.95	0/14	8.89
M_{18}	47213	4173/4218	16880.00	21/45	4.07	0/24	20.51
Total		7674/10225		507/2551		508/2044	

Table 3. Comparison of SLR with path-sensitive analysis using CFG elaborations [22]. (#T: total time with two outlying data points removed, #P: additional proofs.)

	Base			CFG Elaborations [22]		SLR Cont. Insens.		SLR Cont. Sens.	
#Progs	Tot.	Proofs	#T	#P	#T	#P	#T	#P	#T
48	403	178	2.47	+45	76.42	+44	27	+79	36

son is carried out over a collection of small example programs [23] written in the C language. These programs range from 20-400 lines of code, and are designed to evaluate the handling of loops, aliasing, dynamic allocation, type-casts, string library functions and other sources of complexities in practical programs. Because the latter implementation is context-insensitive, we compare against a context-insensitive version of our technique as well. CFG elaboration technique achieves 45 extra proofs with 50x time overhead. Our context-insensitive implementation proves roughly as many properties as CFG elaborations with a much smaller overhead (roughly 10x for context-insensitive and 20x for context-sensitive). Our implementation proved quite a few properties that could not be established by elaborations and vice versa. Not surprisingly, added context-sensitivity to our technique renders it vastly superior.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. *STTT*, 9(3-4):413–414, 2007.
2. J. Bailey and P. J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, 2005.
3. R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *FSE*, pages 361–377. Springer–Verlag, 1997.
4. François Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI '93*, pages 46–55. ACM, 1993.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. Principles of Programming Languages (POPL)*, 1977.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Descriptions of Programming Concepts*. North-Holland, 1978.
7. P. Cousot, P. Ganty, and J-F. Raskin. Fixpoint-guided abstraction refinements. In *SAS*, volume 4634 of *LNCS*, pages 333–348, 2007.
8. M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68. ACM Press, 2002.
9. D. Dhurjati, M. Das, and Y. Yang. Path-sensitive dataflow analysis with iterative refinement. In *SAS*, pages 425–442, 2006.
10. J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *FSE*, 2005.
11. B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, volume 3920 of *LNCS*, pages 474–488, 2006.
12. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in Sys. Design*, 11(2):157–185, 1997.
13. Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS*, volume 1503 of *LNCS*, 1998.
14. F. Ivančić, I. Shlyakhter, A. Gupta, M. K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-SOFT. In *ICCD*, pages 297–308, 2005.
15. Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. of Automated Reasoning*, 40(1):133, 2008.
16. R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, volume 3148 of *LNCS*, pages 265–279, 2004.
17. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symp. on Programming (ESOP)*, 2005.
18. A. Miné. The octagon abstract domain. In *Working Conf. on Reverse Eng.*, 2001.
19. Minh Ngoc Ngo and Hee Beng Kuan Tan. Detecting large number of infeasible paths through recognizing their patterns. In *FSE*, 2007.
20. Xavier Rival. Understanding the origin of alarms in ASTRÉE. In *SAS*, 2005.
21. S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis using symbolic ranges. In *SAS*, pages 366–383, 2007.
22. S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *SAS*, volume 4134 of *LNCS*, pages 3–17, 2006.
23. Sriram Sankaranarayanan. NECLA static analysis benchmarks (2007). Available from `http://www.nec-labs.com/~fsoft`.