Effective use of SMT solvers for Program Equivalence Checking through Invariant Sketching and query-decomposition

Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal

Indian Institute of Technology Delhi

Abstract. Program equivalence checking is a fundamental problem in computer science with applications to translation validation and automatic synthesis of compiler optimizations. Contemporary equivalence checkers employ SMT solvers to discharge proof obligations generated by their equivalence checking algorithm. Equivalence checkers also involve algorithms to infer invariants that relate the intermediate states of the two programs being compared for equivalence. We present a new algorithm, called *invariant-sketching*, that allows the inference of the required invariants through the generation of counter-examples using SMT solvers. We also present an algorithm, called *query-decomposition*, that allows a more capable use of SMT solvers for application to equivalence checking. Both invariant-sketching and query-decomposition help us prove equivalence across program transformations that could not be handled by previous equivalence checking algorithms.

1 Introduction

The general problem of program equivalence checking is undecidable. Several previous works have tackled the problem for applications in (a) translation validation, where the equivalence checker attempts to automatically generate a proof of equivalence across the transformations (translations) performed by a compiler [1,2]; and (b) program synthesis, where the equivalence checker is tasked with determining if the optimized program proposed by the synthesis algorithm is equivalent to the original program specification [3,4]. For both these applications, soundness is critical, i.e., if the equivalence checker determines the programs to be equivalent, then the programs are guaranteed to have equivalent runtime behaviour. On the other hand, completeness may not always be achievable (as the general problem is undecidable), i.e., it is possible that the equivalence checker is unable to prove the programs equivalent, even if they are actually equivalent. For example, recent work on black-box equivalence checking [5] involves comparing the unoptimized (O0) and optimized (O2/O3) implementations of the same C programs in x86 assembly. While their algorithm guarantees soundness, it does not guarantee completeness; their work reported that they could prove equivalence only across 72-76% of the functions in real-world programs, across transformations produced by modern compilers like GCC, LLVM, ICC,

```
C0 int g[144];
C1 int example0() {
C2   int sum = 0;
C3   for (int i = 0; i < 144; i++)
C4   {
C5     sum = sum + g[i];
C6   }
C7   retval = sum/144; //return
C8 }
```

```
A0 example0_compiled:
A1   r1 := 0;     //sum'
A2   r2 := 144;   //loop index(i')
A3   r3 := 0;     //array index(a')
A4   loop:
A5     r1 := r1 + [ base_g + 4*r3];
A6     r2 := r2 - 1;
A7     r3 := r3 + 1;
A8   if (r2 != 0) goto loop
A9   rax := mul-shift-add(r1,144)
```

Fig. 1: C-function `example0()` and abstracted version of its compiled assembly (as produced by `gcc -O2`). We use a special keyword `retval` to indicate the location that holds the return value of the function. In assembly, sum and i variables are register allocated to `r1` and `r2` respectively and r3 is an iterator for indexing the array g. Division operation in C program is optimized to mul-shift-add instructions in assembly. base_g represents the base address of array g in memory. `[x]` is short-hand for 4 bytes in memory at address x.

and CompCert. Our work aims to reason about equivalence in scenarios where these previous algorithms would fail.

To understand the problem of equivalence checking and the general solution, we discuss the proof of equivalence across the example pair of programs in Figure 1. The most common approach to proving that this pair of programs is equivalent involves the construction of a *simulation relation* between them. If $Prog_A$ represents the C language specification and $Prog_B$ represents the optimized x86 implementation, a simulation relation is represented as a table, where each row is a tuple $((L_A, L_B), P)$ such that L_A and L_B are program locations in $Prog_A$ and $Prog_B$ respectively, and P is a set of invariants on the live program variables[1] at locations L_A and L_B. Program locations represent the next instruction (PC values) to be executed in a program and the live program variables are identified by performing liveness analysis at every program location. A tuple $((L_A, L_B), P)$ represents that the invariants P hold whenever the two programs are at L_A and L_B respectively. A simulation relation is valid if the invariants at each location pair are inductively provable from the invariants at all its predecessor location pairs. Invariants at the entry location (the pair of entry locations of the two programs) represent the equivalence of program inputs ($Init$) and form the base case of this inductive proof. If we can thus inductively prove equivalence of return values at the exit location (the pair of exits of the two programs), we establish the equivalence of the two programs. For C functions, the return values include return registers (e.g., `rax` and `rdx`) and the state of the heap and global variables. Formally, a simulation relation is valid if:

$$Init \Leftrightarrow invariants_{(Entry_A, Entry_B)} \tag{1}$$

$$\forall_{(L'_A, L'_B) \to (L_A, L_B)} \; invariants_{(L'_A, L'_B)} \Rightarrow_{(L'_A, L'_B) \to (L_A, L_B)} invariants_{(L_A, L_B)} \tag{2}$$

[1] For assembly code, variables represent registers, stack and memory regions.

Location	Invariants (P)
(C0, A0)	`g` $\mapsto$ `base_g`
(C3, A4)	`144-i = i' , sum = sum', i = a', g` $\mapsto$ `base_g`
(C7, A9)	`(retval`$_C$ `= rax`$_A$`), g` $\mapsto$ `base_g`

Init: $g \mapsto base\_g, M_A =_\Delta M_B$

Table 1: Simplified simulation relation for the programs in Fig. 1. M_A and M_B are memory states in $Prog_A$ and $Prog_B$ respectively. $=_\Delta$ represents equivalent arrays except for Δ, where Δ represents the stack region.

Here, $invariants_{(L_A,L_B)}$ is the same as P and represents the conjunction of invariants in the simulation relation for the location pair (L_A, L_B), $L_A^{'}$ and $L_B^{'}$ are predecessors of L_A and L_B in programs $Prog_A$ and $Prog_B$ respectively, and $\Rightarrow_{(L_A^{'},L_B^{'})\rightarrow(L_A,L_B)}$ represents implication over the paths $L_A^{'} \rightarrow L_A$ and $L_B^{'} \rightarrow L_B$ in programs $Prog_A$ and $Prog_B$ respectively.

Almost all compiler optimizations are *similarity* preserving, i.e. the optimized program simulates the original program, and hence approaches that rely on the construction of a simulation relation usually suffice for computing equivalence across compiler optimizations. There have been proposals in previous work [6] to handle transformations that do not preserve similarity (but preserve equivalence), but we do not consider them in this paper. In our experience, modern compilers rarely (if ever) produce transformations that do not preserve similarity. Table 1 shows a simulation relation that proves the equivalence between the two programs in Figure 1. Given a valid simulation relation, proving equivalence is straight-forward; however the construction of a simulation relation is undecidable in general.

Static approaches to equivalence checking attempt to construct a simulation relation purely through static analysis. On the other hand, *data-driven* approaches [7,8] extract information from the program execution traces to infer a simulation relation. In either case, the construction of a simulation relation involves the inference of the *correlation* between program locations (i.e., the first column of the simulation relation table) and the invariants at each correlated pair of locations (i.e., the second column of the simulation relation table).

A data-driven approach to inferring a correlation involves identifying program locations in the two programs where the control-flow is correlated across multiple execution runs, *and* where the number and values of the live variables in the two programs are most similar. Also, the inference of invariants in these data-driven approaches is aided by the availability of actual runtime values of the live program variables.

On the other hand, static approaches usually employ an algorithm based on the guess-and-check strategy. We discuss a static algorithm for automatic construction of a simulation relation in Section 2. Essentially it involves an incremental construction of a simulation relation, where at each incremental step,

the invariants at the currently correlated program locations are inferred (using a guess-and-check strategy) and future correlations are guided through the invariants inferred thus far. These guess-and-check based approaches are able to infer only simple forms of invariants and run into scalability bottlenecks while trying to infer more complex invariants. This is why data-driven approaches are more powerful because they can sidestep these scalability limitations by being able to infer more expressive invariants using real execution data.

However, data-driven approaches cannot work in the absence of a sufficient number of execution traces. Further, these approaches fail if certain portions (or states) of the program remain unexercised (uncovered) through the execution runs. We provide a counter-example guided strategy for invariant inference that allows us to scale our guess-and-check procedure much beyond existing approaches to guess-and-check. Our strategy resembles previous data-driven approaches; however, we are able to do this without access to execution traces, and only through counter-examples provided by modern SMT solvers. In particular, our algorithm involves *sketch generation* through syntax-guided synthesis, where a *sketch* is a template for an invariant. We use counter-examples to try and *fill* the sketch to arrive at a final invariant. To our knowledge, this is the first sketching-based approach to invariant inference and is the first contribution of our work. We call our algorithm *invariant-sketching*.

Several steps during the construction of a simulation relation involve proof-obligations (or checks) that can be represented as SMT satisfiability queries and discharged to an SMT solver. We find that modern SMT solvers face tractability limitations while computing equivalence across several compiler transformations. This is primarily due to two reasons:

1. Often, equivalence across these types of transformations are not captured in higher-order decision procedures in SMT solvers, and it appears that modern SMT solvers resort to expensive exponential-time algorithms to decide equivalence in these cases;
2. Even if these transformations are captured in SMT solvers, the composition of multiple such transformations across relatively large program fragments makes it intractable for the SMT solver to reason about them.

For the transformations that are not readily supported by modern SMT solvers, we employ *simplification passes* that can be applied over SMT expression DAGs[2], before submitting them to the SMT solver. Simplification passes involve rewriting expression DAGs using pattern-matching rules. We find several cases where the discharge of certain proof obligations during equivalence computation is tractable only after our simplification passes have been applied. We believe that our observations could inform SMT solvers and guide their optimization strategy.

The latter scalability limitation is due to the composition of multiple compiler transformations in a single program fragment. To tackle this, we propose a novel algorithm called *Query-decomposition*. Query decomposition involves breaking down a larger query into multiple sub-queries: we find that while an SMT solver

[2] A DAG is a more compact representation of an expression tree where identical subtrees in the same tree are merged into one canonical node

may find it hard to reason about one large query, it may be able to discharge tens of smaller queries in much less total time. Further, we find that counter-examples obtained from previous SMT solver queries can be used to significantly prune the number of required smaller queries. Using query-decomposition with counter-example based pruning allows us to decide more proof-obligations than were previously possible, in turn allowing us to compute equivalence across a larger class of transformations/programs. The simplification passes and our query-decomposition algorithm with counter-example based pruning are the second contribution of this paper.

In both these contributions, we make use of the `get-model` feature [9] available in modern SMT solvers to obtain counter-examples. Previous approaches to equivalence checking have restricted their interaction with SMT solvers to a one-bit SAT/UNSAT answer; we demonstrate algorithms that can scale equivalence checking procedures beyond what was previously possible through the use of solver-generated counter-examples.

Paper organization: Section 2 provides background on automatic construction of a simulation relation - our work builds upon this previous work to improve its scalability and robustness. Section 3 presents a motivating example for our work. Section 4 describes our novel sketching-based invariant inference procedure. Section 5 focusses on some important limitations of SMT solvers while reasoning about compiler optimizations, and discusses our simplification passes and the query-decomposition algorithm in this context. Section 6 discusses the experiments and results. Section 7 summarizes previous work and concludes.

2 Background: automatic generation of simulation proof

Automatic construction of a provable simulation relation between a program and its compiled output has been the subject of much research with several motivating applications. Our algorithm resembles previous work on black-box equivalence checking [5], in that it attempts to construct a simulation relation incrementally as a *joint transfer function graph* (JTFG). The JTFG is a graph with nodes and edges, and represents the partial simulation relation computed so far. A JTFG node (L_A, L_B) represents a pair of program nodes L_A and L_B and indicates that $Prog_A$ is at L_A and $Prog_B$ is at L_B. Similarly, a JTFG edge $(L_A^{'}, L_B^{'}) \rightarrow (L_A, L_B)$ represents a pair of transitions $L_A^{'} \rightarrow L_A$ and $L_B^{'} \rightarrow L_B$ in $Prog_A$ and $Prog_B$ respectively. Each JTFG node (L_A, L_B) contains invariants relating the live variables at locations L_A and L_B in the two programs respectively. Further, for each JTFG edge, the *edge conditions* (*edgecond*) of its two individual constituent edges ($L_A^{'} \rightarrow L_A$ and $L_B^{'} \rightarrow L_B$) should be equivalent. An edge condition represents the condition under which that edge is taken, as a function of the live variables at the source location of that edge.

The algorithm for constructing a JTFG is succinctly presented in Algorithm 1. Section 3 describes the running of this algorithm on an example pair of programs. The JTFG is initialized with a single node, representing the pair of entry locations of the two programs. It also has the associated invariants encoding

Algorithm 1 Algorithm to construct the JTFG (simulation relation). $edges_B$ is a list of edges in $Prog_B$ in depth-first search order. The `AddEdge()` function returns a new JTFG `jtfg'`, formed by adding the edge to the old JTFG `jtfg`.

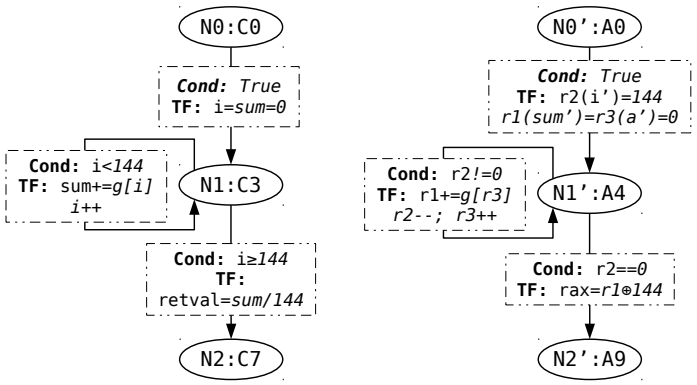
Function *CorrelateEdges(jtfg, edges$_B$)*
  if *edges$_B$ is empty* **then**
    return LiveValuesAtExitAreEq(jtfg)
  end
  edge$_B$ ← RemoveFirst(edges$_B$)
  fromPC$_B$ ← GetFromPC(edge$_B$)
  fromPC$_A$ ← FindCorrelatedPC(jtfg, fromPC$_B$)
  edges$_A$ ← GetEdgesTillUnroll(Prog$_A$, fromPC$_A$, μ)
  foreach *edge$_A$ in edges$_A$* **do**
    jtfg' = AddEdge(jtfg, edge$_A$, edge$_B$)
    PredicatesGuessAndCheck(jtfg')
    if *IsEqualEdgeConditions(jtfg')* $\wedge$ *CorrelateEdges(jtfg', edges$_B$)* **then**
      return true
    end
  end
  return false

equivalence of program values at entry (base case). The loop heads, function calls and exit locations in $Prog_B$ are then chosen as the program points that need to be correlated with a location in $Prog_A$. All other program points in $Prog_B$ are *collapsed* by composing their incoming and outgoing edges into *composite edges*. The `CorrelateEdges()` function picks one (composite) $Prog_B$ edge, say $edge_B$, at a time and tries to identify paths in the source program ($Prog_A$) that have an equivalent *path condition* to $edge_B$'s edge condition. Several candidate paths are attempted up to an unroll factor μ (`GetEdgesTillUnroll()`). All candidate paths must originate from a $Prog_A$ location (`fromPC`$_\mathtt{A}$) that has already been correlated with the source location of $edge_B$ (`fromPC`$_\mathtt{B}$). The unroll factor μ allows equivalence computation across transformations involving loop unrolling. The path condition of a path is formed by appropriately composing the edge conditions of the edges belonging to that path. The edge $edge_B$ is chosen in depth-first search order from $Prog_B$, and also dictates the order of incremental construction of the JTFG. The equivalence of the edge condition of $Prog_B$ with the path condition of $Prog_A$ is computed based on the invariants inferred so far at the already correlated JTFG nodes (`IsEqualEdgeConditions()`). These invariants, inferred at each step of the algorithm, are computed through a Houdini-style [10] guess-and-check procedure. The guesses are synthesized from a grammar, through the syntax-guided synthesis of invariants (`PredicatesGuessAndCheck`). These correlations for each edge ($edge_B$) are determined recursively to allow backtracking (the recursive call to `CorrelateEdges()`). If at any stage, an edge ($edge_B$) cannot be correlated with any path in $Prog_A$, the function returns with a failure, prompting the caller frame in this recursion stack, to try another correlation for a previously correlated edge.

`PredicatesGuessAndCheck()` synthesizes invariants through the following grammar of guessing: $\mathbb{G} = \{ \bigstar_A \otimes \bigstar_B \}$, where operator $\otimes \in \{<, >, =, \leq, \geq\}$ and $\bigstar_A$ and $\bigstar_B$ represent the live program values (represented as symbolic expressions) appearing in $Prog_A$ and $Prog_B$ respectively. The guesses are formed through a cartesian product of values in $Prog_A$ and $Prog_B$ using the patterns in $\mathbb{G}$. Our checking procedure is a sound fixed-point computation which keeps eliminating the unprovable predicates until only provable predicates remain (similar to Houdini). At each step of the fixed point computation, for each guessed predicate at each node, we try to prove it from current invariants at every predecessor node (as also done in the final simulation relation validity check in equation 2).

It is worth noting that we need to keep our guessing procedure simple to keep this procedure tractable; it currently involves only conjunctions of equality and inequality relations between the variables of the two programs. We find that this often suffices for the types of transformations produced by production compilers. In general, determining the right guesses for completing the proof is undecidable: a simple guessing grammar keeps the algorithm tractable, increasing grammar complexity significantly increases the runtime of the equivalence procedure. In our work, we augment the guessing procedure to generate *invariant-sketches* and use counter-examples to fill the sketches to arrive at the final invariants.

3 Running example



(a) Specification (b) Optimized binary

Fig. 2: TFGs of C program and its optimized implementation for the program in Fig. 1.

Figure 2 shows the abstracted versions (*aka* transfer function graphs or TFGs) of the original C specification and its optimized assembly implementation for the program in Figure 1. TFG nodes represent program locations and TFG

edges indicate control flow. Each TFG edge is associated with its *edge-condition* and its *transfer function* (labelled as `Cond` and `TF` respectively in the figure). Notably, TFG and JTFG representations are almost identical. Across the two TFGs, the program has undergone multiple compiler transformations, namely (a) loop reversal (`i` counts from `[0..144)` in the original program but counts backwards from `[144..0)` in the optimized program), (b) strength-reduction of the expensive division operation to a cheaper combination of multiply-shift-add (represented by $\oplus$), and (c) register allocation of variables `sum` and `i`. All these are common optimizations produced by modern compilers, and failing to prove equivalence across any of these (or their composition) directly impacts the robustness of an equivalence checker. (As an aside, while the general loop-reversal transformation does not preserve similarity, it preserves similarity in this example. In general, we find that modern compilers perform loop-reversal only if similarity is preserved).

Applying Algorithm 1 to this pair of TFGs, we begin with a JTFG with one node representing the start node `(N0,N0')`. Our first correlation involves correlating the loop heads of the two programs by adding the node `(N1,N1')` to the JTFG. At this point, we need to infer the invariants at `(N1,N1')`. While all other invariants can be inferred using the grammar presented in the procedure `PredicatesGuessAndCheck` in Section 2, the invariant `i' = 144-i` is not generated by our grammar. This is so because the grammar only relates the variable values computed in the two programs, but the value `144 - i` is never computed in the body of the source program.

One approach to solving this problem is to generalize our guessing grammar such that it also generates invariants of the shape $\{C_A\bigstar_A + C_B\bigstar_B + C_1 = 0\}$, where $\bigstar_A$ and $\bigstar_B$ represent program values (represented as symbolic expressions) appearing in $Prog_A$ and $Prog_B$ respectively, and C_A, C_B, and C_1 are the coefficients of $\bigstar_A$, $\bigstar_B$, and 1 respectively in this linear equality relation. C_A, C_B, and C_1 could be arbitrary constants. We call this extended grammar $\mathbb{G}'$, the grammar of *linear-equality relations*. The problem, however, is that this grammar explodes the potential number of guessed invariants, as the number of potential constant coefficients is huge. In contrast, a data-driven approach may identify the exact linear-relation through the availability of run-time values. We present an algorithm to tractably tackle this in a static setting through the generation of *invariant-sketches* in our grammar. An invariant-sketch is similar to an invariant, except that certain parts of the sketch are left unspecified. e.g., in our case, the constant coefficients C_A, C_B, and C_1 are left unspecified in the generated sketch. These unspecified constants are also called *holes* in the sketch.

We restrict C_A to be 1 and $C_B \in \{-1, 0, 1\}$ and find that this suffices for the types of transformations we have encountered in modern compilers. Generalizing to arbitrary C_A and C_B requires careful reasoning about bitvector arithmetic and associated overflows, and we leave that for future work. However, notice that we place no restrictions on C_1 — e.g., for 64-bit arithmetic, $C_1 \in [-2^{63}..2^{63}-1]$, making it prohibitively expensive to enumerate all the possibilities.

4 Invariant sketches and the use of counter-examples

We now discuss a syntax-guided invariant-sketching algorithm that uses counter-examples generated by SMT solvers to fill the sketches. The guessing grammar $\mathbb{G}'$ generates invariant-sketches, in addition to the invariants generated by $\mathbb{G}$. For example, one of the guesses generated using $\mathbb{G}'$ for our running example will be $(i + C_B i' + C_1 = 0)$. Recall that `i` represents a variable in $Prog_A$ and `i'` represents a variable in $Prog_B$ and C_B, C_1 represent holes in the generated sketch. Notice that we omit C_A as it is restricted to be $= 1$ in $\mathbb{G}'$.

Algorithm 2 InvSketch algorithm to infer invariant between var_y of Prog_A and var_x of Prog_B at Node N.

Function *InvSketch(N, e, var_x, var_y)*
- N1_A = QuerySatAssignment(e, true)
- **if** *$N1_A$ is empty* **then**
  - return (Inv $\mapsto$ {False})
- **end**
- N2_A = QuerySatAssignment(e, (var_x, var_y) != N1_A)
- **if** *$N2_A$ is empty* **then**
  - return (Inv $\mapsto$ {(var_x, var_y) = N1_A})
- **end**
- Coeff_{C_B,C_1} = InferLinearRelation(N1_A, N2_A)
- Inv = FillSketch(Coeff_{C_B,C_1})
- return Inv

For each invariant-sketch, we try to infer the potential values of the holes by querying the SMT solver for a *satisfying assignment* for the variables at the current node. A satisfying assignment N_A at a node N represents a mapping from program variables to some constants; this mapping should be satisfiable, assuming that the invariants at a predecessor node `P` and the edge condition for the edge $P \rightarrow N$ are true. For example, if a predecessor node P has an inferred invariant `x=y` and the edge condition and transfer function across the edge $P \rightarrow N$ are {`true`} and {`x=x+1, y=y+2`} respectively, then the assignment `x=3,y=3` is *not* satisfiable at N. On the other hand, the assignment `x=3,y=4` is satisfiable at N. To obtain satisfying assignments for variables at a node N, we first obtain a satisfying assignment P_A for the invariants at P and the edge condition for the edge $P \rightarrow N$ through an SMT query. We then apply the transfer function of the edge $P \rightarrow N$ on P_A to obtain N_A.

We define a procedure called `QuerySatAssignment(`$e = P \rightarrow N$`, `$\text{cond}_{\text{extra}}$`)`. This procedure generates a satisfying assignment (if it exists) for N, given the current invariants at P and the edge-condition of edge $e = P \rightarrow N$. Further, the satisfying assignment must satisfy the extra conditions encoded by the second argument $\text{cond}_{\text{extra}}$. Algorithm 2 presents our algorithm to infer the invariant, given an invariant-sketch, using satisfying assignments generated through calls to `QuerySatAssignment()`. The algorithm

infers the values of the holes C_B and C_1 (if they exist) in a given invariant-sketch. At a high level, the algorithm first obtains two satisfying assignments, ensuring that the second satisfying assignment is distinct from the first one. Given two assignments, we can substitute these assignments in the invariant-sketch to obtain two linear equations in two unknowns, namely C_B and C_1. Based on these two linear equations, we can infer the potential values of C_B and C_1 using standard linear-algebra methods. If no satisfying assignment exists through any of the predecessors of N, we simply emit the invariant `false` indicating that this node is unreachable given the current invariants at the predecessors (this should happen only if the programs contain dead-code). Similarly, if we are able to generate only one satisfying assignment (i.e., the second SMT query fails to generate another distinct satisfying assignment), we simply generate the invariant encoding that the variables have constant values (equal to the ones generated by the satisfying assignment). If a node N has multiple predecessors, we can try generating satisfying assignments through *either* of the predecessors.

Thus, for each invariant-sketch generated at each step of our algorithm, we check to see if the satisfying assignments for program variables at that node result in a valid invariant. If so, we add the invariant to the pool of invariants generated by our guessing procedure. Notice that we need at most two SMT queries per invariant-sketch; in practice, the same satisfying assignment may be re-usable over multiple invariant-sketches drastically reducing the number of SMT queries required. For our running example, we will first obtain a satisfying assignment at node `(N1,N1')` using invariants at node `(N0,N0')`: `i=0,i'=144`. However we will be unable to obtain a second satisfying assignment at this stage, and so we will generate invariants `i=0,i'=144` at `(N1,N1')`. In the next step of the algorithm, the edge `N1'`→`N1'` will be correlated with the corresponding $Prog_A$ edge `N1`→`N1`. At this stage, we will again try the same invariant-sketch, and this time we can obtain two distinct satisfying assignments at `(N1,N1')`: {`i=0,i'=144`} (due to the edge `(N0,N0')`→`(N1,N1')`) and {`i=1,i'=143`} (due to the edge `(N1,N1')`→`(N1,N1')`). Using these two satisfying assignments, and using standard linear-algebra techniques (to solve for two unknowns through two linear equations), we can infer that $C_B = 1$ and $C_1 = -144$, which is our required invariant guess. Notice that the output of our invariant-sketching algorithm is an invariant *guess*, which may subsequently be eliminated by our sound fixed-point procedure for checking the inductive validity of the simulation relation. The latter check ensures that our equivalence checking algorithm remains sound.

5 Efficient Discharge of Proof Obligations

In our running example, after the edges (`N1`→`N1`) and (`N1'`→`N1'`) have been correlated, the algorithm will infer the required invariants correlating the program values at Node `(N1,N1')`. After that, the edge (`N1'`→`N2'`) will get correlated with the edge (`N1`→`N2`) and we would be interested in proving that the final return values are identical. This will involve discharging the proof obligation

of the form: `(sum=sum')`⇒`((sum/144)=(sum'`⊕`144))`. It turns out that SMT solvers find it hard to reason about equivalence under such transformations; as we discuss in Section 6, modern SMT solvers do not answer this query even after several hours. We find that this holds for some common types of compiler transformations such as: (a) transformations involving mixing of multi-byte arithmetic operations with select/store operations on arrays, (b) transformation of the division operator to a multiply-shift-add sequence of operations, (c) complex bitvector shift/extract operations mixed with arithmetic operations, etc.

We implement *simplification passes* to enable easier reasoning for such patterns by the SMT solvers. A simplification pass converts a pattern in the expression to its simpler canonical form. We discuss the "simplification" of the division operator into a sequence of multiply-shift-add operators to illustrate this. Given a dividend `n` and a constant divisor `d`, we convert it to :

$$n \div d \;\equiv\; (n \;+\; (n \times C_{mul}) \gg 32) \gg C_{shift},$$

where $0 < \texttt{d} < 2^{32}$ `and` $0 \leq \texttt{n} < 2^{32}$ and C_{mul}, C_{shift} represent two constants dependent on d and are calculated using a method given in Hacker's Delight [11]. This simplification ensures that if the compiler performs a transformation that resembles this pattern, then both the original program and the transformed program will be simplified to the same canonical expression structure, which will enable the SMT solvers to reason about them more easily. We find that there exist more patterns that exhibit SMT solver time-outs by default, but their simplified versions (through our custom simplification passes) become tractable for solving through modern SMT solvers.

Even after applying the simplification passes, we find that several SMT queries still time-out because SMT solvers find it difficult to reason about equivalence in the face of several composed transformations performed by the compiler. We observe that while SMT solvers can easily compute equivalence across a smaller set of transformations, they often time out if the number of composed transformations is too many or too intertwined. Taking a cue from this observation, we propose the *query-decomposition* algorithm.

The general form of proof queries in an equivalence checker is: $Precond \Rightarrow (LHS = RHS)$, where $Precond$ represents a set of pre-conditions (e.g., `x=y`) and LHS and RHS expressions (e.g., `x+1` and `y+2`) are obtained through symbolic execution of the C specification (Prog_A) and the optimized program (Prog_B) respectively. The RHS expression may contain several composed transformations for the computation performed in the LHS expression. Query-decomposition involves breaking this one large proof query into multiple smaller queries by using the following steps:

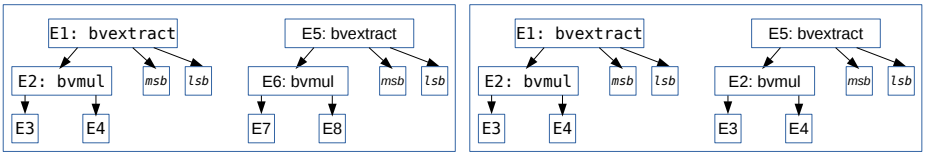
1. We walk the expression DAGs of LHS and RHS and collect all unique subexpressions in LHS and RHS into two different sets, say $\{lhsSubExprs\}$ and $\{rhsSubExprs\}$.
2. We check the equivalence of each $lhsSubExpr \in \{lhsSubExprs\}$ with each $rhsSubExpr \in \{rhsSubExprs\}$ (assuming $Precond$), in increasing order of size of LHS sub expressions. i.e., $Precond \Rightarrow (lhsSubExpr = rhsSubExpr)$.

The size of an expression is obtained by counting the number of operators in its expression DAG. If there are m unique sub-expressions in $\{lhsSubExprs\}$ and n unique sub-expressions in $\{rhsSubExprs\}$, we may need to perform $m * n$ equivalence checks.

3. For any check that is successful in step 2, we learn a substitution map from the bigger expression to the smaller expression. For example, if $lhsSubExpr$ was smaller in size than $rhsSubExpr$, we learn a substitution mapping $rhsSubExpr \mapsto lhsSubExpr$. We maintain a set of substitution mappings thus learned.
4. For any check that is unsuccessful in step 2, we learn a counter-example that satisfies $Precond$ but represents a variable-assignment that shows that $lhsSubExpr = rhsSubExpr$ is not provable. We add all such counter-examples to a set $\{counterExamples\}$.
5. In all future equivalence checks (of the total $m*n$ checks) of type $Precond \Rightarrow (lhsSubExpr = rhsSubExpr)$, we first check the set $\{counterExamples\}$ to see if any $counterExample \in \{counterExamples\}$ disproves the query. If so, we have already decided the equivalence check as false. If not, we rewrite both expressions $lhsSubExpr$ and $rhsSubExpr$ using the substitution-map learned in step 3. For a substitution mapping $e1 \rightarrow e2$, we replace every occurrence of $e1$ in an expression with $e2$ during this rewriting. After the rewriting procedure reaches a fixed-point, we use an off-the-shelf SMT solver to decide the rewritten query.

This decomposition of a larger expression into sub-expressions, and the substitution of equivalent sub-expressions while deciding equivalence of larger expressions ensures that the queries submitted to SMT solvers are simpler than the original query. In other words, through this strategy, the LHS' and RHS' expressions that are submitted to an SMT solver are more similar to each other, as multiple composed transformations have likely been decomposed into fewer transformations in each individual query. Because we only replace provably-equivalent sub-expressions during decomposition, the overall equivalence checking algorithm remains sound.

This bottom-up strategy of decomposing a larger query into several smaller queries would be effective only if (a) we expect equivalent sub-expressions to appear across LHS and RHS, and (b) the total time to decide equivalence for multiple sub-expression pairs is smaller than the time to decide equivalence for a single larger expression-pair. We find that both these criteria often hold while comparing distinct expressions that differ in the transformations performed by a compiler. Figure 3 illustrates this with an example. In this example, the proof query involves deciding equivalence between the top-level expressions $E1$ and $E5$. Also, it turns out that the sub-expressions $E3$ and $E4$ (on the left) are equivalent to the sub-expressions $E7$ and $E8$ (on the right) respectively. This query gets generated when comparing the unoptimized and optimized implementations generated by GCC for a fragment of a real-world program. Notably, modern SMT solvers like Z3/Yices/Boolector time-out even after several hours for such a query. On the other hand, they are able to decide the equivalence



(a) Before Decomposition (b) After Decomposition

Fig. 3: Original and result of query-decomposition for a fragment of expression DAGs for two SMT-expressions.

of individual sub-expressions ($E3 = E7$ and $E4 = E8$) within a few seconds. Experimentally, we have observed that if we substitute $E7$ and $E8$ with $E3$ and $E4$ respectively (in the expression DAG of $E5$), the resulting equivalence check between $E1$ and the rewritten $E5$ (as shown in Figure 3(b)) also completes within a fraction of a second. In section 6, we discuss more real-world functions and our results with transformations produced by multiple compilers, to demonstrate the effectiveness of our query-decomposition procedure. We also evaluate the fraction of intermediate sub-expression queries that can be pruned through the use of the counter-example set.

6 Experiments

We evaluate invariant-sketching and query-decomposition algorithms by studying their effectiveness in a black-box equivalence checker across LLVM IR and x86 assembly code. We compile a C program using LLVM's clang-3.6 to generate unoptimized (O0) LLVM IR bitcode and using GCC, LLVM, ICC (Intel C Compiler), and CComp (CompCert [12]) with O2 optimization to generate the x86 binary executable code. We have written symbolic executors for LLVM bitcode and x86 instructions to convert the programs to their logical QF_AUFBV SMT-like representation. In this representation, program states including the state of LLVM variables, x86 registers and memory are modelled using bit-vectors and byte-addressable arrays respectively. Function calls are modelled through uninterpreted functions. The black-box equivalence checking tool employs the algorithm discussed in Section 2 with $\mu = 1$. The tool also models undefined-behaviour semantics of the C language [13] for improved precision in equivalence checking results. Proof obligations are discharged using Yices [14] (v2.5.4) and Z3 [15] (commit 0b6a836eb2) SMT solvers running in parallel: each proof obligation is submitted to both solvers, and the result is taken from the solver that finishes first. We use a time-out value of five hours for each proof obligation.

Benchmarks and Results: For evaluation, we use C functions from the SPEC CPU Integer benchmarking suite [16] that contain loops and cannot be handled by previous equivalence checking algorithms. Previous work on black box equivalence checking [5] fails to compute equivalence on all these functions. We also include the benchmarks used by previous work on data-driven equivalence

S.No.	Benchmark	Function	SLOC	ALOC	Checking Time (sec)			
					T_{gcc}	T_{clang}	T_{icc}	T_{ccomp}
B1	knucleotide	ht_hashcode	5	28	**17**	**18**	**215**	**10**
B2	nsieve	main	11	39	**1343**	1687	**2265**	868
B3	sha1	do_bench	11	49	**338**	320	385	383
B4	DDEC	lerner1a	12	22	**37**	**13**	**36**	**12**
B5	twolf	controlf	8	16	**73**	**79**	**75**	✗
B6	gzip	display_ratio	21	78	**738**	**121**	**677**	**570**
B7	vpr	is_cbox	8	48	**24**	**25**	**24**	**24**
B8	vpr	get_closest_seg_start	14	57	**27**	**27**	**27**	**27**
B9	vpr	get_seg_end	16	63	**28**	**29**	**29**	**30**
B10	vpr	is_sbox	16	79	**34**	**33**	**32**	***Fail***
B11	vpr	toggle_rr	7	25	**131**	**121**	**99**	✗
B12	bzip2	makeMaps	11	39	**217**	**240**	214	221

Table 2: Benchmarks characteristics. SLOC is source lines of code and determined through the sloc count tool. ALOC is assembly lines of code and is based on gcc-O0 compilation. T_X represents equivalence checking time taken for executable generated by "X" compiler in seconds. ✗ represents that the function could not be compiled with that particular compiler.

checking [7] in our evaluation; we are able to statically compute equivalence for these benchmarks, where previous work relied on execution data for the same. The selected functions along with their characteristics and results obtained for each function-compiler pair are listed in Table 2.

The results in bold-red typeface depict the function-compiler pairs for which previous work fails to prove equivalence statically. Computing equivalence for these programs requires either sophisticated guessing procedures (which we address through our invariant-sketching algorithm) or/and involves complex proof queries that would time-out on modern SMT solvers (addressed by our simplification and query-decomposition procedures). The results in non-bold face depict the function-compiler pairs for which equivalence can be established even without our algorithms — in these cases, the transformations performed by the compiler require neither sophisticated guessing nor do their proof obligations time-out. For most cases, by employing our algorithms, the execution time for establishing equivalence is reasonably small. In general, we observe that the equivalence checking time depends on the size of the C program and the number and complexity of transformations performed by the compiler. For one of the benchmarks (`is_sbox` compiled through `CComp`), equivalence could not be established even after employing our invariant-sketching, simplification and query-decomposition algorithms. We next evaluate the improvements obtained by using counter-examples to prune the number of queries discharged to SMT solver. Recall that additional queries are generated by both invariant-sketching and query-decomposition algorithms. Also, the query-decomposition algorithm maintains a set of counter-examples and a substitution-map learned so far, to

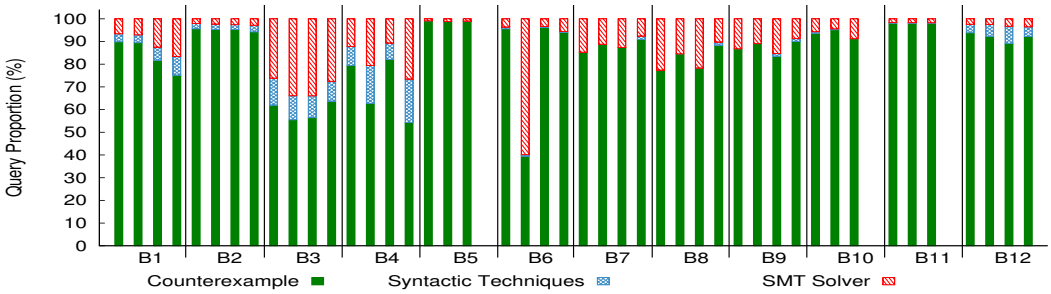


Fig. 4: Proof queries statistics. The bar represents the percentage of queries solved by each strategy for each benchmark-compiler pair in the same order as in Table 2.

reduce the time required to discharge a query. For each query, we first check to see if the query can be answered through the set of currently-available counter-examples. If not, the second step involves rewriting the query through simplification passes and the currently-available substitution-map to see if the resulting query can be answered syntactically. Finally, if equivalence is not decidable even after simplification and substitution, we submit the simplified query to the SMT solver. Figure 4 provides a break-down of the fraction of queries answered by counter-examples, by syntactic simplification and substitution, and by the SMT solver. The counter-example set is able to answer more than 85% of the total proof queries, including the ones generated by our invariant-sketching and query-decomposition algorithms. Similarly, syntactic simplification and substitution are able to answer 3% of the queries, while the remaining 12% of the queries are answered by the SMT solver. Recall that the simplification and substitution passes help ensure that the 12% queries can be answered by the SMT solver efficiently; the SMT solver would often time-out without these simplifications and substitutions.

7 Related Work and Conclusions

Combinational equivalence checking through SAT-based algorithms has been studied extensively both in hardware verification [17] and in software verification [18]. Equivalence checking for sequential programs (e.g., programs with loops) has also been studied extensively in the context of translation validation [2,19], design and verification of compiler optimizations [20,21], and program synthesis and superoptimization [3,4,22,23,24,25]. Modern SMT solvers have further facilitated these applications (over traditional SAT solvers) by raising the level of abstraction at the interface of the SMT solver and the equivalence checker. Improving the capabilities of SMT solvers for various practical applications remains an important field of research [26,27]. For example, Limaye and Seshia [27] describe word-level simplification of queries before submitting them to SMT solvers; our simplification passes are similar to such previous work.

Our work studies the effective utilization of SMT solvers for the problem of equivalence checking for sequential programs containing loops. We demonstrate techniques that allow an equivalence checker to decide equivalence across a wider variety of programs and transformations; our invariant-sketching and query-decomposition algorithms are novel contributions in this context.

Data-driven equivalence checking and data-driven invariant inference are recent approaches ([7,28,8,29]) that utilize the information obtained through running the programs on real inputs (execution traces), for inferring the required correlations and invariants across the programs being compared for equivalence. It is evident that data-driven approaches are more powerful than static approaches in general; however they limit the scope of applications by demanding access to high-coverage execution traces. Our invariant-sketching algorithm allows us to obtain the advantages of data-driven approaches in a static setting, with no access to execution traces. Our experiments include the test programs used in these previous papers on data-driven equivalence checking, and demonstrate that a counter-example guided invariant-sketching scheme can achieve the same effect without access to execution traces. Further, some of the data-driven techniques, such as CEGIR [29] and Daikon [30], are unsound, i.e., they may return invariants that are not inductively provable but are only good enough for a given set of execution traces or the capabilities of a given verification tool. Unsound strategies are not useful for several applications of equivalence checking, such as translation validation and program synthesis. Both invariant-sketching and query-decomposition algorithms preserve soundness.

Recent work on synthesizing models for quantified SMT formulas [31] involves a similar computational structure to our invariant-sketching technique; the primary differences are in our use of a linear interpolation procedure (`InferLinearRelation`), and consequently the small number of invariant-synthesis attempts (at most two) for each invariant-sketch. These techniques make our procedure tractable, in contrast to the approach of synthesizing models for general quantified SMT formulas outlined in [31]. Invariant-sketching also has a parallel with previous approaches on counter-example guided abstraction refinement, such as recent work on worst-case execution time analysis [32]. From this perspective of abstraction refinement, our invariant-sketching algorithm refines an invariant from `(Inv` $\mapsto$ `{False})` to `(Inv` $\mapsto$ `{(var`$_x$`, var`$_y$`) = N1`$_A$`})` to the final linearly-interpolated invariant based on the invariant-sketch. This counter-example guided refinement is aided by invariant-sketches involving linear relations, that are designed to capture the underlying structure of the equivalence checking problem.

The query-decomposition algorithm for effective utilization of SMT solvers is based on our experiences with multiple SMT solvers. It is indeed interesting to note that SMT solvers can decide many smaller queries in much less time than one equivalent bigger query. This observation has motivated our decomposition algorithm, and our experiments show its efficacy in deciding equivalence across programs, where previous approaches would fail.

References

1. Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11, New York, NY, USA, ACM (2011) 295–305
2. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, New York, NY, USA, ACM (2009) 264–276
3. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XII, New York, NY, USA, ACM (2006) 394–403
4. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '13, New York, NY, USA, ACM (2013) 305–316
5. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings. (2017) 127–147
6. Zuck, L., Pnueli, A., Goldberg, B., Barrett, C., Fang, Y., Hu, Y.: Translation and run-time validation of loop transformations. Form. Methods Syst. Des. **27**(3) (November 2005) 335–360
7. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '13, New York, NY, USA, ACM (2013) 391–406
8. Padhi, S., Sharma, R., Millstein, T.: Data-driven precondition inference with learned features. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '16, New York, NY, USA, ACM (2016) 42–56
9. Barrett, C., Stump, A., Tinelli, C.: The smt-lib standard - version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories, Edinburgh, Scotland. (2010)
10. Flanagan, C., Leino, K.: Houdini, an annotation assistant for esc/java. In: FME 2001: Formal Methods for Increasing Software Productivity. Volume 2021 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 500–517
11. Warren, H.S.: Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
12. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd ACM symposium on Principles of Programming Languages, ACM Press (2006) 42–54
13. Dahiya, M., Bansal, S.: Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In: Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings. (2017) 19–34
14. Dutertre, B.: Yices2.2. In Biere, A., Bloem, R., eds.: Computer Aided Verification, Cham, Springer International Publishing (2014) 737–744

15. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'08/ETAPS'08, Berlin, Heidelberg, Springer-Verlag (2008) 337–340
16. Henning, J.L.: Spec cpu2000: Measuring cpu performance in the new millennium. Computer **33**(7) (July 2000) 28–35
17. Popek, G.J., Goldberg, R.P.: Formal requirements for virtualizable third generation architectures. Commun. ACM **17**(7) (July 1974) 412–421
18. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. OSDI'08, Berkeley, CA, USA, USENIX Association (2008) 209–224
19. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '09, New York, NY, USA, ACM (2009) 327–337
20. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. PLDI '00, New York, NY, USA, ACM (2000) 83–94
21. Tate, R., Stepp, M., Lerner, S.: Generating compiler optimizations from proofs. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '10, New York, NY, USA, ACM (2010) 389–402
22. Bansal, S., Aiken, A.: Binary translation using peephole superoptimizers. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. OSDI'08, Berkeley, CA, USA, USENIX Association (2008) 177–192
23. Massalin, H.: Superoptimizer: A look at the smallest program. In: ASPLOS '87: Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems. (1987) 122–126
24. Phothilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: Scaling up superoptimization. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '16, New York, NY, USA, ACM (2016) 297–310
25. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Conditionally correct superoptimization. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2015, New York, NY, USA, ACM (2015) 147–162
26. Slflow, A., Khne, U., Fey, G., Groe, D., Drechsler, R.: Wolfram- a word level framework for formal verification. In: 2009 IEEE/IFIP International Symposium on Rapid System Prototyping. (June 2009) 11–17
27. Jha, S., Limaye, R., Seshia, S.A.: Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In: Proceedings of the 21st International Conference on Computer Aided Verification. CAV '09, Berlin, Heidelberg, Springer-Verlag (2009) 668–674
28. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13, Berlin, Heidelberg, Springer-Verlag (2013) 574–592

29. Nguyen, T., Antonopoulos, T., Ruef, A., Hicks, M.: Counterexample-guided approach to finding numerical invariants. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017, New York, NY, USA, ACM (2017) 605–615
30. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1-3) (December 2007) 35–45
31. Preiner, M., Niemetz, A., Biere, A.: Counterexample-guided model synthesis. In Legay, A., Margaria, T., eds.: Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Heidelberg, Springer Berlin Heidelberg (2017) 264–280
32. Černý, P., Henzinger, T.A., Kovács, L., Radhakrishna, A., Zwirchmayr, J.: Segment abstraction for worst-case execution time analysis. In Vitek, J., ed.: Programming Languages and Systems, Berlin, Heidelberg, Springer Berlin Heidelberg (2015) 105–131