A Solver for Reachability Modulo Theories

Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri

Microsoft Research
{akashl, qadeer, shuvendu}@microsoft.com

Abstract. Consider a sequential programming language with control flow constructs such as assignments, choice, loops, and procedure calls. We restrict the syntax of expressions in this language to one that can be efficiently decided by a satisfiability-modulo-theories solver. For such a language, we define the problem of deciding whether a program can reach a particular control location as the reachability-modulo-theories problem. This paper describes the architecture of Corral, a semi-algorithm for the reachability-modulo-theories problem. Corral uses novel algorithms for inlining procedures on demand (Stratified Inlining) and abstraction refinement (Hierarchical Refinement). The paper also presents an evaluation of Corral against other related tools. Corral consistently outperforms its competitors on most benchmarks.

1 Introduction

The reachability problem, with its roots in the classical theory of finite state machines [20], asks the following question: given a (control flow) graph over a set of nodes and edges, an initial state, and an error state, does there exist a path from the initial to the error state? Subsequent to the recognition that a large class of computer systems can be modeled as finite state machines, this problem received a lot of attention from researchers interested in formal verification of computer systems [19, 7]. Over the years, many variations of this problem have been proposed to model increasingly complex systems. For example, finite control is augmented by a stack to model procedure calls in imperative programs or by a queue to model message passing in concurrent programs. Along a different dimension, researchers have proposed annotating the nodes and edges of the graph by a finite alphabet to enable specification of temporal behavior [30].

We are concerned with the problem of reasoning about programs written in real-world imperative programming languages such as C, C#, and Java. Because such programs routinely use unbounded data values such as integers and the program heap, the framework of finite-state machines is inadequate for modeling them. We propose that the semantic gap between the programming languages and the intermediate modeling and verification language should be reduced by allowing modeling constructs such as uninterpreted functions and program variables with potentially unbounded values, such as integers, arrays, and algebraic datatypes. We refer to the reachability problem on such a modeling language as *reachability-modulo-theories*. Even though this generalization immediately leads to a reachability problem that is undecidable, we feel that this direction is a promising one for the following reasons. First, the increased expressiveness dramatically simplifies the task of translating imperative software to the intermediate language, thus making it much easier to quickly implement translators from languages. In our own work, we have developed high-fidelity translators both for C and .NET bytecode with a relatively modest amount of engineering effort. The presence of an intermediate modeling and verification language simplifies the construction of end-to-end verification systems by decoupling the problem of model construction from the problem of solving reachability queries on the model. Second, our generalization leads to an intermediate verification language whose expression language is expressible in the framework of satisfiability-modulo-theories (SMT) and decidable efficiently using the advanced solvers [15, 12] developed for this framework. Therefore, reachability on bounded program fragments can be decided by converting them into verification conditions [14, 4]. The ability to decide bounded reachability in a scalable fashion can be of tremendous value in automated bug-finding and debugging.

This paper describes CORRAL, a solver for a restricted version of the reachability-modulo-theories problem, in which the depth of recursion is bounded by a user-supplied *recursion bound* (we assume that loops are converted to recursive procedures).[1] As discussed earlier, recursion-bounded reachability-modulo-theories is decidable if the expression language of programs is decidable. In fact, the simplest method to solve this problem is to statically inline all procedures up to the recursion bound, convert the resulting program into a verification condition (VC), and present the VC to an SMT solver. While this approach may work for small programs, it is unlikely to scale because the inlined program may be exponentially large. To make this point concrete, consider the restricted case of recursion-free (and loop-free) programs. For these programs, a recursion bound of 0 suffices for full verification; However, it is still a nontrivial problem to solve because the inlined program could be exponential in the size of the original program. This complexity is fundamental, in fact, the reachability-modulo-theories for recursion-free programs becomes PSPACE-hard even if we allow just propositional variables. If we add other theories such as uninterpreted functions, arithmetic, or arrays that are decidable in NP, reachability-modulo-theories for recursion-free programs is decidable in NEXPTIME; however, we conjecture that the problem is NEXPTIME-hard. Since the efficient (in practice) subset of theories decided by an SMT solver are in the complexity class NP, it is likely that that the exponential complexity of inlining is unavoidable in the worst case; CORRAL provides a solution to avoid (or delay) this exponential complexity.

1.1 The Corral architecture

CORRAL embodies a principled approach to tackling the complexity of recursion-bounded reachability-modulo-theories (RMT). Its overall architecture is shown

[1] CORRAL can also be used in a loop where the recursion bound is increased iteratively, it which case it is a semi-algorithm to the reachability-modulo-theories problem.

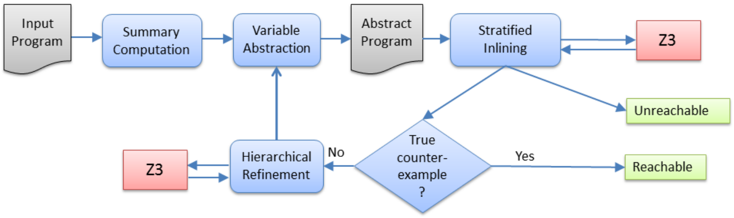

Fig. 1. CORRAL's architecture.

in Figure 1. CORRAL uses BOOGIE [3] as the modeling language for encoding reachability queries. The BOOGIE language already supports the essential requirements of RMT: it has the usual control-flow constructs (branches, loops, procedures), unbounded types (integers and maps) and operations such as arithmetic and uninterpreted functions. Moreover, the BOOGIE framework [3] comes equipped with verification-condition generation algorithms that can convert a call-free fragment of code with an assertion to an SMT formula such that the latter is satisfiable if and only if there is an execution of the code that violates the assertion. This formula can then be fed to various solvers, including Z3 [12]. Thus, the BOOGIE framework can solve the loop-free call-free segment of RMT.

CORRAL has a two-level counterexample-guided abstraction refinement [9] (CEGAR) loop. The top level loop performs a localized abstraction over global variables [9, 23]. Given a set of tracked variables T, it abstracts the input program using T; the initial set of tracked variables is empty. Next, it feeds the abstracted program to the module denoted *Stratified Inlining* (§2) to look for a counterexample. Since VC generation is quadratic in the number of program variables, performing variable abstraction before stratified inlining can improve the latter's performance significantly. If stratified inlining finds a counterexample, it is checked against the entire set of global variables. If the path is feasible, then it is a true bug; otherwise, we call the module denoted *Hierarchical Refinement* (§3.1) to minimally increase the set of tracked variables, and then repeat the process. Besides the outer loop that increases T, both the stratified inlining and hierarchical refinement algorithms have their own iterative loops that require multiple calls to Z3.

Stratified inlining For a single procedure program, we simply generate its VC and feed it to Z3. In the presence of multiple procedures, instead of inlining all of them up to a given bound, we only inline a few, generate its VC and ask Z3 to decide reachability. If it finds a counterexample, then we're done. Otherwise, we replace every non-inlined call site with a summary of the called procedure. This results in a VC that over-approximates the program, which is fed to Z3. A counterexample in this case (if any) tells us which procedures to inline next. By default, CORRAL uses a summary that havocs all variables potentially modified by the procedure, but more precise summaries can be obtained from any static analysis. In our experiments, we used HOUDINI [17] to compute summaries.

Hierarchical refinement This algorithm takes a divide-and-conquer approach to the problem of discovering a minimal set of variables needed to refute an infeasible counterexample. Suppose n is the total number of variables. In the common case when the number of additional variables needed to be tracked is small compared to n, our algorithm makes only $O(log(n))$ path queries to Z3, as compared to a previous algorithm [23] that makes $O(n)$ queries.

We have evaluated CORRAL on a large collection of benchmarks to measure its robustness and performance (§4). To demonstrate robustness, we ran CORRAL on programs obtained from multiple sources; they are either sequential C programs, or concurrent C programs sequentialized using the Lal-Reps algorithm [25]. To demonstrate good performance, we compare against existing methods: hierarchical refinement is compared against the STORM refinement algorithm, stratified inlining is compared against static inlining, and the entire CORRAL system is compared against a variety of software verification tools such as SLAM [2], YOGI [29], CBMC [8], and ESBMC [11]. Our evaluation shows that CORRAL performs significantly better than existing tools. Moreover, every tool other than CORRAL had a difficult time on some benchmark suite (i.e., the tool would run out of time or memory on most programs in that suite).

Contributions In summary, this paper makes the following contributions:

- The stratified inlining algorithm.
- The hierarchical refinement algorithm for refining variable abstraction.
- The design and implementation of CORRAL, a novel architecture that combines summaries, variable abstraction, and stratified inlining to solve the reachability-modulo-theories problem.
- Extensive experimental evaluation to demonstrate the robustness and performance of CORRAL.

2 Stratified Inlining

We consider a simple imperative programming language in which each program has a vector of global variables denoted by g and a collection of procedures, each of which has a vector of input parameters denoted i and a vector of output parameters denoted by o. Given such a program F (possibly with recursion, but no loops), a procedure `m` in the program, an initial condition $\varphi(i, g)$ over i and g, and a final condition $\psi(o, g)$ over o and g, the goal is to determine whether there is an execution of `m` that begins in a state satisfying φ and ends in a state satisfying ψ. The pseudo-code of our algorithm is shown in Fig. 2.

Stratified inlining uses under- and over-approximation of procedure behaviors to inline procedures on demand. To underapproximate a procedure, we simply block all executions through it, i.e., we replace it with “assume false”. To overapproximate a procedure, we use a *summary* of the procedure, i.e., a valid over-approximation of the procedure’s behaviors. The default summary of a procedure havocs all variables potentially modified by it and leaves the output value completely unconstrained. One may use any static analysis to compute better summaries. In some of our experiments (§4.2), we used HOUDINI [17] to compute summaries, which improved the performance of Corral over using the default summaries. We call the overapproximation of a procedure `p` as *summary*(`p`).

The algorithm maintains a partially-inlined program P starting from the procedure `m` (line 1). It also maintains the set C of non-inlined (or *open*) call-sites in P. A call-site is defined as a dynamic instance of a procedure call, i.e., the procedure call along with the runtime stack of unfinished calls. For instance, the call-site $c = [\texttt{m}; \texttt{foo1}; \texttt{foo2}]$ refers to a call to `foo2` from `foo1`, which is turn was called from `m`.

We define the *cost* of a call-site c to be the number of occurrences of the top-most procedure in c. For instance, if $c = [\texttt{m}; \texttt{foo1}; \texttt{foo2}; \texttt{foo1}; \texttt{foo2}; \texttt{foo1}]$ then *cost*(c) is 3 because `foo1` occurs thrice in c. In other words, the cost of a call-site c reflects the number of recursive calls necessary to reach c. Given a set C of call-sites, the function *split-on-cost*(C, r) partitions C into two disjoint sets C_1 and C_2, where the latter has all those call-sites whose cost is r or greater.

```
Input: Program $F$ and its starting procedure m
Input: An initial condition $\varphi$ and a final condition $\psi$
Input: Maximum recursion depth MAX
Output: Correct, Bug($\tau$), or NoBugFound
 1: $P$ := {assume $\varphi$; m; assert $\neg\psi$}
 2: $C$ := open-call-sites($P$)
 3: for $r = 1$ to MAX do
 4:     while true do
 5:         //Query-type 1
 6:         $P' = P[\forall_{c \in C} \; c \leftarrow$ assume false$]$
 7:         if check($P'$) == Bug($\tau$) then
 8:             return Bug($\tau$)
 9:         $C_1, C_2$ := split-on-cost($C, r$)
10:         //Query-type 2
11:         $P' = P[\forall_{c \in C_1} c \leftarrow$ summary($c$), $\forall_{c \in C_2} \; c \leftarrow$ assume false$]$
12:         ret := check($P'$)
13:         if ret == Correct $\wedge \; C_2 == \emptyset$ then
14:             return Correct
15:         if ret == Correct $\wedge \; C_2 \neq \emptyset$ then
16:             break
17:         let Bug($\tau$) = ret
18:         $P$ := $P$ + {inline($c$) | $c \in C, c \in \tau$}
19:         $C$ := open-call-sites($P$)
20: return NoBugFound
```

Fig. 2. The stratified inlining algorithm.

Each iteration of the loop at line 3 looks for a bug (a valid execution of `m` that violates ψ) within the cost r. The loop at line 4 inlines procedures on-demand. Each of its iterations comprise of two main steps. In the first step, each open call-site in P is replaced by its underapproximation (line 6) to obtain a closed program P', which is checked using a theorem prover (line 7). The routine *check* takes a bounded program as input and feeds it to the theorem prover to determine satisfiability of the assertion in the program. If a satisfying path is found, the algorithm terminates with Bug (line 8).

The second step involves overapproximation. Line 11 replaces each call-site c that has not reached the bound r (i.e., $c \in C_1$) with the summary of the called procedure. Other call-sites $c \in C_2$ are still blocked because their cost is r or more. If the resulting program is correct and C_2 was empty, then all open call-sites were overapproximated. Thus, the original program F has no bugs (line 14). If C_2 was not empty, then r is not sufficient to conclude any answer, thus we break to line 3 and increment r. If the check on line 11 found a trace τ, then τ must pass through some call-sites in C (because line 8 was not taken). All open calls on τ are inlined and the algorithm repeats.

Iteratively increasing the recursion bound ensures that in the limit (when MAX approaches ∞), stratified inlining is complete for finding bugs.

The main advantages of the stratified inlining algorithm are:

1. The program provided to the automated theorem prover is generated incrementally. Eager static inlining (which inlines all procedures upfront up to the recursion bound) creates an exponential explosion in the size of the program. Stratified inlining delays this exponential explosion.
2. Stratified inlining inlines only those procedures that are relevant to ruling out spurious counterexamples. Thus, it can often perform the search while inlining few procedures. This ability makes the search property-guided. Because of the use of over-approximations, stratified inlining can be faster than static inlining even for correct programs.
3. If the program is buggy, then a bug will eventually be found, assuming that the theorem prover always terminates and MAX $= \infty$.

3 Variable Abstraction and Refinement

It is often the case that a majority of program variables are not needed for proving or disproving reachability of a particular goal. In this case, abstracting away such variables can help the stratified inlining algorithm because: (1) The VC construction is quadratic in the number of variables [4]; and (2) the theorem prover does not get distracted by irrelevant variables. However, abstraction can lead to an over-approximation of the original program and lead to spurious counterexamples. In this case, we use a refinement procedure to bring back some of the variables that were removed.

We apply variable abstraction to only global variables. At any point in time, the set of global variables that are retained by the abstraction are called *tracked* variables. Note that Boogie programs always have a finite number of global variables; unbounded structures in real programs such as the heap are modeled using a finite number of maps. Thus, the refinement loop always terminates.

Variable abstraction, also known as *localization reduction* [9], has been used extensively in hardware verification. We now briefly describe the variable abstraction algorithm implemented in Corral (Fig. 3); this algorithm was previously implemented in the Storm checker [23]. Later, we present a new refinement algorithm called *hierarchical refinement* (§3.1) that significantly out-performs the refinement algorithm in Storm [23].

Let *IsGlobalVar* be a predicate that is *true* only for global variables. Let *GlobalVars* be a function that maps an expression to the set of global variables that appear in that expression. Let T be the current set of tracked variables. Variable abstraction is carried out as a program transformation, applied statement-by-statement, as shown in Fig. 3. Essentially, assignments to variables that are not tracked are completely removed (replaced by “assume true”). Further, expressions that have an untracked global variable are assumed to evaluate to any value. Consequently, assignments whose right-hand side have such an expression are converted to non-deterministic assignments (havoc statements).

$v := e$ $\mapsto \begin{cases} \text{assume true} & \mathit{IsGlobalVar(v)} \land v \notin T \\ \text{havoc } v & \mathit{GlobalVars(e)} \not\subseteq T \\ v := e & \text{otherwise} \end{cases}$	assume e $\mapsto \begin{cases} \text{assume true} & \mathit{GlobalVars(e)} \not\subseteq T \\ \text{assume } e & \text{otherwise} \end{cases}$
assert e $\mapsto \begin{cases} \text{assert false} & \mathit{GlobalVars(e)} \not\subseteq T \\ \text{assert } e & \text{otherwise} \end{cases}$	havoc v $\mapsto \begin{cases} \text{assume true} & v \notin T \\ \text{havoc } v & \text{otherwise} \end{cases}$

Fig. 3. Program transformation for variable abstraction, with tracked variables T.

3.1 Hierarchical Refinement

In Corral, we abstract the program using variable abstraction and then feed it to the stratified inlining routine. If it returns a counterexample, say τ, which exhibits reachability in the abstract program, then we check to see if τ is feasible in the original program by *concretizing* it, i.e., we find the corresponding path in the input program. If this path is infeasible, then τ is a spurious counterexample. The goal of refinement is to figure out a *minimal* set of variables to track that rule out the spurious counterexample.[2] Storm's refinement algorithm already computes a minimal set, but the algorithms presented in this section find such a set much faster.

Let us define two subroutines: *Abstract* takes a path and a set of tracked variables and carries out variable abstraction on the path. The subroutine *check* takes a path (with an assert) and returns Correct only when the assert cannot be violated on the path. We implement *check* by simply generating the VC of the entire path and feeding it to Z3.

Let G be the entire set of global variables and T be the current set of tracked variables. Storm's refinement algorithm requires about $|G-T|$ number of iterations, and each iteration requires at least one call to *check*. In our experience, we have found that most spurious counterexamples can be ruled out by tracking one or two additional variables. We leverage this insight to design faster algorithms.

Alg. 1 uses a divide-and-conquer strategy. It has best-case running time when only a few additional variables need to be tracked, in which case the algorithm requires $\log(|G-T|)$ number of calls to *check*. In its worst case, which happens when all variables need to be tracked, it requires at most $2|G-T|$ number of calls to *check*. As our experiments show, most refinement queries tend to be towards the best case, which is an exponential improvement over Storm.

Alg. 1 uses a recursive procedure *hrefine* that takes three inputs: the set of tracked variables (T), the set of do-not-track variables (D, initially empty), and a path P. It assumes that P, when abstracted with $G - D$, is correct (i.e., assertion in $\mathit{Abstract}(P, G-D)$ cannot be violated). We refer to Alg. 1 as the top-level call $\mathit{hrefine}(T, \emptyset, P)$. Note that while *hrefine* is running, the arguments T and D can change across recursive calls, but P remains fixed to be the input counterexample.

[2] We do not attempt to find the *smallest* set of variables to track; not only might that be very hard to compute, but a minimal set already gives good overall performance for Corral.

```
Procedure hrefine(T, D, P)
Input: A correct path P with global variables G
Input: A set of variables T ⊆ G that must be tracked
Input: A set of variables D ⊆ G that definitely need not be tracked, D ∩ T = ∅
Output: The new set of variables to track

1:  if $T \cup D = G$ then
2:      return $T$
3:  $P' := \mathit{Abstract}(P, T)$
4:  if $\mathit{check}(P') ==$ CORRECT then
5:      return $T$
6:  if $|G - (T \cup D)| = 1$ then
7:      return $G - D$
8:  $T_1, T_2 := \mathit{partition}(G - (T \cup D))$
9:  $S_1 := \mathit{hrefine}(T \cup T_2,\ D,\ P)$
10: $S'_1 := S_1 \cap T_1$
11: return $\mathit{hrefine}(T \cup S'_1,\ D \cup (T_1 - S'_1),\ P)$
```

(a) **Algorithm 1**: *hrefine*$(T, \emptyset, P)$

```
Procedure vcrefine(T, D, P)
Input: A program P with special Boolean constants B
Input: A set of Boolean constants T ⊆ B that must be set to true
Input: A set of Boolean constants D ⊆ B that must be set to false
Output: The set of Boolean constants to set to true
1:  if $T \cup D = B$ then
2:      return $T$
3:  $\psi = (\bigwedge_{b \in T} b) \wedge (\bigwedge_{b \in B-T} \neg b)$
4:  if $\mathit{check}(\mathrm{VC}(P) \wedge \psi) ==$ CORRECT then
5:      return $T$
6:  if $|B - (T \cup D)| = 1$ then
7:      return $B - D$
8:  $T_1, T_2 := \mathit{partition}(B - (T \cup D))$
9:  $S_1 := \mathit{vcrefine}(T \cup T_2,\ D,\ P)$
10: $S'_1 := S_1 \cap T_1$
11: return $\mathit{vcrefine}(T \cup S'_1,\ D \cup (T_1 - S'_1),\ P)$
```

(b) **Algorithm 2**: *vcrefine*$(T, \emptyset, P)$

Fig. 4. Hierarchical refinement algorithms.

Alg. 1 works as follows. If $T \cup D = G$ (line 1) then we already know that P is correct while tracking T (because the precondition is that P is correct while tracking $G - D$). Lines 3 to 5 check if T is already sufficient. Otherwise, in line 6, we check if only one variable remains undecided, i.e., $|G - (T \cup D)| = 1$, in which case the minimal solution is to include that variable in T (which is the same as returning $G - D$). Lines 8 to 11 form the interesting part of the algorithm. Line 8 splits the set of undecided variables into two equal parts randomly. Because of the check on line 6, we know that each of T_1 and T_2 is non-empty. Next, the idea is to use two separate queries to find the set of variables in T_1 (respectively, T_2) that should be tracked. The first query is made on line 9, which tracks all variables in T_2. The only remaining undecided variables for this query is the set T_1. Thus, the answer S_1 of this query will include $T \cup T_2$ along with the minimal set of variables in T_1 that should be tracked. We capture this in S'_1 and then all variables in $T_1 - S'_1$ should not be tracked. Thus, the second query includes S'_1 in the set of tracked variables and $(T_1 - S'_1)$ in the set of do-not-track variables. The procedure *hrefine* is guaranteed to return a minimal set of variables to track.

Theorem 1. *Given a path P with global variables G, and sets $T, D \subseteq G$, such that T and D are disjoint, suppose that Abstract$(P, G - D)$ is correct. If $R =$ hrefine(T, D, P) then $T \subseteq R \subseteq G - D$, and Abstract$(P, R)$ is correct, while for each set R' such that $T \subseteq R' \subset R$, Abstract(P, R') is buggy.*

Proof of this theorem can be found in our techreport [24]. The following Lemma describes the running time of the algorithm.

Lemma 1. *If the output of Alg. 1 is a set R, then the number of calls to check made by the algorithm is (a) $O(|R - T| \log(|G - T|))$, and (b) bounded above by max$(2|G - T| - 1, 0)$.*

```
if(¬Tracked(v)) {
  assume true;
} else if(¬Tracked(e)) {
  havoc v;
} else {
  v := e;
}
```

(*a*)

```
if(¬Tracked(e)) {
  assume true;
} else {
  assume e;
}
```

(*b*)

```
if(¬Tracked(e)) {
  assert false;
} else {
  assert e;
}
```

(*c*)

Fig. 5. Program transformation for parameterized variable abstraction: (*a*) Transformation for $v := e$; (*b*) assume e; (*c*) assert e. Other statements are left unchanged.

Both Alg. 1 and Storm's refinement share a disadvantage: they spend a significant amount of time outside the theorem prover. Each iteration of Alg. 1 needs to abstract the path with a different set of variables, and then generate the VC for that path. In order to remove this overhead, the next algorithm that we present will require VC generation only once and the refinement loop will be carried out inside the theorem prover.

First, we carry out a *parameterized* variable abstraction of the input path as follows: for each global variable v, we introduce a Boolean constant b_v and carry out the program transformation shown in Fig. 5. The transformed program has the invariant: if b_v is set to *true* then the program behaves as if v is tracked, otherwise it behaves as if v is not tracked. The transformation uses a subroutine *Tracked*, which takes an expression e as input and returns a Boolean formula:

$$\mathit{Tracked}(e) = \bigwedge\nolimits_{v \in \mathit{GlobalVars}(e)} b_v$$

Thus, *Tracked*(e) returns the condition under which e is tracked.

If P_I was the input counterexample and T the current set of tracked variables, then we transform P_I to P using parameterized variable abstraction. Next, we set each b_v, $v \in T$ to *true*. Let B be the set of Boolean constants b_v, $v \notin T$. The refinement question now reduces to: what is the minimum number of Boolean constants in B that must be set to *true* so that P is correct, given that setting all constants in B to *true* makes P correct? We solve this using Alg. 2, which takes P as input.

Alg. 2 is exactly the same as Alg. 1 with the difference that instead of operating at the level of programs and program variables, it operates at the level of formulae and Boolean constants. This buys us a further advantage: the queries made to the theorem prover on line 6 are very similar. One can save VC(P) on the theorem prover stack and only supply ψ for the different queries. This enables the theorem prover to reuse all work done on VC(P) across queries.

4 Evaluation

We have implemented Corral as a verifier for programs written in Boogie. It is supported by a front-end each for compiling C and .NET bytecode to Boogie. The translation from C to Boogie uses several theories for encoding the semantics of C programs. Linear arithmetic is used for modeling pointer arithmetic and a subset of integer operations; uninterpreted functions are used for modeling any other operation not modeled by linear arithmetic; arrays are used to

Name	LOC	Vars	Procs	Conc?	Correct?	Iter	Time (sec)		
							Total	R(%)	S(%)
daytona	660	114	40	Yes	Yes	8	26.9	50	35
daytona_bug2	660	114	40	Yes	No	6	27.0	56	27
kbdclass_read	978	212	48	Yes	Yes	12	194.4	52	29
kbdclass_ioctl	978	212	48	Yes	Yes	6	63.9	43	38
mouclass_read	818	179	44	Yes	Yes	13	185.7	53	28
mouclass_bug3	818	179	44	Yes	No	15	245.5	53	30
ndisprot_write	907	122	46	Yes	Yes	6	24.8	41	44
pcidrv_bug1	661	109	49	No	No	11	37.4	49	37
serial_read	1601	378	77	Yes	Yes	13	1151.7	41	51
mouser_sdv_a	3311	225	131	No	No	4	35.4	33	45
mouser_sdv_b	3898	252	143	No	Yes	12	990.8	15	81
fdc_sdv	5799	421	180	No	Yes	11	659.8	11	85
serial_sdv_a	7373	466	149	No	Yes	6	139.2	47	34
serial_sdv_b	7396	439	168	No	No	6	289.1	46	40

Fig. 6. Running times of Corral on driver benchmarks.

model the heap memory split into multiple maps based on fields and types [10]. The translation of .NET bytecode uses the aforementioned theories in a similar way but also uses two other theories: (1) algebraic datatypes to model object types and delegate values, and (2) generalized array theory [13] to model hashsets and .NET events. It is important to note that Corral is agnostic to the source language used and depends only on the compiled Boogie program.

4.1 Evaluating components of Corral

The first set of experiments show: (1) how the various components of Corral contribute to its overall running time; (2) a comparison of stratified inlining against static inlining; and (3) a comparison of different refinement algorithms. These experiments were conducted on a collection of Windows device drivers in C (compiled to Boogie using Havoc [10]). For each driver, we had various different harnesses that tested different functionalities of the driver. Some of the harnesses were concurrent, in which case the sequential program was obtained using a concurrent-to-sequential source transformation described elsewhere [16, 25]. Such sequentialized programs are often quite complicated (because they simulate a limited amount of concurrency using non-determinism in data) and form a good test bench for Corral. These drivers also had planted bugs denoted by the suffix “bug” in the name of the driver.

A summary of these drivers and Corral’s running time is shown in Fig. 6. We report: the number of non-empty C source lines (LOC), the number of global variables in the generated Boogie file (Vars), the number of procedures (Procs), whether the driver is concurrent (Conc?), whether it has a planted bug or not (Correct), the number of iterations of the refinement loop (Iter), the total running time of Corral (Total), the fraction of time spent in abstraction and refinement (R%) and the fraction of time spent in checking using the stratified inlining algorithm (S%). The refinement algorithm used was Alg. 2. Corral fares reasonably well on these programs. A significant fraction of the time is spent refining, thus, justifying our investment into faster refinement algorithms.

Static Inlining For the above-described run of Corral, we collected all programs that were fed to stratified inlining and ran static inlining on them, i.e., we inlined all procedures (up to the recursion bound) upfront and then fed it to Z3. This did not work very well: static inlining ran out of memory (while creating the VC) on each of the six largest programs, with a recursion bound of 1 or 2. Even when the VC did fit in memory, Z3 timed out (after 1 hour) in many cases. For the rest, stratified inlining was still three times faster. A more detailed comparison can be found in our techreport [24].

Hierarchical Refinement For the above-described run of Corral, we collected all the spurious counterexamples and ran each of the three refinement algorithms on them: Storm's refinement algorithm, Alg. 1, and Alg. 2. All three algorithms returned the same answer on each counterexample. Alg. 2 was a clear winner with very little variation in its running time. The average speedup of Alg. 2 over Alg. 1 was 2.8X and that over Storm was 13.2X. A more detailed comparison can be found in our techreport [24].

4.2 SDV benchmarks

For our next experiment, we compare Corral against state-of-the-art verification tools Slam [2] and Yogi [29]. Both these tools are run routinely by the Static Driver Verifier (SDV) team against a comprehensive regression suite consisting of multiple (real) drivers and properties. The suite consists of a total of 2274 driver-property pairs, of which 1886 are correct (i.e., the property is satisfied by the driver) and 388 are incorrect (i.e., there is an execution of the driver that violates the property). We compare Corral against Yogi. (The comparison against Slam is similar in nature.) We compare the running times of the tools as well as evaluate the effectiveness of recursion bounding. We will use the term program to refer to a driver instrumented with a particular property.

First, note that Yogi does full verification unlike Corral that requires a recursion bound to cut-off search. Thus, Yogi has to do more work than Corral for correct programs. Moreover, both these tools use slightly different modeling of C semantics, leading to different answers for a few programs. It was difficult to remove these differences.

We ran Corral with a recursion bound of 2. The natural question to ask is: how many bugs did Corral miss? It turns out that only on 9 programs Yogi found a bug, but Corral was not able to find one (out of a total of 388 buggy programs). We investigated these 9 programs and they turned out to be instances of just two loops. The first loop required a recursion bound of 3, and Corral was able to find the bugs with this bound. The second loop was of the form $\textbf{for}(i = 0; i < 27; i++)$, and thus, required a bound of 27 for Corral

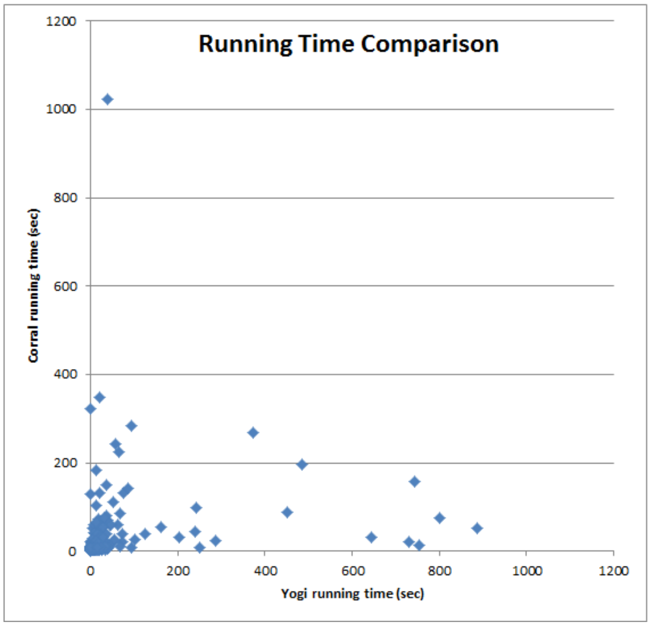

Fig. 7. A comparison of running times of CORRAL against YOGI on buggy programs.

to explore the code that came after this loop. We are currently investigating techniques to deal with such constant-bounded loops.

Aggregating over all programs on which YOGI and CORRAL returned the same answer, YOGI took 55K seconds to produce an answer for all of them, whereas CORRAL required only 27K seconds, a speedup of about 2X. The scatter plot comparing the running times of YOGI and CORRAL on buggy programs is shown in Fig. 7. Both tools took around the same amount of total time on these programs (about 10K seconds). However, the total running time is dominated by many programs with a trivial running time, on which CORRAL is slightly slower. The distribution of points in the scatter plot show a more interesting trend. For instance, there is a larger distribution of points around the X-axis, meaning that CORRAL did very well on programs on which YOGI was having a hard time. Aggregating over buggy instances on which YOGI took at least a minute, CORRAL was twice as fast as YOGI. There is also a smaller distribution of points around the Y-axis on which YOGI did much better. Manual inspection of these programs revealed that the main reason for YOGI's success was summarization of procedures using predicate abstraction. The next subsection equips CORRAL with a simple summarization routine.

We tried using other bounded-model checkers (CBMC [8] and ESBMC [11]) on these programs but they did not perform well. They either ran out of time or memory on most programs, possibly because they use static inlining to deal with

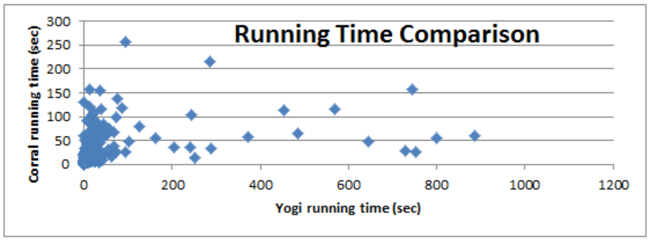

Fig. 8. A comparison of running times of CORRAL augmented with HOUDINI against YOGI on buggy programs.

procedures. Moreover, both SLAM and YOGI do not work well on "sequentialized" versions of concurrent programs (i.e., they would often time out). On the other hand, CORRAL is able to uniformly work on all these programs.

In conclusion, this experiment shows: (1) the practical applicability of recursion bounding in practice for real-world bug hunting, and (2) that CORRAL is competitive with state-of-the-art software model checkers.

Using Summaries We now demonstrate the ability of CORRAL to leverage abstraction techniques by using HOUDINI [17]. HOUDINI is a scalable modular analysis for finding the largest set of inductive invariants from amongst a user-supplied pool of candidate invariants. HOUDINI works as a fixpoint procedure; starting with the entire set of candidate invariants, it tries to prove that the current candidate set is inductive. The candidates that cannot be proved are dropped and the procedure is repeated until a fixpoint is reached. Our usage of HOUDINI is restricted to inferring post-conditions of procedures, where a post-condition is simply a predicate on the input and output vocabulary of a procedure. The inferred post-conditions act as procedure summaries.

The drivers in our suite do not have recursion (but may have loops). In this setting, if we generate n candidate summaries per procedure, and there are P procedures, then HOUDINI requires at most $O(nP)$ number of theorem prover queries, where each query has the VC of at most one procedure only.

For SDV, we generated candidates using two different sources of predicates: first, we used YOGI's internal heuristics, which compute an initial set of predicates for YOGI's iterative refinement loop; and second, we manually inspected some properties and wrote down predicates that captured the *typestate* of the property. Because we never looked at the drivers themselves, this process required minimal manual effort. We now briefly illustrate this process.

A driver goes through two instrumentation passes that are important to understand for generating HOUDINI candidates. The first pass instruments the driver with a property. For illustration, consider a property which asserts that for a given lock, acquire and release operations must occur in strict alternation. A driver is instrumented with this property by introducing a new variable s of type `int` that is initialized to 0 in the beginning of `main`. A lock acquire operation first asserts that $s == 0$ and then sets s to 1. A lock release operation first asserts that $s == 1$ and then sets s to 0. It is easy to see that if the instrumented driver does not fail any assertion then the acquire and release operations happen in strict alternation.

The second instrumentation is carried out internally inside CORRAL as a preprocessing step. Because the stratified inlining algorithm only checks for a condition at the end of `main`, CORRAL introduces a new Boolean variable `error`, initialized to *false*, and replaces each assertion `assert` e with:

$$\texttt{error} := \neg e; \;\; \textbf{if}(\texttt{error}) \;\; \textbf{return};$$

And after each procedure call, CORRAL inserts:

$$\textbf{if}(\texttt{error}) \;\; \textbf{return};$$

Then CORRAL simply asserts that `error` is *false* at the end of `main`.

HOUDINI candidates are derived from predicates that capture the property typestate. For the lock acquire-release property, the typestate predicates are $s == 0$ and $s == 1$. These are used to generate the following 6 candidates to capture either (a) how the typestate can be modified by a procedure; or (b) conditions under which `error` is not set:

$$\begin{array}{ll} old(s) == 0 \Rightarrow s == 0 & old(s) == 1 \Rightarrow s == 0 \\ old(s) == 0 \Rightarrow s == 1 & old(s) == 1 \Rightarrow s == 1 \\ old(s) == 0 \Rightarrow \neg\texttt{error} & old(s) == 1 \Rightarrow \neg\texttt{error} \end{array}$$

Here, $old(s)$ refers to the value of s at the beginning of a procedure and s refers to its value at the end of a procedure.

The generated candidates are fed to HOUDINI and valid ones form procedure summaries. These summaries not only help proving correctness of certain programs, but also help in finding bugs faster: when a part of a program is proved correct, no further inlining is required in that region. A scatter plot on the buggy instances is shown in Fig. 8. The running time of HOUDINI is included in the running time of CORRAL. For simple instances, this adds extra overhead: the distribution of points around the origin are in favor of YOGI. However, as running time increases, CORRAL + HOUDINI is almost always faster. On programs with non-trivial running times, CORRAL + HOUDINI was six times faster than YOGI (up from 2X without HOUDINI).

Computing Proofs CORRAL can prove correctness regardless of the recursion bound when the over-approximation used in stratified inlining is UNSAT. Surprisingly, this simple over-approximation (along with partial inlining) suffices in many cases. Of the 1886 correct programs, CORRAL was actually able to prove 1574 of them correct irrespective of the recursion bound: a proof rate of 83%. With the use of HOUDINI, this number goes up to 1715: a proof rate of 91%.

4.3 SV-COMP benchmarks

For the next experiment, we looked for external sources of benchmarks that are already accessible to other tools. We found a rich source of such benchmarks from the recently-held competition on software verification [6]. Unfortunately, the benchmarks are CIL-processed C files, most of which do not compile using our (Windows-based) front end. Thus, we picked all programs in only two categories and manually fixed their syntax. The first category consists of 36 programs in the `ssh` folder and the second category consists of 9 programs in the `ntdrivers` folder. Roughly half of these programs are correct, and half are buggy. We ran CORRAL with a recursion bound of 10, which is the same used by other bounded model checkers in the competition. We did not attempt to set up the tools that participated in the competition on our machine. Instead, we compare CORRAL against the running times reported online. Although it is unfair to compare running times on different machines (and operating systems), we only show the numbers to illustrate CORRAL's robustness, and a ball-park estimate of how it competes against many tools on a new set of benchmarks.

For each program, CORRAL returned the right answer well within the time limit of 900 seconds. In the `ssh` category, CORRAL takes a total of 168 seconds to finish. This is in comparison to 525.8 seconds taken by the best tool in the category (CPACHECKER). In the `ntdrivers` category, CORRAL took a total of 447.5 seconds, but in this case, the best tool (again CPACHECKER) performed better—it took only 105.2 seconds. No tool other than CORRAL and CPACHECKER was able to return right answers on all programs in `ntdrivers` (they either ran out of time or memory or returned an incorrect answer).

4.4 .NET benchmarks

We downloaded an open-source .NET implementation of the Tetris game and set it up with a harness that clicks random buttons and menus to drive the game. We compiled this program to BOOGIE using BCT [5]. The original program is around 1550 lines of source code (excluding type definitions and comments); it compiles to roughly 23K lines of BOOGIE code and has 357 procedures. We created a collection of 570 queries, one for each target of each branch in the program as well as the beginning of each procedure. We ran CORRAL with a recursion bound of 1. Within a budget of 600 seconds per query, CORRAL was able to resolve 243 queries as reachable, 204 as unreachable, and the rest (123) timed out. These results, together with results described earlier on C programs, demonstrate CORRAL's robustness in dealing with queries from different programming languages.

5 Related Work

Stratified inlining is most closely related to previous work on *structural abstraction* [1], *inertial refinement* [31] and *scope bounding* [27, 21]. However, structural abstraction is based entirely on overapproximations; it does not have the analog of underapproximating by blocking certain calls. Inertial refinement uses both over and under approximations to iteratively build a view of the program that is a collection of regions and uses the notion of minimal correcting sets [26] for the refinement. The scheme in stratified inlining appears to be much simpler because, first, it abstracts only calls and not arbitrary program regions. Second, the refinement is based on a simple analysis of the counterexample from the over-approximate query. Scope bounding refers to limiting the scope of an analysis to a program fragment and has been used previously, for instance, in verifying null-dereference safety of Java programs [27]. In their context, fragments need not contain the entry procedure and may grow backwards (i.e., callers get inlined, instead of callees), which is in contrast to stratified inlining. Moreover, the process of growing fragments is not based on abstract counterexamples. Additionally, none of these previous techniques attempt to perform variable abstraction, whereas Corral effectively orchestrates it along with stratified inlining.

The idea of introducing Boolean variables for doing optimization inside a theorem prover, which we use in Alg. 2, has been used previously, for instance in error localization [22].

A variety of methods have been proposed for software verification based on predicate abstraction and interpolation [2, 18, 29, 28]. The focus of these methods is to infer predicates in order to create a finite vocabulary for invariants and summaries. These techniques are complementary to our work; summaries generated by them could be used to speed up Corral as shown by our experiments with Houdini.

Bounded model checkers such as CBMC [8], ESBMC [11], and LAV [32] perform bounded program verification similar to Corral. However, their focus is on efficient VC generation and modeling of program semantics; they use a technique similar to static inlining and do not use variable abstraction.

References

1. D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *ICSE*, pages 211–220, 2008.
2. T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.
4. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87, 2005.
5. M. Barnett and S. Qadeer. BCT: A translator from MSIL to Boogie. Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation, 2012.
6. D. Beyer, editor. *1st International Competition on Software Verification, co-located with TACAS 2012, Tallinn, Estonia*, 2012.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
8. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.
9. E. M. Clarke, R. P. Kurshan, and H. Veith. The localization reduction and counterexample-guided abstraction refinement. In *Essays in Memory of Amir Pnueli*, pages 61–71, 2010.

10. J. Condit, B. Hackett, S. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages*, 2009.
11. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 2011.
12. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
13. L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52, 2009.
14. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
15. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.
16. M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *Principles of Programming Languages*, 2011.
17. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, pages 500–517, 2001.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, 2002.
19. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.
20. J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1999.
21. F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In *ASE*, 2011.
22. M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, 2011.
23. S. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Computer Aided Verification*, 2009.
24. A. Lal, S. Qadeer, and S. Lahiri. Corral: A solver for reachability modulo theories. Technical Report MSR-TR-2012-09, Microsoft Research, 2012.
25. A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1), 2009.
26. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.
27. A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA*, 2008.
28. K. L. McMillan. Lazy annotation for program testing and verification. In *CAV*, pages 104–118, 2010.
29. A. V. Nori and S. K. Rajamani. An empirical study of optimizations in YOGI. In *ICSE*, pages 355–364, 2010.
30. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.
31. N. Sinha. Modular bug detection with inertial refinement. In *FMCAD*, 2010.
32. M. Vujošević-Janičić and V. Kuncak. Development and evaluation of LAV: an SMT-based error finding platform. In *VSTTE*, 2012.