Refining the Control Structure of Loops using Static Analysis

Gogul Balakrishnan
NEC Labs America
bgogul@nec-labs.com

Sriram Sankaranarayanan
NEC Labs America
srirams@nec-labs.com

Franjo Ivančić
NEC Labs America
ivancic@nec-labs.com

Aarti Gupta
NEC Labs America
agupta@nec-labs.com

ABSTRACT

We present a simple yet useful technique for refining the control structure of loops that occur in imperative programs. Loops containing complex control flow are common in synchronous embedded controllers derived from modeling languages such as Lustre, Esterel, and Simulink/Stateflow. Our approach uses a set of labels to distinguish different control paths inside a given loop. The iterations of the loop are abstracted as a finite state automaton over these labels. Subsequently, we use static analysis techniques to identify infeasible iteration sequences and subtract such forbidden sequences from the initial language to obtain a refinement. In practice, the refinement of control flow sequences often simplifies the control flow patterns in the loop. We have applied the refinement technique to improve the precision of abstract interpretation in the presence of widening. Our experiments on a set of complex reactive loop benchmarks clearly show the utility of our refinement techniques. Abstraction interpretation with our refinement technique was able to verify all the properties for 10 out of the 13 benchmarks, while abstraction interpretation *without* refinement was able to verify only four. Other potentially useful applications include termination analysis and reverse engineering models from source code.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers; model checking; formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs—*Assertions; Invariants; Mechanical Verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

EMSOFT'09, October 12–16, 2009, Grenoble, France.

General Terms

Algorithms, Theory, Verification

Keywords

Abstract interpretation, model checking, static analysis, program understanding, program verification, loop refinement, path-sensitive analysis, synchronous systems

1. INTRODUCTION

Imperative `while` loops with a complex control structure are common in synchronous systems. Synchronous modeling languages such as `Lustre`, `Esterel` or `Simulink/Stateflow`, yield programs that typically consist of an initialization followed by a `while-forever` loop that incorporates the tasks of input sensing, processing, state updates, and error/special case handling. These loops are characterized by complex control flow, including conditional branches, nested loops, as well as, `break` and `continue` statements. The analysis of such loops using abstract interpretation requires the use of widening and narrowing [13]. However, the use of widening may induce a loss in precision that is hard to recover from. Other techniques based on software model checking suffer from the problem of divergence or depth saturation over long running loops [4, 6, 8, 27].

We propose simple techniques based on abstract interpretation to infer a refined control structure for loops that are present in imperative programs. Our approach first partitions the paths through the loops into disjoint sets. The partitioning may be performed using syntactic considerations based on key conditional branches inside loops, assertion checks, break statements, and continue statements (or, alternatively, using data predicates). Each partition S_i of the loop is given a label a_i so that each iteration of the loop is associated with the label corresponding to the partition that contains the sequence of control states visited in the iteration. The execution of the loop is then abstracted as a regular language over the labels $a_1, \ldots, a_m$. The initial regular language L_0 consists of all the iteration sequences that are syntactically feasible in the original program. Subsequently, language L_0 is refined by ruling out infeasible sub-sequences, yielding a refined language $L \subseteq L_0$. The refinement is performed using the results of abstract interpretation or other program analysis and verification techniques. Finally, a loop

body with a refined control structure is extracted from the finite state automaton representing language L.

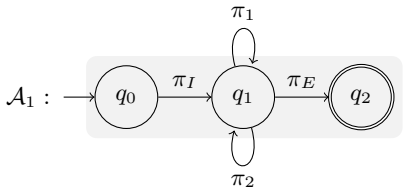
Our approach to refinement has many advantages:

- It may be performed as a *local* (intra-procedural) analysis on each loop by itself.
- Using the refined loop improves the results of abstract interpretation by minimizing the precision loss due to widening, and, in turn, enables proofs for more properties.
- Our approach is domain independent. After refining the control structure of a loop, we may apply any verification technique over the transformed program. Our experimental results show that doing so definitely helps prove more properties (see Fig. 9).
- The refinement has several other benefits, including simpler proofs for termination [10], termination analysis [11], non-termination detection [23], and automatic complexity analysis [22].

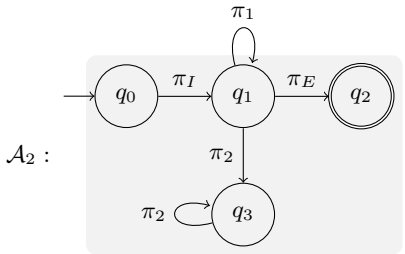
Example 1.1 (Motivating Example). *Fig. 1 shows a code fragment that computes the current year given the number of elapsed days since* 1980. *A non-termination bug in this loop is believed to have caused the recent failure behind many Microsoft Zune(tm) music players.*[1] *The loop has a head* 2 *and an exit* E. *Consider a partition of the loop paths into the following sets:*[2]

$$\begin{array}{lcl} \pi_1 & : & \{2 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 2,\ 2 \rightarrow 3 \rightarrow 9 \rightarrow 11 \rightarrow 2\} \\ \pi_2 & : & \{2 \rightarrow 3 \rightarrow 4 \rightarrow 2\} \\ \pi_I & : & \{1 \rightarrow 2\} \\ \pi_E & : & \{2 \rightarrow E\} \end{array}$$

Sets π_1, π_2 *consist of paths around the loop starting and ending at the loop head. Set* π_I *represents an initiation of the loop and set* π_E *represents the exits from the loop. The regular language* $L_0 : \pi_I(\pi_1|\pi_2)^*\pi_E$, *represents the set of all eventually terminating runs of the loop. The finite state automaton* $\mathcal{A}_1$ *shown below represents this language:*



Using static analysis, we discover that an execution of π_2 *cannot be succeeded by an execution of* π_1 *or* π_E. *This observation allows us to refine the automaton to* $\mathcal{A}_2$ *as follows:*



[1] `www.aeroxp.org/2009/01/lesson-on-infinite-loops`, viewed Aug 7th, 2009

[2] Heuristics for loop partitioning are discussed in Section 4.1.

The refinement $\mathcal{A}_2$ *derived by our approach can be shown to be a valid abstraction of the sequence of iterations of the loop. Notice that* π_2 *does not change the values of any of the program variables. Therefore, if state* q_3 *in the automaton were to be reached, we may conclude the possibility of non-termination of the loop due to repeated iterations of* π_2. *Note, however, that this is a* potential *non-termination. Its feasibility can be established using a tool such as F-Soft that can present concrete witnesses to reachability [27].*

2. RELATED WORK

As mentioned earlier, complex control loops are common in reactive (control) systems which maintain an ongoing interaction with an environment. In practice, model-based development of software using synchronous programming models expressed in formalisms such as Lustre and Esterel can be translated into C programs, each consisting of a single outer `while` loop that encapsulates the remaining control flow [24]. In such data-flow languages, Boolean flags are used to mimic the control flow. This holds true, in general, for visual development environments such as Simulink/Stateflow (tm) [30].

The automatic analysis of source code for embedded systems has become increasingly common. Formal techniques, such as *static analysis* using abstract interpretation and model checking using SAT/SMT solvers, are increasingly important for verifying properties involving *timing* (Absint [1]), *floating point* precision (Fluctuat [21]), and run-time errors (Astreé [7], Goanna[16], F-Soft [27]). In practice, the presence of complex loops poses a challenge for the techniques that underlie such tools.

Abstract interpretation [13] using numerical domains, such as *intervals* [12], *octagons* [32], and *convex polyhedra* [14] necessitate the use of widening and narrowing operators to guarantee termination over loops in the program. It is well-known that the presence of complex loops makes the application of widening during abstract interpretation challenging. Complex loops are a challenge to other verification techniques as well: they may lead to divergence in tools based on software model checking.

The weaknesses in widening and narrowing can be remedied, in part, through the use of disjunctive domains [2] or techniques for refining the control flow based on node, trace or abstract state partitioning [28, 31, 34] to obtain a degree of path sensitivity. However, in practice, disjunctive domains do not handle large programs, especially in the presence of loops. On the other hand, techniques based on trace partitioning depend critically on user-input or the choice of heuristics to work effectively. Further, existing path-sensitive analysis proposals do not fare well with loops. They resort to unrolling or unwinding, which does not prove many properties and often results in a 10-20x slowdown [3, 34]. Our work addresses the problem specifically for loops without resorting to simply unrolling them.

In particular, our previous work uses infeasible path information to refine the control flow graph of the program [3]. The technique presented here specializes this idea to the case of loops. The detection of infeasible paths in our previous work is based on the result of many forward and backward whole program analyses, which uses widening and narrowing heuristics for convergence, and, as such, is unsuitable for proving the infeasibility of loop iteration patterns. The work presented in this paper provides the following improvements: (a) it is based on a *simple* and *local* analysis that focuses

```
1:  year := 1980 ;
2:  while (days > 365){
3:      if (IsLeapYear(year)) {
4:          if (days > 366) {
5:              days − = 366;
6:              year + = 1;
7:          }
8:      } else {
9:          days − = 365;
10:         year + = 1;
11:         }
12: }
E:
```

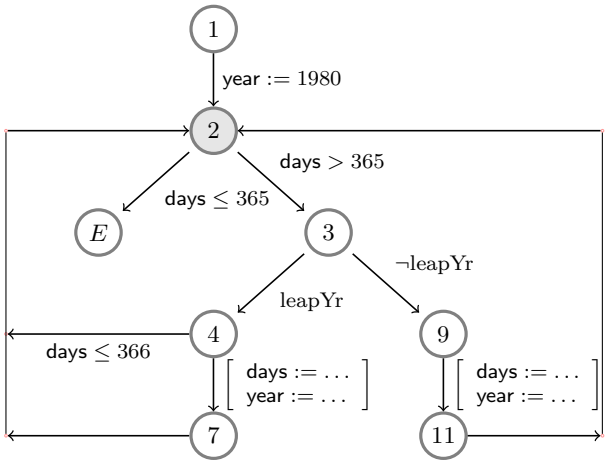


Figure 1: Potentially non-terminating loop found in the clock driver of Zune(tm)

mainly on the analysis of composed loop paths, which can be much smaller than the program as a whole, and (b) in the absence of inner loops, widening is not needed to apply our technique.

The problem of loop refinement was addressed recently by Gulwani et al. to provide improved loop complexity bound estimation [22]. Given a loop, their approach partitions the loop paths based on the control flow inside the loop. Subsequently, a loop refinement is constructed "bottom-up" by exploring the tree of possible sequences of loop iterations and maintaining an invariant for the set of states at each node of the tree. Exploration of this tree is stopped at a leaf: (a) by means of adding back-edges to a previously explored node whenever the invariant at the node subsumes the invariant at the leaf being explored, (b) when the leaf invariant is *false*, or (c) heuristically, by means of back edges and widening if the depth of the tree exceeds a maximal permissible value.

Our approach solves a similar problem as that of Gulwani et al [22]. However, we follow a "top-down" approach by simply removing infeasible sub-sequences starting from the initial loop representation. The key differentiation, therefore, lies in the nature of refinement: "top-down" in our case vs. "bottom-up". Unlike the bottom-up approach, each stage of our approach creates a valid refinement of the original loop. This is useful, in practice, wherein useful and sound intermediate results can be obtained even upon timeouts (resource limitations). In the absence of inner loops, our technique can avoid the use of widening. Furthermore, our scheme can be used in a manner complementary to that of Gulwani et al. For instance, the technique proposed by us may be applied to further refine their results. The timings and overall results obtained by our approach are competitive with that of Gulwani et al. In particular, the examples of complex iteration patterns presented by Gulwani et al. can be handled readily in our approach by considering infeasible sub-sequences of length at most 2.

Widening-upto operators [25], *lookahead* widening [19], *guided-static analysis* [20], and related approaches [35] can also be used to improve the precision loss due to standard widening on loops with iteration-dependent control flow. The advantage of these approaches is that they avoid the size blowup that can be caused by refinement. On the other hand, the use of widening heuristics in this process makes it difficult to control the precision.

In practice, using a powerful abstract domain *locally* to refine a small portion of a program can be used as a preprocessing step for a whole-program analysis using a simpler abstract domain. Therefore, refinement techniques proposed here can be useful even in the presence of recently proposed analysis techniques that do not require widening to compute the fixed point such as *constraint-based analysis* [9], *policy iteration* [17], and *strategy iteration* [18] techniques.

3. PRELIMINARIES

Programs will be represented by their *control flow graph* (CFG) representation. We assume that the CFG is built from a well-structured imperative program (without using `goto`-statements). Specifically, each loop is assumed to be *reducible*, consisting of a single loop head, which dominates all the nodes inside the loop. Edges from the loop nodes back to the head are called *back edges*. Edges from nodes inside a loop to a node outside are called *exit edges*. We assume that all exit edges from a loop have a single target. For simplicity of presentation, we consider loops that do not contain function calls. Non-recursive function calls can be handled using standard techniques such as inlining. The techniques described here can be extended to treat recursive function calls also. Formally, a CFG Π is a tuple $\langle N, E, V, \rho, n_0, n_e \rangle$, 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, $n_e \in N$ is an exit location, and V is a set of typed (global) program variables. Each edge $a \rightarrow b \in E$ is labeled by a *transition relation* $\rho_e(V, V')$, a first-order assertion over *current-state* variables V and *next-state* variables denoted by V'.

3.1 Loops and Fragments

This paper discusses control-based abstractions and refinements of loops inside programs. Therefore, we will formally define loops in a manner that makes our assumptions regarding their behavior explicit.

Def. 3.1 (Loops). *A loop L (as a subset of a CFG Π) consists of a tuple $\langle N_L, E_L, n_h, n_x \rangle$ of nodes N_L, edges $E_L \subseteq$*

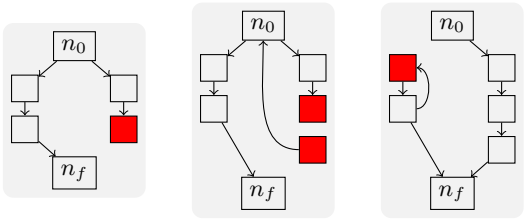


Figure 2: Examples of flow graphs that are not continuous fragments. (Shaded nodes do not satisfy some clause in Def. 3.2.)

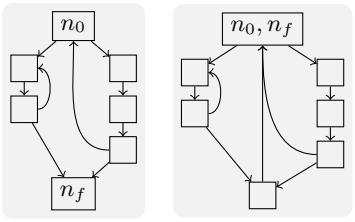


Figure 3: Examples of continuous fragments.

$N_L \times N_L$, *a* loop head $n_h \in N_L$, *and* exit node $n_x \in N_L$. *The subgraph induced by the nodes* $N_L - \{n_x\}$ *and edges* E_L *form a strongly connected component. We assume that there are no outgoing edges in* E_L *starting from the exit* n_x.

A loop M is *nested* inside a loop L iff $N_M \subseteq N_L$ and $E_M \subseteq E_L$. As a result, the loop head and the exit node of the loop M are a part of loop L. A *loop control path* π consists of a subset of nodes and edges in the loop formed by a simple path from the loop head n_h back onto itself. If a loop head of a nested loop M lies on such a path, the nodes and edges in M are assumed to lie on the path π.

Def. 3.2 (Continuous Fragment). *A* continuous fragment S *of a loop between nodes* n_0 *and* n_f *(possibly,* $n_0 = n_f$ *= loop head) consists of a subset of loop nodes* N_S *and edges* E_S *such that:*

(a) $n_0, n_f \in N_S$ *and for each edge* $e : m \to n$, $m, n \in N_S$.

(b) For each node $n \in N_S$ *(with the exception of* n_f*, if* $n_0 \neq n_f$*), at least one outgoing edge belongs to* E_S.

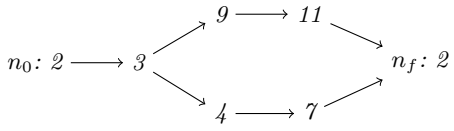
(c) For each node $n \in N_S$ *(with the exception of* n_0*, if* $n_0 \neq n_f$*), at least one incoming edge belongs to* E_S.

(d) Every node $n \in N_S$ *is (control) reachable from* n_0 *through a path in* (N_S, E_S).

(e) Every node $n \in N_S$ *has a path to* n_f *in* (N_S, E_S).

Example 3.1. *Figs. 2 and 3 illustrates the definition of continuous fragment further through negative and positive examples of the concept. The CFGs in Fig. 2 are not continuous fragments, whereas the CFGs in Fig. 3 satisfy the definition.*

Going back to the loop shown in Fig. 1, the set of nodes $\{2, 3, 4, 7, 9, 11\}$ *along with the edges* $2 \to 3$, $3 \to 4$, $4 \to 7$, $7 \to 2$, $3 \to 9$, $9 \to 11$ *and* $11 \to 2$ *form a continuous fragment. We depict such a fragment as:*



Note that $n_0 = n_f$ *in this case. The clauses of the definition for a continuous fragment are satisfied in this example.*

Def. 3.3 (Fragment Partitioning). *A fragment partitioning of a loop* $L : \langle N, E, n_h, n_x \rangle$ *consists of a set of* loop continuous fragments $P_1, \ldots, P_m$ *from* $n_h \rightsquigarrow n_h$, *and a set of* exit continuous fragments $Q_1, \ldots, Q_k$ *from* $n_h \rightsquigarrow n_x$, *such that (a) the set of nodes and edges in every walk (in the graph-theoretic sense, Cf. [26]) from* n_h *back to itself is contained in some loop continuous fragment* P_i *and (b) every walk from* n_h *to* n_x *is contained in some exit fragment* Q_i.

Given a partitioning of a loop into fragments, the set of edges (and nodes) visited in an iteration of a loop must belong to some fragment and the set of paths from the loop head to the exit node must belong to some exit fragment in the partitioning.

Example 3.2. *Sets* π_1, π_2 *form loop fragments and set* π_E *forms an exit fragment of a partitioning of the loop in Example 1.1.*

There are many ways of partitioning a given loop into fragments. For instance, we may consider the set of all control paths through the CFG from $n_h \rightsquigarrow n_h$ as a single loop fragment and the set of all paths from $n_h \rightsquigarrow n_x$ as a single exit fragment. Later in our discussion, we will demonstrate that such a partitioning does not provide a means for distinguishing loop iterations. Better partitioning schemes should classify iterations based on the conditional branches, back edges, and loop exits that are exercised in each iteration. We will assume for the time being that such a partitioning scheme has been specified. In section 4.1, we discuss some heuristics for loop partitioning.

3.2 Abstract Interpretation

Abstract interpretation [13] provides a technique for proving properties about a program's behavior by mapping its behavior from the concrete domain of states onto an abstract domain representing sets of states. Let Σ represent the set of all possible states of a program P. The semantics of an edge e is specified by means of its *concrete post-condition*, which maps a set $S \subseteq \Sigma$ to its concrete post-condition $S' : post_\Sigma(S, e)$ representing the set consisting of all states that are reachable starting from some state $s \in S$ and executing the edge e.

Def. 3.4 (Inductive Map). *A flow-sensitive map* $\eta : N \mapsto 2^\Sigma$ *associates a set of states* $\eta(n)$ *with each node* n *of a CFG (or a loop). A flow-sensitive map is* inductive *(or a fixed point) for a CFG with nodes* N *and edges* E *iff*

$$(\forall\ (e :\ m \to n)\ \in E)\ post_\Sigma(\eta(m), e) \subseteq \eta(n)\,.$$

Sets of states are represented by logical assertions over a suitable theory (e.g., first order theory of numbers). Let $\mathcal{C}(\Sigma)$ represent the space of such assertions (over program states) and $\models$ represent the semantic entailment relation

between assertions. An assertion φ in this theory denotes a set of states $[\![\varphi]\!] \subseteq \Sigma$ that satisfy φ.

An abstract domain $\Gamma : (L, \sqsubseteq, \alpha, \gamma)$, consists of a lattice $(L, \sqsubseteq)$ along with an abstraction function $\alpha : \mathcal{C}(\Sigma) \mapsto L$ mapping sets of states onto abstract objects and $\gamma : L \mapsto \mathcal{C}(\Sigma)$ mapping abstract objects onto concrete sets of states. Corresponding to the concrete post-condition, we define the abstract post-condition $post_L$ (or simply $post$) that over-approximates the effect of executing e on a set of states:

$$(\forall\ S, e)\ post_\Sigma(S, e)\ \models\ \gamma(post_L(\alpha(S), e))\,.$$

Given a program P and an abstract domain Γ, the technique of abstract interpretation iteratively computes a flow-sensitive *fixed point* $\nu : N \mapsto L$, mapping each node to an abstract domain object such that

$$(\forall\ (e :\ m \to n)\ \in E)\ post_L(\nu(m), e) \sqsubseteq \nu(n)\,.$$

Upon observing the similarity between the fixed point above and the inductive map in Def. 3.4, we conclude that ν is a fixed point in Γ iff $\gamma \circ \nu$ is an inductive map (over the concrete domain). As a result, abstract interpretation can be seen as a technique for computing an inductive map η for a given program.

In practice, abstract interpretation is carried out using powerful abstract domains such as *intervals*, *octagons*, and *polyhedra* [12, 32, 14]. These domains can be used separately or in combination to compute powerful invariants that capture the behavior of a program. In the ensuing discussion, we will assume that a powerful abstract domain or a combination of many abstract domains are used to analyze program (whole or fragments) and generate a *concrete* inductive map η (over sets of states). Conceptually, applying the concretization map γ on the result of the abstract interpreter, yields such a map.

Def. 3.5 (Fragment Post-condition). *Let $P : n_e \rightsquigarrow n_f$ be a continuous fragment defined by the nodes N_P and edges E_P. Given an assertion $\varphi_e \in \mathcal{C}(\Sigma)$, the fragment post-condition for P (denoted by $\mathsf{post}^P_\eta(\varphi_e)$) is φ_f iff there exists a map η over N_P such that $\varphi_e \models \eta(n_e)$, $\eta(n_f) \models \varphi_f$, and η is inductive over (N_P, E_P), i.e.,*

$$\forall\ e \in E_P,\ post(\eta(m), e) \models \eta(n)\,.$$

A fragment post-condition captures the concept of propagating a given assertion φ_e forward through a fragment $P : n_e \rightsquigarrow n_f$ to obtain a "post-condition" φ_f that contains all states that are reachable at location n_f starting from some state at location n_e satisfying φ_e. We assume that a fixed abstract interpreter with some suitable combination of abstract domains is used to yield the set of states φ_f.

4. AN ABSTRACTION FOR LOOPS

In this section, we present an abstraction for representing loop iterations. First, we introduce a technique for labeling the iterations of a loop based on a partition of the loop into continuous fragments. Second, we present an abstraction that represents loop iterations as a sequence of these labels and present techniques for verifying a given abstraction using static analysis.

4.1 Loop Partitioning Heuristics

First, we discuss some useful heuristics for automatically partitioning loops. We assume that inner loops (if present) are all represented as a single node for the purposes of loop partitioning. This is achieved by removing back-edges from the outer loop and computing a maximal strongly connected component (MSCC) decomposition. In general, there are numerous useful heuristics for partitioning the loop paths. We describe a few schemes that may be useful in practical settings:

Full Path-Partitioning: In the absence of inner loops, a full partition places each control path around the loop in a partition by itself. For loops with complex control flow, this scheme can lead to a combinatorial explosion in the number of partitions.

Backedge-specific Partition: A natural scheme for partitioning loops with complex control flow consists of placing all cycles that traverse the same backedge into a single partition. In practice, `continue` statements are frequently used to handle special cases by skipping parts of the loop.

Induction Variable Update: Many loop iterations can be classified based on the updates to some induction variable. Such a scheme may help in termination analysis where termination depends on the updates to an induction variable. The partition scheme used in Ex. 1.1 is an instance of such a scheme.

Subset of Branches: The partitioning of paths may be based on the outcome of a subset of the branches in the loop. Such subsets may be chosen using syntactic slicing based on some variables or statements of importance [22].

In practice, a partitioning scheme may be based on a combination of some of the above considerations.

4.2 Representing Loop Iterations

Let $L : \langle N, E, n_h, n_x \rangle$ be a loop with nodes N, edges E, loop head n_h and exit n_x. Let Π be a partitioning of this loop, consisting of *loop fragments* $P_1, \ldots, P_m$, wherein $P_i : n_h \rightsquigarrow n_h$ and a set of *exit fragments* $Q_1, \ldots, Q_k$ where $Q_i : n_h \rightsquigarrow n_x$. Since Π is a partitioning of the loop's control structure, any iteration of the loop can be ascribed to some fragment P_i and furthermore, the exit from a loop can be ascribed to some fragment Q_i. In general, this can be forced by refining the initial partition using a Boolean completion. We assume, for convenience, that the partition is *disjoint* so that no two partitions share the same control path.

Def. 4.1 (Label Alphabet). *The set Ω:*

$$\Omega = \{p_1, \ldots, p_m, q_1, \ldots, q_k, \iota\}\,,$$

forms the label alphabet for a partition Π if for each loop fragment P_i, there exists a unique corresponding label $p_i \in \Omega$ and for each exit fragment Q_i an exit label $q_i \in \Omega$. Furthermore, we assume a special label ι denoting the initial entry into the loop.

Given a partition Π with labels Ω, each iteration of the loop can be labeled with an alphabet p_j and each exit by an alphabet q_j. Therefore, the sequence of labels encountered in any terminating execution of the loop is a word in the language $R_\top$:

$$R_\top :\ \iota \cdot (p_1 \mid \cdots \mid p_m)^* \cdot (q_1 \mid \cdots \mid q_m)\,.$$

Language R_T is suggested by the *natural representation* of the loop in the program. However, the set R_* of actual

sequences observable in any concrete execution of the loop is, in general, a sub-language of $R_\top$. Furthermore, R_* need not be a regular language. Our goal here is to construct a regular language R' such that $R_* \subseteq R' \subseteq R_\top$. For this, there are two possible approaches:

(a) An *abstract interpretation* over the lattice of regular languages $R \subseteq R_\top$ by expressing the required language R as a fixed point over a monotone operator. The work of Gulwani et al. [22] is roughly an instance of this approach.

(b) A *top-down* refinement that simply refines the language $R_\top$ by eliminating *forbidden sub-sequences* up to a certain length using abstract interpretation. This is the approach we take in this paper.

Example 4.1. *Consider the loop and its partitioning discussed in Ex. 1.1. We associate labels p_1 and p_2 corresponding to the loop fragments π_1 and π_2, respectively. Similarly, the label ι corresponds to π_I and finally the label q_1 corresponds to π_E.*

The language $R_\top$ corresponds to that of the automaton $\mathcal{A}_1$ from Ex. 1.1. Finally, a refinement of the language $R_\top$ corresponds to $\mathcal{A}_2$. In fact, for this example the language $L(\mathcal{A}_2)$ is the "best refinement" possible, i.e, $R_* = L(\mathcal{A}_2)$.

5. TOP-DOWN REFINEMENT

In this section, we present a scheme to refine the label language $R_\top$ that is induced by the partitioning Π of a given loop L into loop and exit fragments. The scheme is based on identifying forbidden sub-sequences using abstract interpretation and removing words that contain such sub-sequences from the language $R_\top$.

Let L be a loop with a given initial condition $\Theta \subseteq 2^\Sigma$ representing the set of all the possible initial states at the start of its execution. Let Ω be the label alphabet corresponding to a partitioning Π. A *sub-sequence* of size $j > 0$ consists of a sequence of labels $a_1 \dots a_j$ such that $a_i \in \Omega - \{\iota\}$ and for $i < j$, $a_i \notin \{q_1, \dots, q_k\}$. An *initialized sub-sequence* is of the form $\iota a_1 \dots a_j$, where $a_1, \dots, a_j$ is a valid sub-sequence.

Example 5.1. *Returning to Ex. 4.1, legal sub-sequences include $\iota p_1 q_1$, $p_1 p_2$, $p_2 p_2 p_2$. The sequences $q_1 \iota$, $q_1 p_1$, and $p_1 q_1 q_2$ are not legal sub-sequences.*

Def. 5.1 (Fragment Composition). *Let $S_1 : n_1 \rightsquigarrow n_2$ and $S_2 : n_2 \rightsquigarrow n_3$ be continuous fragments. The composition $S_1 \circ S_2$ consists of a graph G whose nodes are given by the disjoint sum of the nodes:*

$$N(S_1 \circ S_2) = (N(S_1) \times \{1\}) \cup (N(S_2) \times \{2\})$$

and edges are given by

$$E(S_1 \circ S_2) \quad : \quad \{(m,i) \to (n,i) | m \to n \in E(S_i), i \in [1,2]\} \bigcup \{(n_2,1) \to (n_2,2)\}$$

The transition relations associated with each edge remain unchanged. The newly added edge $(n_2,1) \to (n_2,2)$ is labeled by the identity *relation* nop.

The composition operation may be extended to an uninitialized sub-sequence $s : a_1 \dots a_j$. Let $P(s)$ be the fragment denoted by the composition of fragments in s, $P(s) : P(a_1) \circ P(a_2) \circ \cdots P(a_j)$.

Lemma 5.1. *The composition of fragments $S_1 : n_1 \rightsquigarrow n_2$ and $S_2 : n_2 \rightsquigarrow n_3$ forms a continuous fragment $S_1 \circ S_2 : (n_1,1) \rightsquigarrow (n_3,2)$.*

A sub-sequence $a_0, \dots, a_j$ is *forbidden* iff no execution of the loop may traverse the fragments represented by $a_0, \dots, a_j$ consecutively. To prove that a given sub-sequence may be forbidden, we use abstract interpretation over the graph obtained by composing the various fragments corresponding to $a_0, \dots, a_j$.

Def. 5.2 (Forbidden Sub-sequence). *An uninitialized sub-sequence $s : a_1, \dots, a_j$ is* forbidden *(w.r.t a given abstract interpreter) iff for the fragment $P(s) : m \rightsquigarrow n$ obtained by the composition of fragments in s, $\mathsf{post}^{P(s)}([\![true]\!]) \equiv \emptyset$.[3] In other words, an abstract interpretation of the fragment $P(s)$ with node m labeled by the assertion true establishes the invariant false at node n.*

For an initialized sub-sequence $s : \iota a_1, \dots, a_j$, let $P(s)$ be the composition of fragments corresponding to $a_1, \dots, a_j$. The sequence is forbidden iff $\mathsf{post}^{P(s)}([\![\Theta]\!]) \equiv \emptyset$. In other words, node m is initialized using the initial condition Θ.

Theorem 5.1. *Let s be a forbidden sub-sequence. No execution of the loop can visit the labels in s in sequence.*

Proof. Proof follows from the soundness of abstract interpretation. □

Example 5.2. *Returning to Ex. 1.1, we recall the fragment labels from Ex 4.1. The composition is shown in Fig. 4. The sequence $p_2 p_1$ is forbidden. This may be obtained by means of an abstract interpretation using a combination of linear inequalities with uninterpreted predicates (leapYr). The condition $\mathsf{days} \leq 366$ in the edge $4 \to 2$ directly contradicts the condition $\mathsf{days} > 366$ for the edge $4 \to 7$. Similarly the condition leapYr in edge $3 \to 4$, directly contradicts $3 \to 9$.*

Let s be a forbidden sub-sequence over Ω. The initial language $R_\top$ may be refined by subtracting loop iterations that traverse s. To this end, the refinement of a language R_T w.r.t a forbidden sub-sequence s is derived by simply subtracting the regular language $R_s : \Omega^* s \Omega^*$ of all strings from R_T that contain s as a sub-sequence. Let $s_1, \dots, s_k$ be a set of forbidden sub-sequences. We may refine the language $R_\top$ by subtracting such sub-sequences:

$$R' = R_\top - \bigcup_i R_{s_i} .$$

Example 5.3. *The automaton $\mathcal{A}_2$ in Ex. 1.1 is obtained by subtracting the forbidden sequences $p_2 p_1$ and $p_2 q_1$ from the language $R_\top : \iota(p_1|p_2)^* q_1$.*

Loop structure refinement can therefore be achieved by discovering forbidden sub-sequences and refining the language of iteration labels using the forbidden sub-sequences. A practical scheme consists of considering all possible iteration sequences of up to some length $K > 0$ and removing the sequences that are forbidden.

[3]The assertion *true* can be strengthened by a previously obtained invariant of the loop L

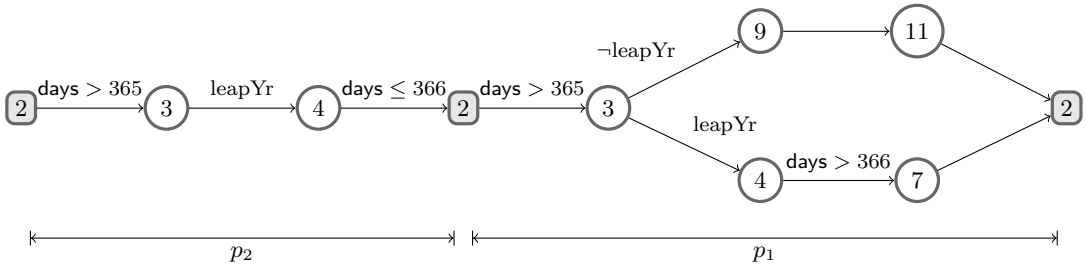


Figure 4: A composition of two fragments corresponding to loop iterations labeled p_1, p_2.

Intermediate Fragments

We may improve the process of refinement by computing intermediate fragments for each forbidden sub-sequence. The goal here is to generalize a forbidden sub-sequence $a_1 a_2 \ldots$ to a general form $(a_{i_1^1} + \cdots + a_{i_k^1}) * a_1 (a_{i_1^2} + \cdots + a_{i_k^2}) * a_2 \ldots$ using a proof of infeasibility.

Example 5.4. *Fig. 5 shows an example loop with calls to functions* `foo` *and* `bar`*. Our goal is to establish that no calls to* `foo` *can occur after a call to function* `bar`*. In order to do so, we partition the loops into fragments wherein* `foo` *is called,* `bar` *is called and neither of these functions are called:*

$$P_1 \;:\; 2 \xrightarrow{i \le N} 3 \to 4 \xrightarrow{j \le M} 5 \xrightarrow{j++} 7 \to 13 \xrightarrow{i++} 2$$
$$P_2 \;:\; 2 \xrightarrow{i \le N} 3 \to 9 \xrightarrow{j \ge M} 10 \to 12 \xrightarrow{j++} 13 \xrightarrow{i++} 2$$
$$P_3 \;:\; 2 \to 3 \to \{4, 9\}, 4 \to 7, 9 \to 11, \{7, 9\} \to 13 \to 2$$

The notation $\{a, b\} \to c$ is used to denote the set of edges $a \to c$ and $b \to c$. Let p_1, p_2, p_3 be the labels for P_1, P_2, P_3 respectively.

By using abstract interpretation, we can prove that the sub-sequence $p_2 p_1$ is forbidden. The comparison $j \ge M$ in the edge $9 \to 10$ followed by the update $j{+}{+}$ in the edge $5 \to 7$ make the comparison $j \le M$ in edge $4 \to 5$ infeasible. Furthermore, we may establish forbidden sub-sequences of the form $p_2 p_3 p_1$, $p_2 p_3^2 p_1$, and so on.

We now present a technique for inferring a larger set of forbidden sub-sequences from the proof of infeasibility of a single sub-sequence. Specifically, our technique extracts a minimal set of invariants from the composition of $p_2 p_1$ and uses these invariants to efficiently infer the infeasibility of all sub-sequences that match $p_2 p_3^* p_1$.

For simplicity, consider a forbidden sub-sequence (possibly initialized) of length 2 obtained by the composition of two fragments $S : n_1 \rightsquigarrow n_2$ and $T : n_2 \rightsquigarrow n_3$, such that $\mathsf{post}^{S \circ T}(\varphi) \equiv \mathit{false}$ for a suitable initial condition φ. The technique presented here extends naturally to larger sub-sequences. We recall the definition of fragment composition from Def. 5.1. Node $(n_2, 2)$ in this composition is defined as the *interface node* for $S \circ T$.

Let $\eta : N(S \circ T) \mapsto \mathcal{C}(\Sigma)$ be some inductive invariant map that establishes $\eta(n_3, 2) \equiv \mathit{false}$ and let $\psi : \eta(n_2, 2)$ be the invariant labeling the interface node. Let R be a fragment (different from S, T) such that $\mathsf{post}^R(\psi) \models \psi$. In other words, the fragment R *preserves* the invariant ψ.

Theorem 5.2. *If $S \circ T$ is an infeasible fragment composition with an interface invariant ψ, and a fragment R preserves the invariant ψ, then compositions of the form $S \circ R^i \circ T$ are infeasible for all $i \ge 0$.*

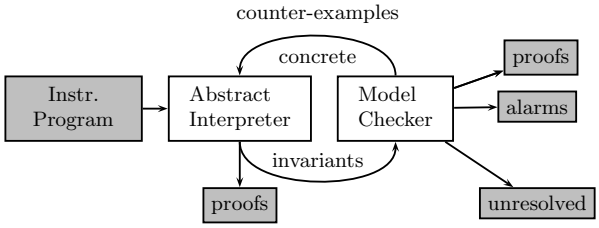


Figure 7: The F-Soft C program verification platform.

Example 5.5. *Consider the fragments P_1, P_2 introduced in Ex. 5.4. The composition $P_2 \circ P_1$ is shown in Fig. 6(a) along with the invariants. The invariant at the interface node is $\psi : \; j > M \;\wedge i \le N + 1$. The preservation of this invariant by the fragment P_3 is also shown in Fig. 6(b).*

It should be noted that the conjunct $i \le N + 1$ in ψ is unnecessary for proving the infeasibility of $P_2 \circ P_1$.

In general, the removal of invariant conjuncts superfluous to the proof of infeasibility of a composition has the effect of weakening the interface invariant ψ, which enables a larger set of intermediate fragments to preserve this invariant.

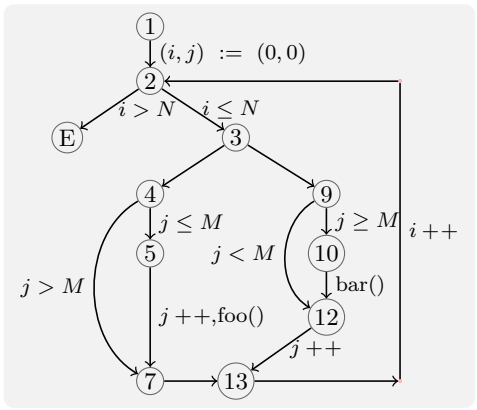
Extensions to Push-Down Systems Our approach readily extends to handling recursive functions that can be modeled as a push-down system [15]. We recall that the intersection of a context-free language with a regular language yields a context-free language. This property ensures that our approach of refining iteration sequences by means of subtracting an infeasible regular language can be used to treat recursive programs.

6. IMPLEMENTATION

The algorithms discussed in this section have been implemented as a part of the F-Soft C program analysis framework [27]. F-Soft combines source-level instrumentation, abstraction, and type lowering along with static analysis using abstract interpretation and bounded model checking using SAT solvers. This enables us to prove properties as well as find concrete error traces for violations.

Instrumentation. Given a C program, we systematically instrument pointers and arrays to track their allocated status (pointer to stack, heap, invalid, etc.), allocated extents, and sentinels for the null terminator character. The effect of assignments to pointers, pointer arithmetic, pointer indirection, and operations such as casting are accurately modeled. In particular, pointer indirections $*p$ are handled soundly by adding conditional branches over the possible points-to set of the base pointer p. Our tool abstracts the programs in

```
1:      (i,j)  := (0,0);
2:      while ( i ≤ N) {
3:          if (nondet) {
4:              if (j ≤ M) {
5:                  j  := j+1;
6:                  foo(..);
7:              }
8:          } else {
9:              if (j ≥ M) {
10:                 bar(..);
                }
12:             j  := j +1;
            }
13:         i  := i +1;
14:     }
```



Control Flow Graph

Figure 5: Loop with calls to function foo and bar.

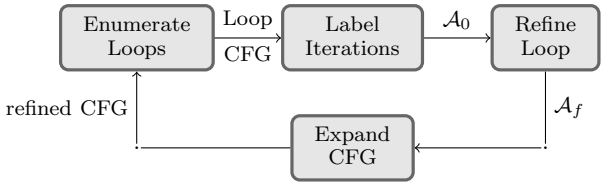


Figure 8: Loop Refinement Flow.

two ways: (a) We model the contents of arrays at some specific indices (first and last, for example). (b) We expand field accesses of recursive structures such as linked lists up to a fixed depth. Accesses to unmodeled elements of arrays, pointers to untracked locations, and fields beyond the depth bound result in non-deterministic values.

After instrumentation and various other type lowering transformations, we obtain a CFG where all variables have basic types such as integers, floating points, and various types of integer types such as *short*, *unsigned*, etc. This vastly simplifies the implementation of our static analyses and model checking engines. However, as a result, we may obtain false alarms due to the abstraction of array elements and recursive data structures, as described above.

Static Analysis. Abstract interpretation [13] is used in F-Soft as the main *proof engine*. Our abstract interpreter is inter-procedural, flow and context sensitive. It is built in a domain-independent and extensible fashion, allowing for various abstract domains such as *constants*, *intervals* [12], *octagons* [32], *symbolic ranges* [33] and *polyhedra* [14]. These domains are applied in increasing order of complexity. After each analysis is run, the proved properties are removed and the model is simplified by constant propagation and slicing. The resulting model is analyzed by a more complex domain.

Loop Refinement

Figure 8 shows the refined flow on the abstract interpreter engine that incorporates the techniques for loop refinement discussed in this work. The refinement consists of first enumerating program loops. The enumeration is performed in the order of nesting depth in the code. A loop that is nested innermost is first refined before refining an outer loop. (With numerical domains, the effects of widening are more pronounced on the outer loops. By performing a refinement of the inner loop, we have a better chance of improving the invariants obtained for the outer loops with less blowup in size. Our approach also works if outer loops are refined first.) The loop iterations are labeled based on the assertions, exit points (break statements), and continue statements visited in the iteration. The initial regular approximation is formed using the labels and the refinement is carried out as described in section 5 by identifying forbidden sub-sequences of depth $k > 0$. In our experiments, we bound k to 2.

After each loop is refined, its CFG is modified in place using the expanded form obtained by the automaton representing the possible sequence of loop labels. The resulting CFG is larger than the original CFG. However, with the loop structure refined, we obtain better results from the static analysis, especially using widening.

7. EXPERIMENTS

Our implementation was evaluated on a number of small, but challenging, loops drawn from academic benchmarks consisting of synchronous programs [24], loop refinement benchmarks [22], (synchronous) models of complex reactive systems [5], and the C code generated corresponding to the state-chart of the *vehicle control mode* distributed as an example of larger scale embedded system design in Simulink [29]. Each benchmark example was translated into the C language using built-in functions such as `nondet()` and `assume()` that are interpreted by our tool to model non-determinism. Most of the examples consisted of a single non-nested loop. However, some of the complex models consisted of inner nested loops which were encoded using flags. The refinement automatically discovers such nested loops.

Experiments consist of first analyzing the model without loop refinements followed by an analysis of the model after loop refinement. We used the full path-partitioning heuristic described in section 4.1 for partitioning the loop. The invariants computed at loop heads are reported for the sake of comparison. All benchmarks except *consprodjava* and *Veh.Clim.Ctrl* were (manually) annotated with a set of safety properties.

Table 9 shows the results of our experiments. The ta-

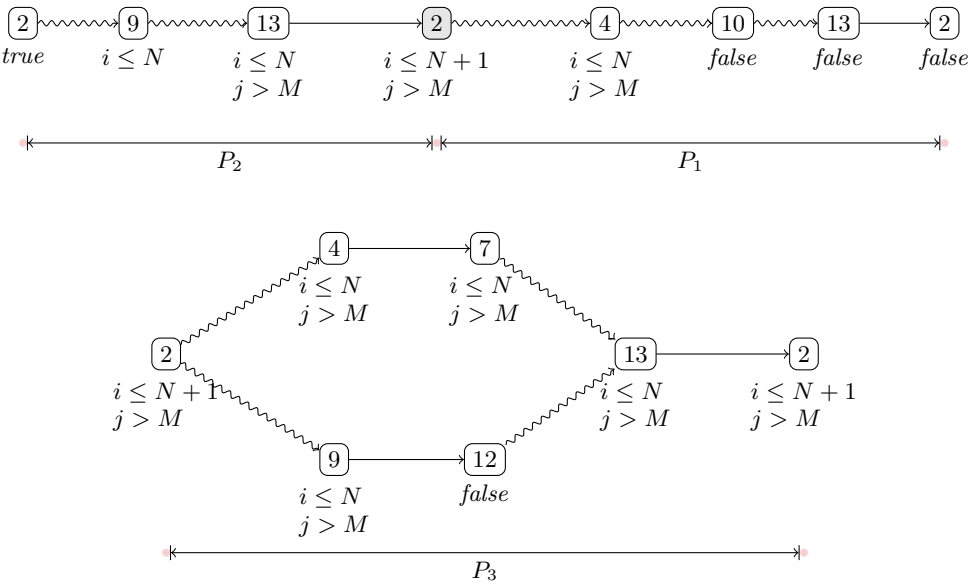


Figure 6: The interface invariant from the composition of P_2 and P_1 (shown above) is preserved by the fragment P_3.

Name	SLOC	Num. Basic Blocks			# Loop	Refine	Impr.	# Aut.	Total	w/o Refinement		w/ Refinement	
		orig.	ref.	simp.	Seg.	Time (s)	Invar?	States	Props	Prf.	Time (s)	+Prf.	Time (s)
Loop1 [22]	36	19	43	42	3	.01	SAME	4	2	2	.02	0	.02
Loop2 [22]	46	24	80	72	4	.02	SAME	4	2	2	.04	0	.08
Loop3 [22]	45	34	118	18	5	.05	YES	4	3	2	.03	+1	.05
Loop4 [22]	43	27	80	79	4	.03	YES	4	1	1	.02	0	.04
Loop5 [22]	54	36	75	75	4	.02	SAME	4	4	4	.04	0	.08
Loop6 [22]	38	21	48	41	3	.02	INCOMP	4	2	1	.01	0	.01
Berkeley [5]	96	59	523	267	10	.5	YES	9	2	0	.3	+2	.04
Synapse [5]	62	42	353	211	9	.3	YES	9	2	1	.1	+1	.02
MESI [5]	79	66	997	316	16	1.2	YES	15	5	3	.2	+2	0.05
MOESI [5]	91	94	2711	467	23	5.2	YES	19	8	3	.8	+5	.16
consprodjava [5]	270	123	8089	542	50	15.4	YES	51	n/a	n/a	6.6	n/a	5
Zune	41	25	68	58	4	.03	YES	5	2	1	.03	0	.04
Lift [28]	81	51	642	274	12	.7	YES	11	2	0	.2	+2	.2
CDCntrl [28]	88	49	1123	609	15	.55	YES	9	2	0	.3	+1	.75
Veh.Clim.Ctrl [29]	538	149	3161	964	9	72.3	YES	10	n/a	n/a	9.7	n/a	45.2

Figure 9: Experimental results for the benchmarks. **Legend — SLOC:** Simplified Lines of Code; number of basic blocks **orig.:** before refinement, **ref.:** immediately after refinement, **simp.:** after refinement and simplification; #Loop Seg.: size of the partition; **Impr. Invar?** comparison of invariants before/after refinement; **Prf.**: Number of proofs.

ble compares the invariants, number of properties proved (**Prf.**), the size of the models, and the time taken to analyze before and after refinement. Note that the number of proofs after refinement (last but one column labeled **+Prf.**) reports the number of proofs *in addition* to the properties proved before refinement. In all but one example, the process of refinement yielded an invariant that was at least as strong as the invariant derived before refinement. For a fair comparison, our refinement technique itself *does not* utilize the invariant computed by the original analysis pass. For one example (Loop6), the invariants after refinement were logically incomparable with the invariants before (denoted **INCOMPAR**). This happens due to the iteration strategy and the way widening/narrowing is used by our abstract interpreter. In practice, adding the invariants computed by the first pass back into the model during refinement can guarantee that the analysis of the refined loop produces invariants at least as powerful as the original loop. The table also reports on the size of the automaton generated as part of the refinement (**#Aut. States**).

The results clearly show that our loop refinement technique can scale to relatively large and complex loops in a short amount of time. While the initial result of the refinement can cause a blowup in the size of the loop, repeated simplification of the refined loop can be applied to reduce its size. The refinement time seems to scale exponentially with the number of labels. Therefore, the design of heuristics to partition the loop paths effectively is key to the application of our technique.

In most examples, the invariants obtained by analyzing the refined loop (using the interval, octagon and polyhedral abstract domains) were strictly more powerful than the original loop invariants. Furthermore, in the case of the CD controller example from Jeannet et al. [28], the loop refinement proved the property in question with a refined automaton that was comparable to Jeannet et al.

8. CONCLUSION

We have demonstrated a simple and elegant technique for loop refinement in which the loop iterations are represented by a regular language and the loop is refined by removing infeasible sub-sequences from the initial regular language using a proof technique such as abstract interpretation. We have applied it to many complex benchmark that are representative of the loops present in real embedded systems. Our experimental results demonstrate that our refinement can enhance the power of existing static analysis techniques. Furthermore, the proposed technique can also improve techniques that prove termination or establish non-termination.

9. REFERENCES

[1] Absint WCET timing analysis tool. `http://www.absint.com/`.

[2] Bagnara, R., Hill, P. M., Ricci, E., and Zaffanella, E. Precise widening operators for convex polyhedra. In *Static Analysis Symposium* (2003), vol. 2694 of *LNCS*, Springer, pp. 337–354.

[3] Balakrishnan, G., Sankaranarayanan, S., Ivančić, F., Wei, O., and Gupta, A. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *SAS* (2008), vol. 5079 of *LNCS*, pp. 238–254.

[4] Ball, T., and Rajamani, S. K. The SLAM toolkit. In *CAV* (2001), vol. 2102 of *LNCS*, pp. 260–264.

[5] Bardin, S., Finkel, A., Leroux, J., and Petrucci, L. FAST: Fast accelereation of symbolic transition systems. In *CAV* (July 2003), vol. 2725 of *LNCS*.

[6] Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. The software model checker BLAST. *STTT 9*, 5-6 (2007), 505–525.

[7] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. A static analyzer for large safety-critical software. In *PLDI* (June 2003), pp. 196–207.

[8] Clarke, E., Kroening, D., and Lerda, F. A tool for checking ANSI-C programs. In *TACAS* (2004), vol. 2988 of *LNCS*.

[9] Colón, M., Sankaranarayanan, S., and Sipma, H. Linear invariant generation using non-linear constraint solving. In *CAV* (July 2003), vol. 2725 of *LNCS*, Springer, pp. 420–433.

[10] Colon, M., and Sipma, H. Synthesis of linear ranking functions. In *Tools and Algorithms for Construction and Analysis of Systems* (April 2001), vol. 2031 of *LNCS*.

[11] Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. In *PLDI* (2006).

[12] Cousot, P., and Cousot, R. Static determination of dynamic properties of programs. In *Proc. ISOP'76* (1976), Dunod, Paris, France, pp. 106–130.

[13] Cousot, P., and Cousot, R. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977), pp. 238–252.

[14] Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among the variables of a program. In *POPL'78* (Jan. 1978), pp. 84–97.

[15] Esparza, J., Hansel, D., Rossmanith, P., and Schwoon, S. Efficient algorithms for model checking pushdown systems. In *CAV* (2000), vol. 1855 of *LNCS*.

[16] Fehnker, A., Brauer, J., Huuck, R., and Seefried, S. Goanna: Syntactic software model checking. In *Auto. Tech. for Verif. & Analysis* (2008).

[17] Gaubert, S., Goubault, E., Taly, A., and Zennou, S. Static analysis by policy iteration on relational domains. In *ESOP* (2007), vol. 4421 of *LNCS*, pp. 237–252.

[18] Gawlitza, T., and Seidl, H. Precise fixpoint computation through strategy iteration. In *ESOP* (2007), vol. 4421 of *LNCS*, pp. 300–315.

[19] Gopan, D., and Reps, T. W. Lookahead widening. In *CAV* (2006), vol. 4144 of *LNCS*, pp. 452–466.

[20] Gopan, D., and Reps, T. W. Guided static analysis. In *SAS* (2007), vol. 4634 of *LNCS*, pp. 349–365.

[21] Goubault, E., and Putot, S. Static analysis of numerical algorithms. In *SAS* (2006), vol. 4134 of *LNCS*, Springer, pp. 18–34.

[22] Gulwani, S., Jain, S., and Koskinen, E. Control-flow refinement and progress invariants for bound analysis. PLDI 2009 (to appear).

[23] Gupta, A., Henzinger, T. A., Majumdar, R., Rybalchenko, A., and Xu, R.-G. Proving non-termination. In *POPL* (2008), ACM, pp. 147–158.

[24] Halbwachs, N. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[25] Halbwachs, N., Proy, Y., and Roumanoff, P. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design 11*, 2 (1997), 157–185.

[26] Harary, F. *Graph Theory*. Addison-Wesley, 1969.

[27] Ivančić, F., Yang, Z., Ganai, M. K., Gupta, A., Shlyakhter, I., and Ashar, P. F-soft: Software verification platform. In *CAV* (2005), vol. 3576 of *LNCS*.

[28] Jeannet, B., Halbwachs, N., and Raymond, P. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99* (Sep 1999), vol. 1694 of *LNCS*, pp. 39–50.

[29] MathWorks Inc. Electronic vehicle climate control simulink/stateflow model. `www.mathworks.com/products/simulink/demos.html`.

[30] Mathworks Inc. Simulink/stateflow (tm) model-based control design environment. Cf. `www.mathworks.com/simulink`.

[31] Mauborgne, L., and Rival, X. Trace partitioning in abstract interpretation based static analyzers. In *ESOP* (2005), vol. 3444 of *LNCS*, pp. 5–20.

[32] Miné, A. A new numerical abstract domain based on difference-bound matrices. In *PADO II* (May 2001), vol. 2053 of *LNCS*, Springer, pp. 155–172.

[33] Sankaranarayanan, S., Ivančić, F., and Gupta, A. Program analysis using symbolic ranges. In *SAS* (2007), vol. 4634 of *LNCS*, pp. 366–383.

[34] Sankaranarayanan, S., Ivančić, F., Shlyakhter, I., and Gupta, A. Static analysis in disjunctive numerical domains. In *SAS* (2006), vol. 4134 of *LNCS*.

[35] Simon, A., and King, A. Widening polyhedra with landmarks. In *APLAS* (2006), vol. 4279 of *LNCS*.