From Tests To Proofs

Ashutosh Gupta[1], Rupak Majumdar[2], and Andrey Rybalchenko[1]

[1] Max Planck Institute for Software Systems
[2] University of California, Los Angeles

Abstract. We describe the design and implementation of an automatic invariant generator for imperative programs. While automatic invariant generation through constraint solving has been extensively studied from a theoretical viewpoint as a classical means of program verification, in practice existing tools do not scale even to moderately sized programs. This is because the constraints that need to be solved even for small programs are already too difficult for the underlying (non-linear) constraint solving engines. To overcome this obstacle, we propose to strengthen static constraint generation with information obtained from static abstract interpretation and dynamic execution of the program. The strengthening comes in the form of additional linear constraints that trigger a series of simplifications in the solver, and make solving more scalable. We demonstrate the practical applicability of the approach by an experimental evaluation on a collection of challenging benchmark programs and comparisons with related tools based on abstract interpretation and software model checking.

1 Introduction

Programmers make mistakes, and much time and effort is spent on finding and fixing these mistakes. While it has long been known that *program invariants* are the key to proving a program correct with respect to a safety property [10, 17], their applicability has been limited in practice since they often require explicit and expensive programmer annotations. To circumvent this problem, there has been considerable research effort in program analysis for *automatic* inference of program invariants [1, 2, 4, 16, 27]. In these algorithms, a set of constraints is generated from the program text whose solution provides an inductive invariant proof of program correctness.

In the *abstract interpretation* based approach [4, 7, 24] to inductive invariant inference, one computes the fixpoint of the program semantics relative to an abstract domain. In case the abstract domain has infinite height (for example, the domain of polyhedra), termination of the fixpoint computation is enforced by a widening operator. In the *counterexample-guided abstraction refinement (CEGAR)* approach [1, 16], one starts with a set of predicates, and uses spurious counterexamples produced by model checking to dynamically discover new predicates that serve as building blocks for the proof of program correctness. Finally, in the *constraint-based approach* [5, 14, 27], a parametric representation of an invariant map serves a starting point. Then, inductiveness and safety conditions are encoded as constraints on the parameters. Once these constraints have been determined, any satisfying assignment is guaranteed to yield an inductive invariant of the program. For example, an invariant template in linear arithmetic will

File	State-of-the-art techniques				This paper
	INTERPROC	BLAST	INVGEN	INVGEN+Z3	
Seq	×	diverge	23s	1s	**0.5s**
Seq-z3	×	diverge	23s	9s	**0.5s**
Seq-len	×	diverge	T/O	T/O	**2.8s**
nested	×	**1.2s**	T/O	T/O	2.3s
svd(light)	×	50s	T/O	T/O	**14.2s**
heapsort	×	**3.4s**	T/O	T/O	13.3s
mergesort	×	**18s**	T/O	52s	170s
SpamAssassin-loop*	✓	22s	T/O	5s	**0.4s**
apache-get-tag*	×	5s	**0.4s**	10s	0.7s
sendmail-fromqp*	×	diverge	0.3s	5s	**0.3s**

Table 1. Comparison of invariant-based verification tools on benchmark problems.

specify for each program point an expression of the form $\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_n \leq 0$, where $x_1, \ldots, x_n$ are program variables, and $\alpha_0, \ldots, \alpha_n$ are unknown parameters. The control flow graph of the program will specify constraints on the parameters at each program point, such that a global solution for all the α's produces an invariant.

While these techniques hold the potential for extremely sophisticated reasoning about programs, each technique by itself often fails to verify programs, since in practice reasoning about correctness often requires combining the strength of each individual approach. In this paper, we demonstrate the potential of such a combination. We describe the design and implementation of a constraint-based invariant generator for linear arithmetic invariants. In our implementation, we use information from static abstract interpretation-based techniques as well as from dynamic testing to aggressively simplify constraints. Our experimental results demonstrate that using these optimizations our invariant generator can automatically verify many problems for which all the existing approaches we tried are unsuccessful.

It is important to mention that for each of our examples there is (in theory) a polyhedral abstract domain equipped with a suitable widening operator that can successfully prove the desired assertion. Our approach targets the cases for which the *existing* abstract interpreters fail due to heuristic choices made in the implementation that trade off precision for speed. For example, Figure 1(a) shows a program from [13] for which an abstract interpreter implementing the standard convex hull-based widening cannot prove the assertion. In our experiments, the abstract interpretation tool INTERPROC finds the invariants $\mathtt{z} = 10\mathtt{w}$ and $\mathtt{y} \leq 100\mathtt{x}$ at line 2 but not the crucial $\mathtt{y} \geq \mathtt{x}$. We observed that our approach finds the missing fact $\mathtt{y} \geq \mathtt{x}$ which together with the invariants found by INTERPROC, is sufficient to prove the assertion.

Table 1 shows the results of running a collection of state-of-the-art program verification tools on a set of common benchmark programs for software verification, including some challenge programs from [21], which are marked with the star symbol "*". INTERPROC [22] is a tool based on abstract interpretation (we used the PPL library together with the octagon domain when applying INTERPROC). BLAST [16] is a software model checker based on counterexample refinement. INVGEN is our previous implementation of constraint-based invariant generation using constraint logic programming (CLP) as a constraint solver [2]. InvGen+Z3 is the same constraint-based invariant generator but using the Z3 decision procedure [8] as the constraint solver, which applies the Boolean satisfiability-based encoding proposed in [14]. As is evident from Table 1, the results we obtained for the existing tools on the benchmark examples are disappointing. In Column 2, there is a "×" mark for each program for which InterProc was too imprecise to verify the assertion. In Column 3, the counterexample refinement procedure of Blast diverges on several examples. In Columns 4 and 5, the invariant generation procedures time out, denoted by "T/O", on most examples as the constraints become too hard to solve (both for CLP and for SAT). In contrast, our technique is able to efficiently solve all the examples, as shown in the last column.

While our invariant generator can be used in isolation, we have also integrated it with the Blast software model checker and have used it as the counterexample refinement engine using path programs [3]. Invariants for path programs provide additional predicates that refine the abstraction for the software model checker, and can produce better refinement predicates than usually available with current techniques, e.g. [15]. Software model checkers with path program-based counterexample analysis are well-suited for our techniques because they (automatically) generate small program units to either test for bugs or provide invariants. Using this integration, we have applied our implementation to verify a set of software verification benchmark programs [21] recently introduced as a challenge to the community. The examples in the benchmark set are extracted from common security-critical code, and contain assertions related to buffer bounds checking. Our implementation was able to verify all the (correct) programs in the benchmark in about 10s of total time.

Related Work Our work is influenced by recent advances in automatic static inference of inductive invariants using constraint solving [6, 14, 26] as well as by the use of dynamic analysis to estimate and infer likely system properties [9].

Constraint-based invariant synthesis techniques using templates in linear [2, 5, 14] and polynomial [20, 26] arithmetic have been extensively studied, but their application has been limited by the cost of the constraint solving process. As we demonstrate in our experiments, even on quite small examples the constraint solver is likely to time-out. Our static and dynamic constraint simplification techniques limit the search space for the constraint solvers. Our experiments demonstrate orders of magnitude improvements over existing making it feasible to apply these techniques to larger programs.

Software model checking tools, e.g. [1, 16, 19], have previously used invariants from abstract interpretation—most notably alias analysis, but also octagonal constraints [19]—to strengthen the transition relation of the program. The contribution of this work to the research on software model checking is a powerful predicate inference engine using invariant generation. We also perform detailed comparisons of the benefits of combining invariant generation with abstract interpretation, as well as combining invariant generation with CEGAR-based software verification.

Pure dynamic analysis has been used to identify likely, but not necessarily correct, program invariants [9]. The technique uses program tests to evaluate candidate predicates from some a priori fixed database. The predicates that evaluate to true on all test runs are returned as likely invariants. The basic technique is not sound, as the test suite

```c
int x=0; y=0; z=0 w=0;
while(*){
 if(*){
   x++; y+=100;
 }else if(*){
   if (x>=4){ x++; y++; }
 }else if(y>10*w && z>=100*x){
   y=-y;
 }
 w++; z+=10;
}
 if( x>=4 && y <=2) error();
```

(a)

```c
1  int i,j,k,n,m;
2
3  assume(n<=m);
4  for (i=0;i<n;i++)
5   for (j=0;j<n;j++)
6    for (k=j; k<n+m;k++)
7      assert(i+j<=n+k+m);
```

(b)

Fig. 1. (a) Example from [13]. (b) Example `nested.c`.

could be inadequate. Hence in a second step, the inferred invariants are provided to a verification-condition based program verifier. If the verifier succeeds, the combination of the dynamic step and the verification ensures program safety, while removing the need for providing manual invariants. However, there are some shortcomings of this technique. First, since the predicates are chosen from some fixed set (usually for efficiency in evaluation), the required program invariants may not fall into this fixed class. Second, the generated invariants are not in general inductive, therefore if the verifier fails, it is not evident if either a guessed invariant is wrong (that is, more tests should be generated to remove it from the discovered set), or if the guessed invariant does represent all reachable states, but is too weak to allow the verifier to complete the proof.

2 Example

We illustrate our idea using the example program `nested.c` shown in Figure 1(b). We want to construct an invariant that proves the assertion in line 7.

The core idea of our tool is to perform constraint-based invariant synthesis. Our algorithm automatically discovers, through an iterative process, that we need an invariant templates to be a conjunction of four inequalities for each loop head. The invariants for intermediate locations (between loop heads) can be computed from assertions for these locations by propagating strongest postconditions (or weakest preconditions). For clarity of presentation, we shall only show details relevant to the first conjunct in each template. We use the template map η such that

$$\begin{aligned}
\eta.4 &= \alpha + \alpha_{\mathtt{i}}\mathtt{i} + \alpha_{\mathtt{j}}\mathtt{j} + \alpha_{\mathtt{k}}\mathtt{k} + \alpha_{\mathtt{m}}\mathtt{m} + \alpha_{\mathtt{n}}\mathtt{n} \leq 0 \wedge \ldots \wedge \ldots \wedge \ldots\ , \\
\eta.5 &= \beta + \beta_{\mathtt{i}}\mathtt{i} + \beta_{\mathtt{j}}\mathtt{j} + \beta_{\mathtt{k}}\mathtt{k} + \beta_{\mathtt{m}}\mathtt{m} + \beta_{\mathtt{n}}\mathtt{n} \leq 0 \wedge \ldots \wedge \ldots \wedge \ldots\ , \\
\eta.6 &= \gamma + \gamma_{\mathtt{i}}\mathtt{i} + \gamma_{\mathtt{j}}\mathtt{j} + \gamma_{\mathtt{k}}\mathtt{k} + \gamma_{\mathtt{m}}\mathtt{m} + \gamma_{\mathtt{n}}\mathtt{n} \leq 0 \wedge \ldots \wedge \ldots \wedge \ldots\ .
\end{aligned}$$

To obtain an invariant map from these templates, we need to instantiate the set of parameters $\{\alpha, \alpha_{\mathtt{i}}, \alpha_{\mathtt{j}}, \alpha_{\mathtt{k}}, \alpha_{\mathtt{m}}, \alpha_{\mathtt{n}}, \beta, \beta_{\mathtt{i}}, \beta_{\mathtt{j}}, \beta_{\mathtt{k}}, \beta_{\mathtt{m}}, \beta_{\mathtt{n}}, \gamma, \gamma_{\mathtt{i}}, \gamma_{\mathtt{j}}, \gamma_{\mathtt{k}}, \gamma_{\mathtt{m}}, \gamma_{\mathtt{n}}\}$. We proceed by constructing a system of constraints, say Ψ, over the set of template parameters that imposes the invariant conditions on the template map, following a classical approach from the literature [5, 28]. We omit the details for brevity. Unfortunately, even for this small example, we obtain a system of non-linear arithmetic constraints which exceeds the capacity of our constraint solver. Our idea is to scale the invariant generation engine by using information obtained from abstract interpretation as well as from concrete and symbolic runs of the program.

We first observe that for this example, some components of the required invariants can be generated by techniques based on abstract interpretation, e.g., by using octagon and polyhedral domains [7, 24]. By running INTERPROC (using PPL) on this example, we obtain the following invariant map η_α that annotates the loop locations with valid assertions:

$$\eta_\alpha.4 = \ \mathtt{n} \le \mathtt{m} \wedge \mathtt{i} \ge 0\ , \qquad \eta_\alpha.5 = \ \mathtt{n} \ge \mathtt{j} \wedge \mathtt{n} \le \mathtt{m} \wedge \mathtt{i} \ge 0 \wedge \mathtt{j} \ge 0 \wedge \mathtt{n} \ge 1\ ,$$
$$\eta_\alpha.6 = \ \mathtt{n} + \mathtt{m} \ge \mathtt{k} \wedge \mathtt{n} \ge \mathtt{j} + 1 \wedge \mathtt{n} \le \mathtt{m} \wedge \mathtt{k} \ge \mathtt{j} \wedge \mathtt{i} \ge 0 \wedge \mathtt{j} \ge 0.$$

While theoretically the analysis could have found all polyhedral relationships, in practice tools like INTERPROC employ several heuristics that sacrifice precision for speed. In this case, INTERPROC misses the inequality $\mathtt{n} + \mathtt{m} \ge \mathtt{i}$ valid at lines 5 and 6 and crucial for proving the assertion. Our algorithm takes the output generated by the abstract interpreter and uses it as an initial, static strengthening to support constraint based invariant generation.

In the second step, our algorithm collects dynamic information by executing the program. We first present a direct approach that uses program states to compute additional constraints that support invariant generation. Then, we show an extension that can handle unbounded collections of states. The extended method uses symbolic execution to collect such sets of states. We formalize these direct and symbolic approaches in Section 4.

Direct approach Our direct approach starts with a collection of some reachable program states, which can be obtained by applying test generation techniques. We only track states at the head locations of the loops. Suppose we get the following set of states $\{s_1, \dots, s_4\}$ by running the program on test inputs:

$$s_1 = (pc = 4, \mathtt{i} = \mathtt{j} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1), \quad s_2 = (pc = 4, \mathtt{j} = 3, \mathtt{i} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1),$$
$$s_3 = (pc = 5, \mathtt{i} = \mathtt{j} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1), \quad s_4 = (pc = 6, \mathtt{i} = \mathtt{j} = \mathtt{k} = 0, \mathtt{m} = \mathtt{n} = 1).$$

Here, the variable pc represents the control location. We shall use these states to simplify the constraints for invariant generation.

We observe that since template expressions must be true for all reachable program states, in particular, they must hold for the states collected by testing. That is, for each reachable state we can substitute program variables appearing in the template by their values determined by the states and use this information to strengthen the constraint Ψ.

Thus, we can conjoin the following set of linear inequalities to the system of constraints Ψ, which determines the invariant map:

$$\alpha + \alpha_{\mathtt{m}} + \alpha_{\mathtt{n}} \le 0\ , \text{ from } s_1 \qquad \alpha + 3\alpha_{\mathtt{j}} + \alpha_{\mathtt{m}} + \alpha_{\mathtt{n}} \le 0\ , \text{ from } s_2$$
$$\beta + \beta_{\mathtt{m}} + \beta_{\mathtt{n}} \le 0\ , \text{ from } s_3 \qquad \gamma + \gamma_{\mathtt{m}} + \gamma_{\mathtt{n}} \le 0\ , \text{ from } s_4$$

These additional constraints are linear. They can be applied by the solver to trigger a series of simplification steps. After the solving succeeds, we obtain the following invariant map:

$$\eta.4 = \ \mathtt{n} \le \mathtt{m}, \mathtt{i} \ge 0\ , \qquad \eta.5 = \ \mathtt{n} + \mathtt{m} \ge \mathtt{i}, \mathtt{n} \le \mathtt{m}, \mathtt{i} \ge 0\ ,$$
$$\eta.6 = \ \mathtt{n} + \mathtt{m} \ge \mathtt{i}, \mathtt{k} \ge \mathtt{j}, \mathtt{n} \le \mathtt{m}, \mathtt{i} \ge 0\ .$$

Symbolic approach We observe that we can simulate the effect of dynamic simplification using a large/unbounded set of reachable states. For this purpose we use symbolic execution, which computes assertions representing sets of reachable program states. We assume the example discussed so far and three reachable symbolic states below:

$$\varphi_1 = (pc = 4 \wedge \mathtt{i} = 0 \wedge \mathtt{n} \le \mathtt{m})\ ,$$
$$\varphi_2 = (pc = 5 \wedge \mathtt{i} = 0 \wedge \mathtt{j} = 0 \wedge \mathtt{n} \ge 1 \wedge \mathtt{n} \le \mathtt{m})\ ,$$
$$\varphi_3 = (pc = 6 \wedge \mathtt{i} = 0 \wedge \mathtt{j} = 0 \wedge \mathtt{k} = 0 \wedge \mathtt{n} \ge 1 \wedge \mathtt{n} \le \mathtt{m})\ .$$

These symbolic states can be applied to derive additional linear constraints on the template parameters. Due to the reachability of φ_1, φ_2, and φ_3 the implications

$$\varphi_1 \rightarrow \eta.4\ ,\ \ \varphi_2 \rightarrow \eta.5\ ,\ \varphi_3 \rightarrow \eta.6$$

hold for all valuations of program variables. The validity of these implications can be translated into a linear constraint, say Φ, over template parameters. (See Section 4 for details.) We conjoin the constraint Φ with the constraint Ψ that encodes the invariance condition. As a result, the solver performs additional simplifications that lead to improved running time.

Relevant strengthening In fact, after running our algorithm we can discover which inequalities computed using abstract interpretation and added as strengthening to the program were actually useful for finding the invariant that proves the assertion. This information is crucial for keeping minimal the number of facts reported to the software model checker as refinement predicates. For this purpose, we examine the solutions that the constraint solver assigned to the variables encoding the implication validity. For our example, the following inequalities found by INTERPROC were useful: $\mathtt{n} \le \mathtt{m} \wedge \mathtt{i} \ge 0$ at line 4, $\mathtt{n} \le \mathtt{m} \wedge \mathtt{i} \ge 0$ at line 5, and $\mathtt{k} \ge \mathtt{j} \wedge \mathtt{n} \le \mathtt{m} \wedge \mathtt{i} \ge 0$ at line 6.

3 Preliminaries

We start by describing the invariant-based approach for the verification of temporal safety properties and illustrate constraint-based invariant generation.

Programs and invariants We assume an abstract representation of programs by transition systems [23]. A *program* $P = (X, \mathcal{L}, \ell_{\mathcal{I}}, \mathcal{T}, \ell_{\mathcal{E}})$ consists of a set X of variables, a set $\mathcal{L}$ of control locations, an initial location $\ell_{\mathcal{I}} \in \mathcal{L}$, a set $\mathcal{T}$ of transitions, and an error location $\ell_{\mathcal{E}} \in \mathcal{L}$. Each transition $\tau \in \mathcal{T}$ is a tuple (ℓ, ρ, ℓ'), where $\ell, \ell' \in \mathcal{L}$ are control locations, and ρ is a constraint over variables from $X \cup X'$. The variables from X

denote values at control location ℓ, and the variables from X' denote the values of the variables from X at control location ℓ'. The error location $\ell_{\mathcal{E}}$ is used to represent assertion statements. Each failed assertion leads to $\ell_{\mathcal{E}}$. We assume that the error location $\ell_{\mathcal{E}}$ does not have any outgoing transitions. The sets of locations and transitions naturally define a directed graph, called the *control-flow graph* (CFG) of the program, which puts the transition constraints at the edges of the graph.

A *state* of the program P is a valuation of the variables X. The set of all states is denoted by Σ. We shall represent sets and binary relations over states using constraints over X and X' in the standard way. A *computation* of P is a sequence of location and state pairs $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots$ such that ℓ_0 is the initial location and for each consecutive $\langle \ell_i, s_i \rangle$ and $\langle \ell_{i+1}, s_{i+1} \rangle$ there is a transition $(\ell_i, \rho, \ell_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$. A state s is *reachable* at location ℓ if $\langle \ell, s \rangle$ appears in some computation. The program is *safe* if the error location $\ell_{\mathcal{E}}$ does not appear in any computation. A *path* of the program P is a sequence $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), \ldots$ of transitions, where ℓ_0 is the initial location. The path π is *feasible* if there is a computation $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots$ such that each consecutive pair of states (s_i, s_{i+1}) is induced by the corresponding transition, i.e., $(s_i, s_{i+1}) \models \rho_i$. A path that ends at the error location is called an *error path* (or *counterexample path*).

An *invariant* of P at a location $\ell \in \mathcal{L}$ is a super set of states that are reachable at ℓ, which we represent by an assertion over X. An *inductive invariant map* assigns an invariant to each program location such that for each transition $(\ell, \rho, \ell') \in \mathcal{T}$ the implication $\eta.\ell \wedge \rho \rightarrow (\eta.\ell')'$ is valid, where $(\eta.\ell')'$ is the assertion obtained by substituting variables X with the variables X' in $\eta.\ell'$. We observe that due to the invariance condition we have $\eta.\ell_{\mathcal{I}} = \mathit{true}$. An invariant map is *safe* if it assigns an empty set to the error location, i.e., $\eta.\ell_{\mathcal{E}} = \mathit{false}$.

A safe inductive invariant map serves as a proof that the error location cannot be reached on any program execution, and hence that the program is safe. The *invariant-synthesis* problem is to construct such a map for a given program.

Constraint-based Invariant Generation In the *constraint-based approach* [6, 20, 25, 26, 27] to invariant generation, the computation of an invariant map is reduced to a global constraint solving problem over the program locations. The approach consists of three steps. First, a *template* assertion that represents an invariant for each location is fixed in an *a priori* chosen language. A template assertion refers to the program variables X as well as a set of parameters. A parameter valuation determines an invariant. Second, a set of *constraints* over these parameters is defined in such a way that the constraints correspond to the definition of the invariant. This means that every solution to the constraint system yields a safe inductive invariant map. Third, a valuation of parameters is obtained by solving the resulting constraint system.

The language of arithmetic has been widely used to specify invariant templates [20,25,26]. A *linear inequality* over the variables $X = (x_1, \ldots, x_n)$ is an expression of the form $a_0 + a_1 x_1 + \ldots + a_n x_n \leq 0$ if $a_0, \ldots, a_n$ are rational numbers. The language of linear arithmetic consists of conjunctions of linear inequalities. An invariant template in linear arithmetic treats $\alpha_0, \ldots, \alpha_n$ as unknown parameters. For example, the template $\alpha + \alpha_x x + \alpha_y y + \alpha_z z \leq 0$ represents a linear inequality term over the variables x, y, and z. Here, the parameters are α, α_x, α_y, and α_z. A possible template instantiation is $-4 + x + 2y - z \leq 0$.

An invariant template and its expressiveness are determined by the number of conjuncts that appear in the template for each program location. Adding more conjuncts increases the expressive power at the cost of a more expensive constraint solving task. Usually, templates are constructed incrementally, by starting with the weakest template that assigns a single conjunct to each program location and then refining it by adding additional conjuncts if the constraint solving fails to instantiate the template.

Given a template specification for an invariant map, we generate a set of constraints that encode the inductiveness and safety conditions. To encode the inductiveness condition, we generate a constraint $\eta.\ell \wedge \rho \rightarrow (\eta.\ell')'$ for each transition (ℓ, ρ, ℓ'). Note that this implication is implicitly universally quantified over X and X'. Furthermore, the conjunction of such implications for all transitions is existentially quantified over the template parameters. Using Farkas' lemma [28], we eliminate universal quantification. The result is a set of existentially quantified non-linear constraints over the template parameters as well as over the parameters introduced by Farkas' lemma (see [25] for the technical details). Techniques involving Gröbner bases and real quantifier elimination can be used similarly to generate and solve constraints for more general polynomial constraints [20, 26], and for the combined theory of linear arithmetic and uninterpreted functions [2].

We assume a function $\mathsf{InvGenSystem}$ that computes constraints from programs and templates. An application of $\mathsf{InvGenSystem}$ on a program and templates for each program location produces a constraint over the template parameters that encodes the invariant map conditions. For the implementation details see [2, 5].

We illustrate $\mathsf{InvGenSystem}$ using a single transition between location ℓ and ℓ' with the transition relation $x \leq y \wedge x' = x + 1 \wedge y' = y$. We assume a template $\varphi = (\alpha + \alpha_x x + \alpha_y y \leq 0 \wedge \beta + \beta_x x + \beta_y y \leq 0)$ consisting of two conjuncts at the location ℓ, and a singleton conjunction $\psi = (\gamma + \gamma_x x + \gamma_y y \leq 0)$ at the location ℓ'. The starting point is the implication $\varphi \wedge \rho \rightarrow \psi'$. To simplify the exposition, we first eliminate the primed program variables and obtain $\varphi \wedge x \leq y \rightarrow \psi[x + 1/x]$, which we present in the matrix form below.

$$\begin{pmatrix} \alpha_x & \alpha_y \\ \beta_x & \beta_y \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} -\alpha \\ -\beta \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} \gamma_x{+}1 & \gamma_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq -\gamma$$

Now, we apply Farkas' lemma to encode the validity of implication and obtain the following constraint:

$$\exists \lambda \geq 0.\ \lambda \begin{pmatrix} \alpha_x & \alpha_y \\ \beta_x & \beta_y \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} \gamma_x{+}1 & \gamma_y \end{pmatrix} \wedge \lambda \begin{pmatrix} -\alpha \\ -\beta \\ 0 \end{pmatrix} \leq -\gamma$$

This constraint determines the values of template parameters and the additional parameter λ. It contains non-linear terms that result from the multiplication of λ with $(\alpha_x\ \beta_x)$ and $(\alpha_y\ \beta_y)$.

Constraint Solving The constraints generated above are non-linear, since they contain multiplication terms over the parameters from the invariant templates, as well as the additional parameters introduced by Farkas' lemma. The existing solving approaches include symbolic techniques based on instantiations and case splitting, e.g. [5], and using SAT solvers by applying an appropriate propositional encoding, e.g. [14].

For the rest of the paper, we assume a function Solve that takes as input a set of non-linear constraints and returns either a satisfying assignment to the constraints, or that the constraint set is unsatisfiable. Unfortunately, in all but the most basic programs, constraint-based invariant synthesis using the above technique is too expensive. For most realistic programs, the procedure Solve times out.

4 Constraint Simplification

We now describe how we can use additional static and dynamic information to restrict the search space determined by the set of static constraints. Technically, we do this by computing additional constraints on the program transition relation and on the template parameters and conjoining them with the constraint system defining invariant map. Program computations provide a source of such additional dynamic constraints.

INVGEN+ABSINT: Simplification from abstract interpretation Our first simplification uses an abstract interpreter to compute program invariants, and uses the result of the abstract interpretation algorithm to strengthen the program transition relation. That is, suppose that η_α is an invariant map computed by an abstract interpretation algorithm. In our constraint generation, we replace the constraint $\eta.\ell \wedge \rho \rightarrow (\eta.\ell')'$ for a transition (ℓ, ρ, ℓ') with the constraint $\eta.\ell \wedge (\eta_\alpha.\ell \wedge \rho) \rightarrow (\eta.\ell')'$.

INVGEN+TEST: Simplification from tests Individual program computations can be used to simplify the constraints for invariant generation. The crux of the algorithm INVGEN+TEST lies in the observation that an invariant template must hold when partially evaluated on a reachable state of the program.

Let $t(X)$ be a template over the program variables X and s be a reachable program state. We write $t(s/X)$ to denote a template expression that is obtained from t by substituting each variable $x \in X$ with its value $s(x)$ in the state s. Then, the constraint $t[s/X]$ imposes an additional constraint over the template parameters. Note that this constraint is *linear*, i.e., its processing does not require application of expensive non-linear solving techniques.

We show the algorithm INVGEN+TEST in Figure 2. The algorithm takes as input a program P and an invariant template map η with parameters $\mathcal{P}$. It can return an invariant map for P, output that no invariant map exists for the given invariant templates, or find a counterexample to the program safety. There are three conceptual steps of the algorithm. The first step (line 1) constructs a set Ψ of constraints on the invariant template parameters that encode the initiation, inductiveness, and safety conditions. The second step (lines 2–9) runs a set of tests and generates additional constraints on the parameters based on the test executions. Finally, the third step (line 10) solves the conjunction of the static constraints from line 1 and the additional constraints generated during testing.

The loop in lines 3–9 executes the program on a set of tests. We instrument the program so that for each program location ℓ reached in the test, the concrete values of all

input
  P: program; η: invariant template map with parameters $\mathcal{P}$
vars
  Ψ : static constraint; Φ : dynamic constraint
begin
1 Ψ := $\mathsf{InvGenSystem}(P, \eta)$
2 Φ := $true$
3 **repeat**
4    $s_1, \dots, s_n$:= $\mathsf{GenerateAndRunTest}(P)$
5   **if** $s_n(pc) = \ell_{\mathcal{E}}$ **then**
6   **return** "counterexample $s_1, \dots, s_n$"
7   **else**
8      $\Phi := \Phi \wedge \bigwedge_{i=1}^{n} (\eta.s_i(pc))[s_i/X]$
9 **until** no more tests
10 **if** $\mathcal{P}^*$:= $\mathsf{Solve}(\Psi, \Phi)$ succeeds **then**
11   **return** "inductive invariant map $\eta[\mathcal{P}^*/\mathcal{P}]$"
12 **else**
13   **return** "no invariant map for given template"
end.

Fig. 2. Algorithm INVGEN+TEST for invariant generation supported by dynamic simplification using program executions. InvGenSystem creates a constraint over the template parameters that encodes invariant map conditions for the program P, see Section 3. The function GenerateAndRunTest selects program computations.

the program variables that appear in the template $\eta.\ell$ are recorded. If a test hits the error location, then of course, we have found a bug, and we return this error (lines 5,6). Otherwise, the recorded values provide an additional constraint on the template parameters. For example, if the template for a location is $\alpha x + \beta y + \gamma \leq 0$, and a dynamic execution reaches this location with the concrete state $x = 35, y = -9$, we know that the parameters α, β, and γ must satisfy the constraint $35\alpha - 9\beta + \gamma \leq 0$. We call this a *dynamic constraint* on the parameters and add this constraint to the auxiliary constraint Φ.

The testing loop terminates due to an externally supplied coverage criterion. At this point, the constraint solver is invoked to find a satisfying assignment for the parameters in $\mathcal{P}$ that satisfy both the static constraints in Ψ and the dynamic constraints in Φ. If there is no such solution, the algorithm returns that there is no invariant map for the program using the current template map. On the other hand, any satisfying assignment provides an invariant map. Our algorithm maintains the invariant that at any point in lines 3–13, a satisfying assignment to the constraints $\Psi \wedge \Phi$ is guaranteed to be a valid invariant map.

INVGEN+SYMB: Simplification from symbolic execution We observe that the basic algorithm conjoins dynamic, *linear* constraints for each state that is reached by the test generator. A large number of such constraints may overwhelm the constraint solver, despite their low processing cost. We improve the basic algorithm by taking into account *sets* of reachable states using a single strengthening constraint.

```
3      repeat
4.1      π := GeneratePath(P)
4.2      (* π_i = (ℓ_i, ρ_i, ℓ_{i+1}) for 1 ≤ i ≤ n *)
5        if ℓ_{n+1} = ℓ_E and π is feasible then
6          return "counterexample π"
7        else
8.1        φ := (∃X. ρ_1 ∘ ··· ∘ ρ_n)[X/X']
8.2        Φ := Φ ∧ Encode(φ → η.ℓ_{n+1})
9      until no more paths
```

Fig. 3. Algorithm INVGEN+SYMB. It can be obtained by replacing lines 3–9 of the algorithm INVGEN+TEST with the above statements. The function GeneratePath selects program paths. Encode creates *linear* constraints over template parameters that encode the validity of the given implication.

We assume a template $t(X)$ and a set of reachable states represented by an assertion $\varphi(X)$. We can obtain such sets of states by performing symbolic execution along a collection of program paths. Then, the implication $\varphi(X) \rightarrow t(X)$ must hold for all valuations of X since every state in φ is reachable.

Following the method in Section 3, we encode the validity of the implication by a constraint over the template parameters. In this case, the encoding yields *linear* constraints. In contrast to the cases when the left-hand side of the implication contains template assertions, in the above implication program variables have *constant* coefficients. Thus, when multiplying additional parameters (appearing due to the application of Farkas' lemma) with coefficients attached to the program variables we obtain linear terms, which, in turn, result in linear constraints.

For example, we consider a template $t(x, y, z)$ that consists of two conjuncts $\alpha + \alpha_x x + \alpha_y y + \alpha_z z \leq 0 \wedge \beta + \beta_x x + \beta_y y + \beta_z z \leq 0$. We assume a set of states $\varphi = (-x \leq 0 \wedge -y \leq 0 \wedge x + y - z \leq 0)$ reached by symbolic execution. The encoding of the implication $\varphi \rightarrow t$ yields the constraint

$$\exists \Lambda \geq 0.\ \Lambda \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & -1 \end{pmatrix} = \begin{pmatrix} \alpha_x & \alpha_y & \alpha_z \\ \beta_x & \beta_y & \beta_z \end{pmatrix} \wedge \Lambda \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \leq \begin{pmatrix} -\alpha \\ -\beta \end{pmatrix} ,$$

which is clearly linear.

We assume a function Encode that translates an implication between an assertion representing a set of states and a template into a linear constraint over template parameters. Our extended algorithm INVGEN+SYMB applies Encode on sets of reachable states computed by symbolic execution of the program. The algorithm is presented in Figure 3. Since it extends the basic algorithm INVGEN+TEST by adding the symbolic treatment of reachable states, we only present the modified part.

The algorithm INVGEN+SYMB interleaves symbolic execution and collection of constraints. It relies on an external function GeneratePath that selects paths through the control-flow graph of the program, see line 4.1. For a given path, we compute an assertion representing states that are reachable by executing its transitions, see line 8.1. We use the relational composition operator $\circ$, which is defined by $\rho \circ \rho' =$

File	InterProc	InvGen	InvGen+Z3	InvGen + InterProc	InvGen+Z3 + InterProc + Symb	InvGen + InterProc + Symb
Seq	×	23.0s	1s	**0.5s**	6s	**0.5s**
Seq-z3	×	23.0s	9s	**0.5s**	6s	**0.5s**
Seq-len	×	T/O	T/O	T/O	4s	**2.8s**
nested	×	T/O	T/O	17.0s	3s	**2.3s**
svd(light)	×	T/O	T/O	**10.6s**	T/O	14.2s
heapsort	×	T/O	T/O	19.2s	48s	**13.3s**
mergesort	×	T/O	**52s**	142s	T/O	170s
SpamAssassin-loop	✓	T/O	5s	**0.28s**	1s	0.4s
apache-get-tag	×	**0.4s**	10s	0.6s	3s	0.7s
sendmail-fromqp	×	0.3s	5s	0.3s	5s	**0.3s**
Example1(b)	×	T/O	T/O	0.4s	1s	0.35s

Table 2. Comparison of variations of invariant verification techniques and InterProc on additional benchmark problems inspired by [21]. "✓" and "×" indicate whether the invariant computed by InterProc proves the assertions, and "T/O" stands for time out.

$\exists X''.\ \rho[X''/X'] \wedge \rho'[X''/X]$, to compute the transition relation of the whole path. The existential quantification in line 8.1 projects this relation to the successor states φ, i.e., it computes the range of the relation. We use variable renaming to keep the resulting assertion consistent with the templates over program variables. We conjoin the constraint resulting from the translation of the implication between the reachable states φ and the corresponding template $\eta.\ell_{n+1}$ to the dynamic constraint Φ before proceeding with the next path. We assume an external procedure that selects a finite set of paths. In our implementation, we apply directed symbolic execution that attempts to unroll loops at least one time.

The following theorem states that our optimizations are sound (and relatively complete).

Theorem 1. *[Correctness] If Algorithm* InvGen+AbsInt, InvGen+Test, *or* InvGen+Symb *on input program P and invariant template map η returns (a) "counterexample $s_1, \ldots, s_n$," then there is an execution of the program that reaches the error location; (b) "inductive invariant map η^*," then η^* is an invariant map for P, and the program P is safe; (c) "no invariants with template η," then there is no invariant map for P with the given invariant template map η.*

5 Experiences

Implementation We implemented the algorithms InvGen+Test and InvGen+Symb using SICStus Prolog [29], the linear arithmetic solver clp(q,r) [18] and the Z3 solver [8] as the backend to solve non-linear constraints. When describing the application of InvGen together with Z3, we shall write InvGen+Z3. We apply the InterProc [22] tool for abstract interpretation over numeric domains, and use the PPL backend for polyhedra, mainly due to its source code availability. In principle, a variety of other tools could be used instead, e.g., the ASPIC tool implementing the lookahead widening and acceleration techniques [11, 12]. INVGEN provides a frontend for C programs, which relies on CIL infrastructure for C program analysis and transformation and abstracts from non-arithmetic operations appearing in the input program. We implement the following additional variable elimination optimization. The additional constraints obtained from dynamic and static strengthening are linear. In particular, the additional variables that encode implication between symbolic states and templates, Λ in the previous section, can be eliminated. We perform this simplification step before applying the (expensive) techniques for solving non-linear constraints. For our constraint logic programming-based implementation, this results in a reduction of the number of calls to the linear arithmetic solver. When using the SAT approach, it allows us to avoid applying the propositional search to constraints that can be solved symbolically.

In our experimental evaluation, we observed that INVGEN+TEST and INVGEN+SYMB offer similar efficiency improvement, with a few exceptions when INVGEN+SYMB was significantly better. To keep the tables with experimental data compact, we only describe evaluation of the strengthening that uses symbolic execution INVGEN+SYMB.

Software Verification Challenge Benchmarks We applied INVGEN on a suite of software verification challenge programs described in [21]. The examples in this benchmark are extracted from large applications by mining a security vulnerability database for buffer overflow problems. We use the corrected versions of these programs, using the buffer access checks as assertions. The suite consists of 12 programs.[1] Using polyhedral abstract domain, INTERPROC computes invariants that are strong enough to prove the assertion for half of them. The constraint based invariant generation together with the SAT-based encoding, i.e., INVGEN+Z3, generates invariants for all programs within 36.5 seconds of total time. Using the CLP backend, INVGEN handles 11 examples within 6.3 seconds, and times out on one program, which is handled by INVGEN+Z3 in 5 seconds. Using the static and dynamic strengthening described in this paper, we obtain the following running times. The combination INVGEN+Z3+INTERPROC+SYMB solves all examples in 29.5 seconds, while INVGEN+INTERPROC+SYMB handles all examples within 9.6 seconds. These experiments demonstrate that the various optimizations can have an effect on verification, but the running times were too short to draw meaningful conclusions.

Impact of Dynamic Strengthening The collection from [21] did not allow us to perform a detailed benchmarking of our algorithm, since the running times on these examples were too short. We obtained a set of more difficult benchmarks inspired by [21] by adding additional loops and branching statements, and provide a detailed comparison that describes the impact of static and dynamic strengthening in isolation in Table 2. INTERPROC computes 50 inequalities for each loop head, which results in a significant increase in the number of variables in the constraint system. While being an obstacle for

[1] Due to short running times, we present the aggregated data and do not provide any table containing entries for individual programs.

the propositional search procedure in Z3, the increased number of variables does not significantly affect the CLP-based backend since the additional variables appear in linear terms. In summary, the performance of INVGEN+Z3 decreases and the performance of INVGEN goes up by adding facts from INTERPROC.

Integration with BLAST We have modified the abstraction refinement procedure of the BLAST software model checker [15] by adding predicate discovery using path invariants [3]. Table 3 shows how constraint based invariant generation can be effective for refining abstractions. The number of counterexample refinement iterations required is reduced in all examples. For several examples we achieved termination of previously diverging abstraction refinement, and for others the reduction ranges between 25 and 400 percent.

File	BLAST	BLAST + INVGEN + INTERPROC + SYMB
Seq	diverge	8
Seq-len	diverge	9
fregtest	diverge	3
sendmail-fromqp	diverge	10
svd(light)	144	43
Spamassassin-loop	51	24
apache-escape	26	20
apache-get-tag	23	15
sendmail-close-angle	19	15
sendmail-7to8	16	13

Table 3. INVGEN + INTERPROC + SYMB for predicate discovery in BLAST. We show the number of refinement steps required to prove the property.

Summary Our experimental evaluation leads to the following observations:

– For complex constraint solving problems, the additional strengthening facilitates significant improvement. It ranges from reducing the running time by two orders of magnitude to making timing out examples solvable within seconds.
– If the constraint solving is already fast in the purely static case, then the strengthening does not cause any significant running time penalty.

Acknowledgments. The second author was sponsored in part by the NSF grants CCF-0546170 and CNS-0720881. The third author was supported in part by Microsoft Research through the European Fellowship Programme.

References

1. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
2. D. Beyer, T. Henzinger, R. Majumdar, and A.Rybalchenko. Invariant synthesis for combined theories. In *Proc. VMCAI*, LNCS 4349, pages 378–394. Springer, 2007.
3. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Proc. PLDI*, pages 300–309. ACM Press, 2007.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. PLDI*, pages 196–207. ACM, 2003.

5. M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV*, LNCS 2725, pages 420–432. Springer, 2003.
6. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Proc. VMCAI*, pages 1–24. Springer, 2005.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78*, pages 84–96. ACM Press, 1978.
8. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, LNCS 4963, pages 337–340. Springer, 2008.
9. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):1–25, 2001.
10. R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. AMS, 1967.
11. L. Gonnord and N. Halbwachs. Combining widening and acceleration in Linear Relation Analysis. In *Proc. SAS*, LNCS 4134, pages 144–160. Springer, 2006.
12. D. Gopan and T. Reps. Lookahead widening. In *Proc. CAV*, LNCS 4144, pages 452–466. Springer, 2006.
13. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *Proc. TACAS*, LNCS 4963, pages 443–458. Springer, 2008.
14. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292. ACM, 2008.
15. T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.
16. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
17. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.
18. C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.
19. H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV*, LNCS 4144, pages 137–151. Springer, 2006.
20. D. Kapur. Automatically generating loop invariants using quantifier elimination. Technical Report 05431 (*Deduction and Applications*), IBFI Schloss Dagstuhl, 2006.
21. K. Ku, T. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *Proc. ASE*, 2007.
22. G. Lalire, M. Argoud, and B. Jeannet. The interproc analyzer. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`.
23. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
24. A. Miné. The octagon abstract domain. *Higher-Order and Symb. Comp.*, 19:31–100, 2006.
25. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *Proc. SAS*, LNCS 3148, pages 53–68. Springer, 2004.
26. S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. POPL*, pages 318–329. ACM, 2004.
27. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. VMCAI*, LNCS 3385, pages 25–41. Springer, 2005.
28. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.
29. The Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 2001. Release 3.8.7.