InvGen: An Efficient Invariant Generator

Ashutosh Gupta and Andrey Rybalchenko

Max Planck Institute for Software Systems (MPI-SWS)

Abstract. In this paper we present InvGen, an automatic linear arithmetic invariant generator for imperative programs. InvGen's unique feature is in its use of dynamic analysis to make invariant generation order of magnitude more efficient.

1 Introduction

Program verification relies on invariants for reasoning about sets of reachable states [3]. Synthesizing invariants that satisfy a given assertion is a difficult task. The scalability of existing approaches to invariant generation is severely limited due to the high computation cost of the underlying symbolic reasoning techniques.

We present InvGen, an automatic tool for the generation of linear arithmetic invariants for imperative programs. InvGen uses a constraint-based approach to generate invariants [2]. InvGen combines it with static and dynamic analysis techniques to solve constraints efficiently [4]. InvGen provides an order of magnitude efficiency improvement wrt. the existing tools, see [5]. This improvement enables InvGen's application for automatic software verification, e.g., for refinement of predicate abstraction by generating invariants for program fragments determined by spurious counterexamples [1].

In this paper, we describe the design and implementation of InvGen and present a collection of optimizations that were necessary to achieve the desired scalability. We also describe our experience with applying InvGen on challenge benchmarks [8] and micro-benchmarks, which containing code fragments that are difficult to analyze automatically. The experimental results indicate that InvGen is a practical tool for software verification.

2 Usage

InvGen takes as input a program over linear arithmetic expressions that is given in C syntax or as a transition relation.[1] InvGen uses a template based technique to compute invariants, which requires that a set of templates is given as input to the tool. A template is a Boolean combination of linear inequalities over program variables with coefficients that are kept as parameters. As a result, InvGen either returns an invariant that proves the non-reachability of the error location or fails.

[1] See [5] for the syntax of transition relations.

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 634–640, 2009.

InvGen offers a collection of heuristics to improve scalability of invariant generation, and allows one to explore various combination of these heuristics. Support of dynamic/static analysis can be disabled/enabled with some parameters. Dynamic analysis can be performed using concrete or symbolic execution. Other tools can be used to generate test executions for InvGen. InvGen can run in the server mode, in which it communicates its input/output via standard input and output streams. The full set of options is described online [5].

3 Tool

In this section we briefly present the algorithm used by InvGen and focus on the tool design and implementation.

3.1 Algorithm

InvGen computes linear arithmetic invariants that prove the non-reachability of the error location. We will call such invariants *safe*. InvGen applies the template based approach [2] for the invariant generation, which assumes an *invariant template* at each cut-point location in the program, i.e., at loop entry locations. Each invariant template consists of parameterized linear inequalities over the program variables. The specific goal of invariant generation is to find an instantiation of the template parameters that yields a safe invariant. The invariant templates at the start and error locations of the program are `true` and `false`, respectively. A template instantiation yields an invariant if each program path `stmt1;...;stmtN;` between each pair ℓ and ℓ' of adjacent cut-points satisfies the *inductiveness condition*: the instantiated invariant templates $\eta(\ell)$ and $\eta(\ell')$ at the respective cut-points together with the program path form a valid Hoare triple $\{\eta(\ell)\}$`stmt1;...;stmtN;`$\{\eta(\ell')\}$.[2] We translate the inductiveness condition into an arithmetic constraint over the template parameters. Each solution of this constraint yields a safe invariant.

```
1: x=0;
2: assume(n>0);
3: while(x<n){
4:   x++;
5: }
6: assert(x==n);
```

Fig. 1. Example `simple.c`

```
1: x=0;
2: while(x<n){
3:   x++;
4: }
5: if(n>0)
6:   assert(x==n);
```

(a)

```
1: assume(p);
2: while(p){
3:   ...
4:   assume(q);
5: }
6: assert(q);
```

(b)

Fig. 2. Examples requiring disjunctive invariants

[2] $\{P\}$`stmts`$\{Q\}$ is a valid Hoare triple if each computation that starts in a state satisfying the assertion Q either terminates in a state satisfying Q or diverges [7]. In our case, program paths are loop free and hence terminating.

See Figure 1. The example program `simple.c` contains a statement `assert(p)`, which is a short form for `if (!p) ERROR;`. The entry location of the `while` loop is a cut-point location ℓ_3. At this location we assume an invariant template $\alpha_{\mathtt{x}}\mathtt{x}+\alpha_{\mathtt{n}}\mathtt{n} \leq \alpha$, where $\alpha_{\mathtt{x}}, \alpha_{\mathtt{n}}, \alpha$ are unknown parameters. The inductiveness condition for the path from the start location to the loop entry requires the validity of the triple $\{\mathtt{true}\}\mathtt{x} = 0; \mathtt{assume}(\mathtt{n} > 0); \{\eta(\ell_3)\}$, the condition for the loop iteration is $\{\eta(\ell_3)\}\mathtt{assume}(\mathtt{x} < \mathtt{n}); \mathtt{x} = \mathtt{x} + 1; \{\eta(\ell_3)\}$, and the path from the loop entry to the error location produces $\{\eta(\ell_3)\}\mathtt{assume}(\mathtt{x} \geq \mathtt{n}); \mathtt{assume}(\mathtt{x} \neq \mathtt{n}); \{\mathtt{false}\}$. The resulting arithmetic constraint has a following solution for template parameters: $\alpha_{\mathtt{x}} = 1$, $\alpha_{\mathtt{n}} = -1$, and $\alpha = 0$. Hence, $\mathtt{x} - \mathtt{n} \leq 0$ is a safe invariant.

Arithmetic constraints that encode the inductiveness condition contain non-linear arithmetic expressions and are difficult to solve. InvGen improves performance of the constraint solving by adding information collected using dynamic and static analysis techniques.

In the dynamic analysis phase, we execute the program according to a coverage criterion and collect the reached states. By definition, each reached state must satisfy program invariants. For each reached state, we substitute the program variables occurring in the invariant template at the corresponding control location by their values in the state. Thus, we obtain linear constraints over template parameters. For `simple.c`, assume a reached state at the loop entry such that $(\mathtt{x} = 0, \mathtt{n} = 1)$. After substituting program variable by their values in the above template for the location ℓ_3 we obtain the constraint over template parameters $\alpha_{\mathtt{x}}0 + \alpha_{\mathtt{n}}1 \leq \alpha$. Such linear constraints improve the efficiency of constraint solving [4].

In the static analysis phase, we first apply abstract interpretation in a eager fashion to compute a collection of invariants – which are not necessarily strong enough to prove the non-reachability of the error location – that holds for the program, and then annotate the program with these invariants. The invariants obtain using abstract interpretation allow InvGen to focus on the synthesis missing, complex invariants in a goal directed way. Such invariants may be missed by the abstract interpreter due to the necessary precision loss of join and widening operators and the limited precision of efficient abstract domains.

3.2 Design

We present the design of InvGen in Figure 3. The input program is passed to the dynamic and static analyzers. The results of each analysis together with the program and the templates are passed to the constraint generator. The generated constraints are solved by a constraint solver. If the solver succeeds then InvGen returns a safe invariants.

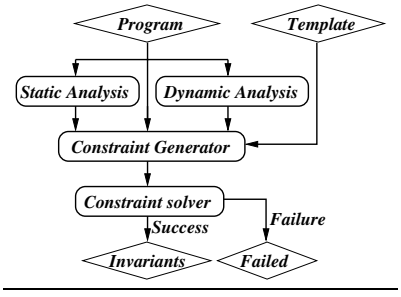

Fig. 3. InvGen design

3.3 Implementation

Figure 4 outlines the implementation of InvGen. It is divided into two executables, *frontend* and InvGen.

The frontend executable contains a CIL [10] based interface to C and an abstract interpreter InterProc [9]. The frontend takes a program procedure written in C language as an input, and applies InterProc on the program three times using the interval, octagon, and polyhedral abstract domains. Then, the frontend outputs the transition relation of the program that is a annotated with the results computed by InterProc. Se [5] for the output format.

Next, we describe the components of InvGen, following Figure 4.

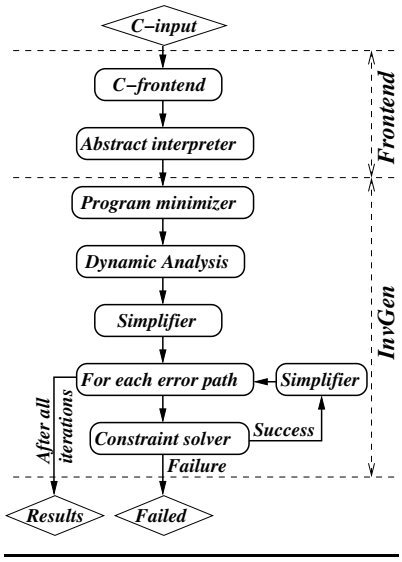

Fig. 4. InvGen implementation

Program minimizer. InvGen minimizes the transition relation of the program to reduce the complexity of constraint solving. InvGen computes a minimal set of cut-point locations, and replaces each cut-point free path by a single, compound program transition. The unsatisfiable and redundant transitions are eliminated. At this phase, the invariants obtained from InterProc can lead to the elimination of additional transitions.

Dynamic analysis. InvGen collects dynamic information for the minimized program using either concrete and symbolic execution. In case of concrete execution, InvGen collects a finite set of reachable states by using a guided testing technique. Otherwise, InvGen performs a bounded, exhaustive symbolic execution of the program. By default, the bound is set to the number of cut-points in the program. The user can limit the maximum number of visits for each cut-point during the symbolic execution.

Simplifier. InvGen simplifies all arithmetic constraints locally at each step of the algorithm.

Consider the phase when the dynamic analysis using symbolic execution produces additional linear constraints over template parameters. These constraints contain existentially quantified variables, and the scope of the quantifier is rather limited. For `simple.c`, assume that a symbolic state $\mathtt{x} = 0 \wedge \mathtt{n} \geq 0$ is reached at the loop entry location. For any template evaluation, the symbolic state is subsumed by the invariant, hence $(\mathtt{x} = 0 \wedge \mathtt{n} \geq 0) \rightarrow (\alpha_{\mathtt{x}}\mathtt{x} + \alpha_{\mathtt{n}}\mathtt{n} \leq \alpha)$. After the elimination of program variables InvGen obtains the constraints $\lambda \geq 0 \wedge -\lambda = \alpha_n \wedge \alpha \geq 0$, where λ is an existentially quantified variable.

λ does not appear anywhere in the constraint and can be eliminated locally. As a result, we obtain $\alpha_n \leq 0 \wedge \alpha \geq 0$.

InvGen also simplifies constraints obtained by the concrete execution and abstract interpretation.

Constraint solver. The inductiveness conditions result in non-linear arithmetic constraints. For `simple.c`, the inductiveness condition for the error path $\{\eta\}\texttt{assume}(\texttt{x} > \texttt{n}); \{\texttt{false}\}$ translates to the implication $\alpha_{\texttt{x}}\texttt{x} + \alpha_{\texttt{n}}\texttt{n} \leq \alpha \wedge \texttt{x} > \texttt{n} \rightarrow 1 \leq 0$ with universal quantification over the program variables. After the elimination of the program variables, InvGen obtains the non-linear constraint $\lambda \geq 0 \wedge \delta \geq 0 \wedge \lambda\alpha_x = \delta \wedge \lambda\alpha_n + \delta = 0 \wedge \lambda\alpha + 1 \leq \delta$, where λ and δ are existentially quantified, non-negative variables that encode the implication validity. In practice, such existentially quantified variables range over a small domain, typically they are either 0 or 1.

InvGen leverages this observation in order to solve the constraints by performing a case analysis on the variables with small domain. Each instance of case analysis results in a linear constraint over template parameters, which can be solved using a linear constraint solver. This approach is incomplete, since InvGen does not take all possible values during the case analysis, however it is effective in practice.

Multiple paths to error location. A program may have multiple cut-point free paths that lead to the error location, which we refer to as error paths. InvGen deals with multiple error paths in an incremental fashion for efficiency reasons. Instead taking inductiveness conditions for all error paths into account, InvGen computes a safe invariant for one error path at a time. Already computed invariants are used as strengthening when dealing with the remaining error paths.

4 Discussion

We highlight several important aspects of InvGen in detail.

Abstract interpretation. The template instantiation computed by InvGen may rely on the invariants discovered during the abstract interpretation phase. InvGen keeps track of such invariants and reports them to the user.

Template generation. The applicability of InvGen depends on the choice of invariant templates. If the template is too expressive, e.g., if it admits a number of conjuncts that is larger than required, then the efficiency of InvGen is decreasing due to the increased difficulty of constraint solving. A template that is not expressive enough cannot yield a desired invariant.

In our experience, the majority of invariants require a template that is a conjunction of two linear inequalities. This surprisingly small number of conjuncts is due to the strengthening using invariants obtained during the abstraction interpretation phase—InvGen focuses on the missing invariants. By default, for each cut-point location InvGen assumes a conjunction of two parametric linear inequalities as a template.

Table 1. Performance of InvGen on benchmark inspired by [8]. InvGen* does not apply dynamic analysis. InvGen** applies neither static nor dynamic analysis. "T/O" means time out after 10 minutes.

File	InvGen**	InvGen*	InvGen
Seq	23.0s	**0.5s**	**0.5s**
Seq-z3	23.0s	**0.5s**	**0.5s**
Seq-len	T/O	T/O	**2.8s**
nested	T/O	17.0s	**2.3s**
svd(light)	T/O	**10.6s**	14.2s
heapsort	T/O	19.2s	**13.3s**
mergesort	T/O	**142s**	170s
Spam-loop	T/O	**0.28s**	0.4s
apache-tag	**0.4s**	0.6s	0.7s
sendmail-qp	0.3s	0.3s	**0.3s**

Table 2. Application of InvGen for the predicate discovery in Blast using path invariants. We show the number of refinement steps required to prove the property.

File	Blast	Blast + InvGen
Seq	diverge	8
Seq-len	diverge	9
fregtest	diverge	3
sendmail-qp	diverge	10
svd(light)	144	43
Spam-loop	51	24
apache-escape	26	20
apache-tag	23	15
sendmail-angle	19	15
sendmail-7to8	16	13

Disjunction. The user can provide disjunctive templates to InvGen, which yields disjunctive invariants. Currently, InvGen is not practical for disjunctive invariants.

Nevertheless, programs that require disjunctive invariants appear often in practice, as illustrated in Figure 2. Example 2(a) relies on a disjunctive loop invariant $n \geq x \vee n < 0$ to prove the assertion. Similarly, Example 2(b) requires a disjunctive loop invariant $p \vee q$.

Avoiding such program patterns improves the effectiveness of InvGen. For example, adding a commonly used assumption $n \geq 0$ at the start location of the program in Figure 2(a) eliminates the need for disjunctive invariants.

Concrete vs. symbolic dynamic analysis. In our evaluation, dynamic analysis using symbolic execution performs better than with the concrete execution. Symbolic execution produces a smaller set of constraints and leads to the improvement of the constraint solving efficiency that is never worse the one achieved by the concrete execution.

5 Experiences

Verification benchmarks. We applied InvGen on a collection of programs that are difficult for state-of-the-art software verification tools [8]. The collection consists of 12 programs. Due to short running times, we present the aggregated data and do not provide information for each program individually. Using the polyhedral abstract domain, InterProc computes invariants that are strong enough to prove the assertion for six programs in the collection. InvGen handles 11 examples in 6.3 seconds, and times out on one program.

Effect of static and dynamic analysis. The collection [8] does not allow us to perform a thorough experimental evaluation of InvGen, since the running times on these examples are too short. We constructed a set of more interesting programs that is inspired by [8] by extending the programs from this collection with additional loops and branching statements. Table 1 shows the effect of static and dynamic analysis when dealing with the constructed examples.

Random input. We developed a tool for the generation of random program. We try these random programs with InvGen without testing then with testing feature. We generated 17 programs with three cut-point and one error location in each of them, see [5]. We observed that for these programs, dynamic analysis either improves or at least does not decrease the efficiency of invariant generation. In four cases, the efficiency of InvGen increased by two orders of magnitude.

Integration with Blast. We have modified the abstraction refinement procedure of the Blast software model checker [6] by adding predicate discovery using path invariants [1]. Table 2 indicates that constraint based invariant generation can be an effective tool for refining predicate abstraction. For several examples, Blast diverged due to the disability to find the right set of predicates. In these cases, InvGen enabled the successful termination of the verification attempt, while for other cases it reduced the number of the number of refinement iterations by 25–400%.

References

1. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Proc. PLDI, pp. 300–309. ACM Press, New York (2007)
2. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
3. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science. AMS (1967)
4. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: Proc. TACAS. Springer, Heidelberg (2009)
5. Gupta, A., Rybalchenko, A.: InvGen: an efficient invariant generator, `http://www.mpi-sws.org/~agupta/invgen/`
6. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. In: POPL 2004: Principles of Programming Languages, pp. 232–244. ACM, New York (2004)
7. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of ACM (1969)
8. Ku, K., Hart, T., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: Proc. ASE (2007)
9. Lalire, G., Argoud, M., Jeannet, B.: The interproc analyze, `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`
10. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)