Assertion Checking using Dynamic Inference

Anand Yeolekar, Divyesh Unadkat

No Institute Given

Abstract. We present a technique for checking assertions in code that combines model checking and dynamic analysis. Our technique first constructs an abstraction by summarizing code fragments in the form of pre and post conditions. Spurious counterexamples are then analyzed by Daikon, a dynamic analysis engine, to infer invariants over the fragments. These invariants, representing a set of traces, are used to partition the summary with one partition consisting of the observed spurious behaviour. Partitioning summaries in this manner increases precision of the abstraction and accelerates the refinement loop. Our technique is sound and compositional, allowing us to scale model checking engines to larger code size, as seen from the experiments.

Keywords: Verification; Model Checking; Dynamic Inference; Scalability.

1 Introduction

Embedded software that is classified safety- or business-critical needs to be rigorously analyzed for bugs before deployment. Over the years, many dynamic and static approaches have been proposed to address this problem. Dynamic approaches include various techniques for testing from manual to fully automated [1, 2]. Static analysis techniques include abstract interpretation and model checking. Dynamic approaches scale to large-size code and may report test cases leading to bugs, but are of little help in proving their absence, as stated by Dijkstra, so crucial for safety-critical software. Model checking is thus preferred, having the advantage of obtaining traces that developers can use for diagnostics. Unfortunately, model checking algorithms run into the state space explosion problem analyzing even moderately-sized code. Even with SAT and SMT solvers scaling over the years [3] and abstraction techniques in place, model checking has not scaled to the level of static analysis tools [4, 5].

From a software developer's perspective, analysis techniques that are sound (proof of absence of bugs) and precise (no false positives) are useful. With this in mind, we present a CEGAR-based [6, 7] technique that combines dynamic and static approaches to scale assertion checking to large-size code. Figure 1 outlines our approach. We first decompose the code structurally into its functions, called units, based on the location of the assertion to be checked. These units are summarized in the form of pre- and post-conditions. We choose an unconstrained pre and post to form the initial sound abstraction of the units. The given assertion is checked over the abstracted code. Following the abstraction-refinement

paradigm, spurious counterexamples are used to refine the summary. We propose to refine the summary using dynamic analysis over the spurious traces to infer *likely* invariants, that can be used to partition the state space represented by the summary. The state space is partitioned into two parts such that, the spurious traces represented by the invariants form one part, with the remaining state space forming the second.

Invariants inferred from program traces can reveal useful information about the program behaviour, often complementing static approaches [8, 9]. Dynamic analysis, however, may infer unsound invariants at program locations. We propose to overcome this limitation by iteratively verifying the inferred pre-post pair over the units. The precondition invariants are cast as *assumes* and the post as *assertions* to be checked over the unit. Note that running a model checker at this level usually scales much better than at the application level. Violated invariants in the postconditions are dropped and counterexamples reported in the process are used to improve Daikon's inference. This pre-post pair is used to partition the summary, maintaining soundness of the resulting abstraction. Iterative partitioning improves the precision of the summary and accelerates the refinement process.

We believe that our approach utilizing dynamic inference can yield stronger program properties with fewer refinements, in contrast to static approaches for abstraction refinement, such as predicate-refinement [6] and specification-refinement [10]. Our approach can also be viewed as orthogonal to predicate refinement - the abstraction constructed by our technique in any refinement iteration can be analyzed by an off-the-shelf predicate-refinement tool. Our approach is fully compositional and takes advantage of modularity, if any, in the program structure.

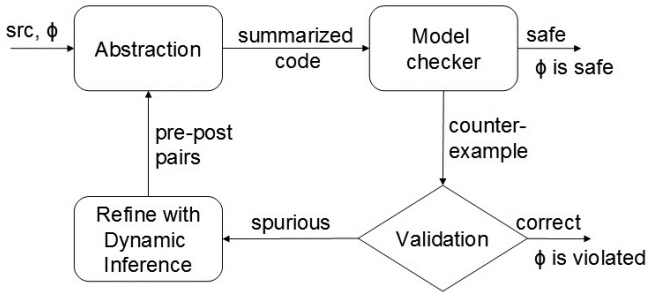

Fig. 1. Outline of our approach

The contributions of this work are as follows:

- A compositional approach to scale model checking using dynamic inference
- A fully automated tool that demonstrates the approach
- Case studies showing scalability of our approach on benchmark code

The rest of the paper is organized as follows. In section 2, we define the summarization and refinement of code fragments, present an algorithm for assertion checking, and address theoretical issues. Section 3 explains the implementation of our algorithm in a tool called DIV. In section 4 we describe experiments conducted with DIV, SatAbs and CBMC. We discuss work related to our approach in section 5 and conclude in section 6.

2 Compositional assertion checking

In this section, we present the approach of computing and refining summaries of code fragments using dynamic inference. Our approach is inspired by the work of scaling test generation presented in [11] and overcomes the limitation of unsound summaries, making it suitable for verification.

2.1 Summarizing functions

Daikon [12] implements dynamic detection of *likely* invariants. Daikon analyzes data values from program traces to match a fixed template set of invariants. Invariant templates include constants $x = k$, non-zero $x \neq 0$, intervals $a \leq x \leq b$, linear relationships $y = ax + b$, ordering $x \leq y$, containment $x \in y$, sortedness, size of arrays $size[arr] \leq n$ and so on. Each invariant inferred is an (linear) expression instantiated on program variables, and can be translated to statements in C language. Invariants are reported at program points such as function entry and exit, expressing the function's preconditions and postconditions, respectively.

Daikon's inference. An invariant inferred by Daikon holds for traces *seen* so far. As trace data accumulates, new invariants can be reported as well as, some invariants can be dropped, due to the nature of the machine-learning algorithms of Daikon. With more trace data, crossing the *confidentiality threshold*[1] enables reporting of more invariants, and those violated are withdrawn. In addition, Daikon implements analysis to suppress redundant invariants, filter out "obvious" ones and limit the number of instantiations from templates. As with any other dynamic analyses, Daikon's inference is unsound and incomplete. We next explain how to generate sound summaries of functions as abstraction units, with the inferred invariants as the starting point.

Function summary. Let S be a program containing assertion ϕ to be checked. Let f be a candidate function to be summarized, such that the location of ϕ is not reachable from entry location of f. Let $pre, post$ be the set of pre and postcondition invariants inferred by Daikon over a set of executions of f. The invariants in pre and $post$ define a state space of S. The summary of f is defined as a set $\hat{f} = \{\langle pre_0, post_0\rangle, ..., \langle pre_k, post_k\rangle\}$, with the restriction

[1] Minimum number of samples for an invariant to hold.

that (i) the preconditions are disjoint that is, $pre_i \wedge pre_j = \Phi$, (ii) each pre-post pair forms a Hoare triple $\{pre_i\} f \{post_i\}$, and (iii) the union of preconditions over-approximates the set of states of S reachable at all call points of f in S. The pre-post pairs of the summary partition the function's input-output space. The summary permits a non-deterministic mapping of input points to output points within the state space defined by a pre-post pair.

```
int compare(int a,int b) {
1. if sign(a)==sign(b)
2.   if abs(a)>abs(b) return a;
3.   else return b;
4. if a>b return a;
5. else return b;  }
int main() {
6. unsigned int x=nondet;
7. assert(compare(x,-x)==x);
}
```

(a)

```
int compare'(int a,int b) {
1. int ret;
2. ret=nondet;
3. return ret;
}
int main() {
4. unsigned int x=nondet;
5. assert(compare(x,-x)==x);
}
```

(b)

Fig. 2. Example code and the summarized version

Example 1. Consider the example code in figure 2(a), containing an assertion in `main` to be checked. `compare` returns the smaller of `a,b` when both are negative, otherwise the larger. Figure 2(b) illustrates a trivial summarization of `compare` obtained with $\{\langle true, true\rangle\}$ as the summary that is, unconstrained pre and postconditions. Newly introduced variable `ret` indicates the return value of the function.

Abstraction. A *trace* is a finite sequence $(..., \langle loc, s\rangle, ...)$ where s is the program state at location loc in the source code. Let $t = (\langle l_0, s_0\rangle, \langle l_1, s_1\rangle, ..., \langle l_k, s_k\rangle)$ and $\hat{t} = (\langle l_0, \hat{s_0}\rangle, \langle l_k, \hat{s_1}\rangle)$ be traces of f and $\hat{f}$, respectively. For sequential code, the traces are equivalent $t \approx \hat{t}$ if $s_0 = \hat{s_0}$ and $s_k = \hat{s_k}$ i.e. the input-output mapping matches. Since the summary allows non-deterministic input-output behaviour, the set of traces of $\hat{f}$, denoted as $\|\hat{f}\|$, over-approximates the set of traces of f i.e., $\|\hat{f}\| \supseteq \|f\|$. Let $\hat{S}$ denote the program obtained by replacing f with $\hat{f}$ in S. Then it follows that $\|\hat{S}\| \supseteq \|S\|$ that is, $\hat{S}$ is an abstraction of S. In figure 2, `compare'` is an abstraction of `compare`.

2.2 Summary refinement

The summary abstracts the function's computation but allows spurious behaviours. A model checker analysing code containing summarized functions may return spurious counterexamples, necessitating summary refinement. A summary $\hat{f}_2$ refines $\hat{f}_1$, denoted $\hat{f}_2 \prec \hat{f}_1$, if $\|\hat{f}_2\| \subseteq \|\hat{f}_1\|$. Correspondingly, $\hat{S}_2$ is a refinement of $\hat{S}_1$ and $\|\hat{S}_2\| \subseteq \|\hat{S}_1\|$.

We propose to refine summaries by using Daikon to infer new invariants from the spurious counterexamples that can strengthen the abstraction. The central idea of the refinement scheme is to partition the state space represented by the pre-post pairs into two parts such that, the spurious traces form the first part, with the remaining state space forming the second.

Let $\langle pre_i, post_i \rangle \in \hat{f}$ contain the spurious input-output mappings $(i_1, \hat{o_1}), ..., (i_n, \hat{o_n})$, extracted from the counterexample obtained on $\hat{S}$ while checking ϕ. Let p_1, q_1 be the pre and postcondition invariants inferred by Daikon on executing f with inputs $i_1, ..., i_n$. We use p_1, q_1 to partition $\langle pre_i, post_i \rangle$, with the first part as $\langle pre_i \wedge p_1, post_i \wedge q_1 \rangle$.

When refining summaries with dynamic analysis, the problem is obtaining sound pre-post pairs from the possibly unsound invariants. We overcome this by using a model checker to iteratively verify $\{pre_i \wedge p_1\} f \{post_i \wedge q_1\}$, by casting the preconditions as *assume* and postconditions as *assert* statements in the language of the model checker. Violated invariants are dropped from the postcondition and counterexamples returned are added to the set of executions to improve the invariant inference, until the pre-post pair is verified by the model checker. Note that the model checker is applied at unit level, where scalability does not pose a problem.

We obtain the pre-post pair of second part in the following manner. For the precondition, we complement the pre of the first, giving $p_2 = \neg p_1$. To obtain the corresponding postcondition q_2, constraints in $p_2 \wedge pre_i$ are used to synthesize inputs to execute f and infer invariants. Following the process above, we obtain the sound pre-post pair $\langle pre_i \wedge p_2, post_i \wedge q_2 \rangle$. These two pre-post pairs replace $\langle pre_i, post_i \rangle$ in $\hat{f}$ to give $\hat{f}'$. Note that $\hat{f}'$ meets all the requirements of a summary as defined above namely, preconditions being disjoint, soundness of all pre-post pairs and preconditions over-approximating state space at call points of f.

To show $\hat{f}' \prec \hat{f}$, consider a trace $(\langle l_0, i \rangle, \langle l_k, o \rangle)$ such that $i \rhd pre_i \wedge p_1, o \rhd post_i \wedge \neg q_1$, where $\rhd$ denotes the point lies in the state space defined by the invariants. Such a trace, where the input lies in the first part and the output in the other, exists in $\hat{f}$ but is disallowed in $\hat{f}'$. Further, every trace in $\hat{f}'$ exists in $\hat{f}$ as $pre_i \wedge p_1 \Rightarrow pre_i, post_i \wedge q_1 \Rightarrow post_i$, etc. Clearly, $\|\hat{f}'\| \subseteq \|\hat{f}\|$, establishing refinement. In algorithm 1, we explain how to refine summaries when inferred invariants are too weak to partition, and how to block the spurious mappings $(i_k, \hat{o_k})$ from reappearing in $\hat{f}'$.

```
pre:            pre:
 a>b             a>b
post:           post:
 ret==a          ret==a or ret==b
    (a)             (b)
```

Fig. 3. Generating a sound pre-post pair

Example 2. Consider our running example `compare'` from figure 2(b). Assume that the model checker returned a counterexample (spurious) violating the assertion, with assignments `a=4,b=-4` at entry of `compare'` and `ret=7` at the exit. Figure 3(a) shows invariants, `a>b,ret==a`, inferred by Daikon after analyzing the execution of `compare`. On verifying the pre-post invariants with a model checker, we obtain a counterexample violating `ret==a`. On running Daikon again with the newly added counterexample, we obtain invariants shown in figure 3(b). Subsequently, the model checker reports safety of this pre-post pair over `compare`. The second pre-post pair is obtained as explained above, completing the partitioning of the parent summary $\{\langle true, true\rangle\}$. Figure 4(a) illustrates the refinement of `compare'` to `compare''`.

```
int compare''(int a,int b) {
1. int pid=nondet,ret;
2. asssume(pid>=0 && pid<2);
3  if (pid==0)
     assume(a>b);
4. else
     assume(!(a>b));

5. ret=nondet;
6. if (pid==0)
     assume(ret==a||ret==b);
7. else
     assume(ret==a||ret==b);
8. return ret;
}
int main() {
9.  unsigned int x=nondet;
10. assert(compare(x,-x)==x);
}
```

(a)

```
int compare'''(int a,int b) {
1. int pid=nondet,ret;
2. asssume(pid>=0 && pid<3);
3  if (pid==0) assume(a>b && a>0);
4. else if (pid==1)
     assume(!(a>0) && (a>b));
5. else assume(!(a>b));
6. ret=nondet;
7. if (pid==0)
     assume(ret==a&&(ret==a||ret==b));
8. else if (pid==1)
     assume(ret==b&&(ret==a||ret==b));
9. else assume(ret==a||ret==b);
10. return ret;
}
int main() {
11. unsigned int x=nondet;
12. assert(compare(x,-x)==x);
}
```

(b)

Fig. 4. Refining a function summary

Example 3. Assume that model checking the code of figure 4(a) returned a spurious counterexample with `a=2,b=-2` at the entry and `ret=-2` at the exit of `compare''`. The final refinement is shown in figure 4(b), with $\langle a > b, ret = a \vee ret = b\rangle$ getting refined to $\{\langle a > 0 \wedge a > b, ret = a \wedge (ret = a \vee ret = b)\rangle, \langle \neg(a > 0) \wedge a > b, ret = b \wedge (ret = a \vee ret = b)\rangle\}$. The model checker reports safety of the assertion after analyzing `compare'''`.

2.3 Algorithm for assertion checking

Algorithm 1 Assertion checking

1: $check(S, \phi) =$
2: $\hat{f} = \{\langle true, true\rangle\}$ ▷ `initial summary`
3: **while** true **do**
4: $ce = modelcheck(\hat{S}, \phi)$
5: **if** $ce = null$ or $ce = valid$ **then**
6: terminate
7: **end if**
8: **for all** $\langle p_k, q_k\rangle \in \hat{f} \mid k \in pid(ce, \hat{f})$ **do** ▷ `partition the parent`
9: $testdb = inputs(ce, \hat{f}, k)$
10: **repeat** ▷ `first child pre-post pair`
11: $\langle pre, post\rangle = daikon(f, testdb)$
12: $\langle p_{k1}, q_{k1}\rangle = \langle pre \wedge p_k, post \wedge q_k\rangle$
13: $ce' = modelcheck(\{p_{k1}\}f\{q_{k1}\})$
14: $testdb = testdb \cup ce'$
15: **until** ce' *is null*
16: $p_{k2} = \neg p_{k1} \wedge p_k$
17: $testdb = synthinputs(f, p_{k2})$
18: **repeat** ▷ `second child pre-post pair`
19: $\langle dummy, post\rangle = daikon(f, testdb)$
20: $q_{k2} = post \wedge q_k$
21: $ce' = modelcheck(\{p_{k2}\}f\{q_{k2}\})$
22: $testdb = testdb \cup ce'$
23: **until** ce' *is null*
24: **for all** $\langle i, \hat{o}\rangle \in iopairs(ce, \hat{f}, k)$ **do** ▷ `process point pre-post pairs`
25: **if** $\hat{o} \rhd q_{k1}$ **then**
26: $o = simulate(f, i)$
27: $add(\hat{f}, \langle i, o\rangle)$
28: **end if**
29: **end for**
30: $remove(\hat{f}, \langle p_k, q_k\rangle)$ ▷ `replace the parent with children`
31: $add(\hat{f}, (\langle p_{k1}, q_{k1}\rangle, \langle p_{k2}, q_{k2}\rangle))$
32: **end for**
33: **end while**

Algorithm 1 presents our approach for checking assertion ϕ using dynamically computed function summaries. For simplicity, we present the case with only one function f summarized in S.

The initial abstraction $\hat{f}$ for f is chosen as the unconstrained pre-post pair, $\{\langle true, true\rangle\}$ (line 2). By definition, the summary over-approximates the set of states reachable in S at all call sites of f. To compute this set is as hard as checking the assertion itself, so we choose *true* as an over-approximation of this set.

The algorithm implements CEGAR [7] with dynamic inference in the loop, beginning line 3. We obtain $\hat{S}$ by replacing f with $\hat{f}$ in S. The summarized program is passed to the model checker, which may return a counterexample trace (line 4). The algorithm terminates if the model checker reports assertion safety or the counterexample violates ϕ in S (lines 5-7).

When the counterexample trace violating the assertion turns out to be spurious, we proceed to refining the summary (lines 8-33). We assume the availability of trace processing operators `pid, inputs, iopairs` that extract the values of $\hat{f}$'s pre-post identifiers, inputs and corresponding output variables respectively, from the trace when supplied with relevant arguments.

A trace is spurious due to incorrect input-output mapping within the pre-post pairs of the summary chosen by the model checker. Refinement proceeds by identifying such pre-post in $\hat{f}$ along the trace (line 8). The central idea is to increase precision of the abstraction by partitioning the pre-post pairs such that, the spurious traces form the first part, with the remaining state space forming the second. In the process, we also eliminate spurious mappings.

We collect inputs of f, extracted from the spurious trace, belonging to the pre-post pair (line 9). The algorithm executes f (at unit-level) with these inputs and invokes Daikon to infer invariants (line 11). These invariants are used to partition the state space represented by the pre-post pair (line 12). The model checker is invoked at unit level (line 13) to check whether the postcondition invariants hold over f, given the preconditions. Counterexamples reported, if any, are added to the set of executions (line 14) to repeat the process, improving Daikon's inference, yielding a sound pre-post pair (lines 10-14).

The negated precondition of the first part is used to build the precondition of the second part (line 16), thus maintaining the correctness condition of the summary. `synthinputs` synthesizes values to input variables of f (line 17) from the precondition constraints, using an off-the-shelf constraint solver. Lines 18-23 repeat above steps to obtain the corresponding postcondition, completing the second part. The difference in loops on lines 10-15 and 18-23 is that in the former, Daikon is used to infer both pre and post, while in the latter, only post is inferred (pre being available by negating pre obtained earlier).

Due to the nature of Daikon's inferencing mechanism and loops 10-15 and 18-23 weakening the postconditions, we cannot guarantee that partitioning pre-post pairs will always eliminate the spurious input-output mapping in the *ce*. Lines 24-29 check this and create *point pre-post pairs* to eliminate spuriousness. For a spurious input-output pair $(i, \hat{o})$ retained in the newly created pre-post pair

(line 25), `simulate` executes f to discover the correct input-output mapping (i, o) (line 26) and appends to the summary (line 27), blocking a family of spurious mappings $(i, *)$ from reappearing in subsequent counterexamples. The newly created pre-post pairs replace the parent pre-post, refining the summary (lines 30-31).

2.4 Remarks

We discuss some properties of algorithm 1.

Soundness. $\hat{S}$ over-approximates the set of traces of S at any stage of refinement. Thus if ϕ is not violated in $\hat{S}$, then ϕ is safe in S, subject to the bound supplied to the model checker.

Progress. To guarantee progress, we need to ensure that spurious counterexamples are eliminated. As discussed earlier, spurious mappings from the *ce* retained within pre-post pairs are eliminated by inserting the corrected input-output mappings. This blocks the spurious system-level counterexample in subsequent model checking runs, ensuring progress.

To ensure faster convergence, refinement should significantly improve the precision of the abstraction. Summary refinement partitions the state space such that known spurious behaviours are separated out from the unknown ones. The partitioning crucially depends on Daikon (lines 11,19) to form new pre-post pairs. We depend on the conjunction $pre \wedge p_k$ to partition the parent. Precondition invariants inferred by Daikon with this *testdb* may turn out to be weaker than parent's pre that is, $pre \Rightarrow p_k$, which means the parent pre-post cannot be partitioned. This is still not a problem if Daikon is able to infer stronger postcondition invariants which are subsequently verified as sound by the modelchecker. In this case, the parent pre-post is refined through postcondition strengthening without getting partitioned.

The worst case occurs when both pre and postcondition invariants inferred are weaker than parent's pre-post, or the $repeat-until$ loop weakens the post. In this case, appending point pre-post pairs to the parent is the only refinement. In practice, this situation was observed only rarely, and did not impact scalability.

Termination. When algorithm 1 terminates, we have a (validated) counterexample or the assertion is proven safe. Every loop in the algorithm is terminating, as we use a bounded model checker and Daikon has a finite template set of invariants. In the worst case, the algorithm keeps on accumulating point partitions for each spurious counterexample.

Compositionality. Our choice of $\{true\}$ as the unconstrained initial precondition of summaries allows to decouple the units from the rest of the system. Summaries are computed and refined at unit-level, independent of others, making our approach fully compositional. Assertion ϕ is checked at system-level that is, when the summaries are composed to form $\hat{S}$.

An implication of our choice is that the summary can become too abstract and the model checker may be employed in checking behaviours that are infeasible in the context of S. Our experimental results indicate that this has not impacted the performance or scalability of our approach.

3 Implementation

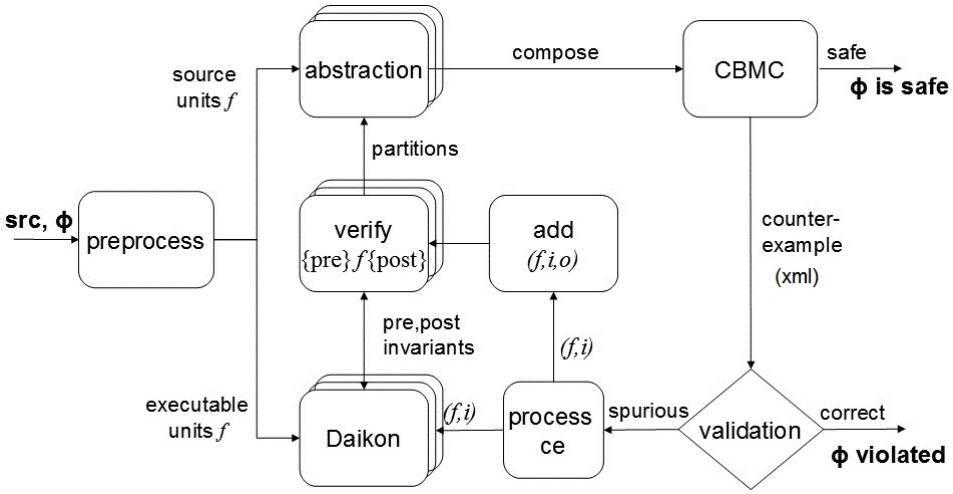

Fig. 5. Tool architecture

Figure 5 depicts our technique for checking assertions in C code. We use Daikon[13] for invariant generation over C code and CBMC [14] for model checking. The *preprocessor* is built using our internally developed tool suite PRISM consisting of a front-end for parsing C code into an intermediate representation, program analysis framework and an unparser. It constructs the function call graph, identifies functions for abstraction, identifies input-output variables of these functions and generates a driver for execution. The functions chosen for abstraction are the roots of subtrees of those functions *not* on the (statically computed) program call stack, when control reaches the location of the user-provided assertion on any path. The *abstraction* routine maintains a list of pre-post pairs for the abstracted functions. The *validation* routine replays counterexamples returned by CBMC on the original code to check for assertion violations. The *ce processing* routine extracts (abstracted) function partition ids, inputs and outputs from the XML trace. We use Kvasir, a tool from the Daikon toolsuite that instruments C code and generate traces for Daikon's consumption. Invariants inferred by Daikon are filtered, translated to CBMC *assume* and *assert* statements, and unparsed as described in [11]. The *verify* routine invokes

CBMC to check $\{pre\}f\{post\}$; counterexamples if any, are fed back to Daikon to improve invariants. The *add* subroutine appends point pre-post pairs to the summary after executing the function. The routine *synthinputs* (not shown in the figure) synthesizes inputs for execution and inference, by using an off-the-shelf constraint solver over the negated preconditions generated by *verify*. The tool has been implemented in Java in about 3KLoC under Linux.

4 Experiments

This section furnishes details about the experimental setup, tool parameters, observations and results from the case-studies.

4.1 Setup and tool parameters

Table 1 lists the casestudies and results of experiments. For evaluating our strategy, we used benchmark programs (column 1) ranging from a simple program from [15] (row 1) to increasingly complex ones from Kratos [16] (rows 2-15). Columns 2 and 3 list the number of functions and lines of code, respectively. The code was seeded with assertions on line numbers given in column 4 (LoA - line of assert statement). Some of these assertions were ones which could not be checked in [11] due to either timeouts or algorithm unsoundness. Due to runtime termination issues, Kratos sources were modified to exit the loops in functions *eval* and *start_simulation* after maximum 10 iterations. The preprocessing phase in our tool removed hard-coded variable initializations and assigned non-deterministic values to these global variables, converting them to program inputs. Some programs contained a non-deterministic initialization to local variables; this was replaced with initialization to a constant, removing the reactive behaviour.

A timeout of 1 hour was imposed on model checking. The unroll depth was set to the largest loop size in the application when running CBMC, to avoid loop-unwinding assertions. We synthesized 10 test inputs per partition (using preconditions) in addition to the inputs obtained via counterexamples, to improve Daikon's invariant detection. Uninteresting invariants were filtered out using Daikon's filtration options, such as disabling disequality comparisons. In addition, we implemented a filtering strategy that dropped invariants comparing unrelated variables, e.g. invariants comparing only the function's updated variables at the entry point (i.e. preconditions), and invariants relating only input variables at the exit (i.e. postconditions). The tool was fully automatic - no human intervention was required, and no user annotations were used for any of the case studies.

We also experimented with SatAbs[17] tool. We used SatAbs with default parameters and a timeout of 1 hour per assertion. All experiments were conducted on an Intel Xeon 4-core processor running at 2.4 GHz, with processes allowed to take 3GB of memory.

(1)	(2)	(3)	(4)	(5)		(6)	(7)	(8)	(9)	(10)
App	**# Funcs**	**LoC**	**LoA**	**Result**		**#Abs Funcs**	**# Parts**	**Avg Invs**	**Avg Iters**	**#Refs**
				SatAbs	**DIV**					
SimpleEx	3	24	17	safe	safe	1	1	6	2	1
bist_cell	18	449	299	safe	safe	8	14	26	25	2
			199	unsafe	unsafe	2	2	2	10	2
pc_sfifo3	19	555	358	safe	safe	8	13	9	7	1
			508	unsafe	unsafe	5	5	8	32	5
token_ring10	35	1567	1152	TO	safe	14	14	-	-	0
			451	TO	unsafe	23	23	-	-	0
			675	TO	unsafe	12	12	2	8	2
transmitter12	39	1777	1322	TO	safe	16	16	-	-	0
			519	TO	unsafe	27	27	2	9	1
			775	TO	TO	15	15	-	-	-
transmitter13	41	1899	1715	TO	safe	17	17	-	-	0
			560	TO	unsafe	29	29	2	4	2
			836	TO	TO	15	15	-	-	-
token_ring13	41	1936	1341	TO	safe	17	17	-	-	0
			127	TO	unsafe	29	29	1	2	1
			579	TO	unsafe	29	29	2	8	1
			875	TO	TO	15	15	-	-	-

Table 1. Experimental Results

4.2 Results

Column 5 in table 1 presents the results of applying the SatAbs and DIV for checking assertions seeded in the code. `safe, unsafe, TO` indicates that the assertion was reported safe, the assertion was violated, or the model checker timed out, respectively. Column 6 reports the number of abstracted functions based on the call-graph approach. The column also states the total number of functions abstracted in the process. Column 7 reports the number of partitions generated across all abstracted functions (excluding point partitions). Column 8 reports average number of invariants generated per partition. Column 9 reports average number of iterations of the loops on lines 10-14 and 19-23 in algorithm 1. Finally, column 10 reports the number of refinements of SatAbs and DIV required to achieve the result.

4.3 Analysis & Observations

CBMC when applied alone on the assertions, without our summarization technique, either *timed out* or went *out of memory* for the chosen assertions, except row 1. SatAbs terminated successfully for rows 1-5 and timed out for the rest of the cases. DIV scaled better than both CBMC and SatAbs.

– **Scalability:** Our approach demonstrates that summarizing code fragments scales model checking. The novelty of our approach is to use relatively low-cost dynamic analysis to construct summaries and verify them at unit level, making the analysis sound, compositional and scalable.

– **Summary refinement:** Our approach of forming a pre-post pair consisting of the observed spurious behaviour using inferred invariants yields highly precise summaries during refinement. This is confirmed by the low number of refinements required by DIV (column 10) to terminate with a result. Further, our approach shows that executing code fragments with *well chosen inputs* (derived from preconditions) enables Daikon to infer useful and stronger invariants.
As seen from columns 6 and 7, the number of pre-post pairs is comparable to the number of abstracted functions. This indicates that most of the pre-post pairs did not require partitioning and were refined by discovering stronger postconditions, sufficient to check the assertion. We confirmed this by observing the invariants.

– **Overhead of dynamic inference:** Columns 8-9 give an idea of the complexity of *repeat-until* loops of algorithm 1, used to form new pre-post pairs. Though several iterations were required to obtain sound pre-post pairs using Daikon per partition, the model checker was observed to quickly terminate within a minute. Our invariant filtration strategy combined with instantiating only selected invariants from Daikon's template resulted in low number of invariants representing pre-post, as seen from column 8.

– **Call graph-based summarization:** The call-graph approach maximizes functions that can be summarized. Overall, we were able to summarize more than 50% functions, as seen from column 6. Even with this aggressive abstraction, for rows 11,14,18 CBMC timed out. In these cases, we observed that less than 40% functions could be summarized. When the call graph is unbalanced or the assertion is placed deep down in the graph, few functions can be abstracted, resulting in scalability issues.

– **Code modularity:** Our abstraction technique is well suited to take advantage of modularity in code, where functions have low coupling. In particular, a summary is refined only when a counterexample trace passes through the function, optimising the refinement procedure. As seen in rows 6,7,9,12,15, the result was obtained with the initial abstraction $\hat{f} = \{\langle true, true \rangle\}$ in place, without any refinement. In all other cases, we observed that not all abstracted functions underwent refinement.

5 Related

Dynamic analysis to scale model checking has been used by various researchers [18–21]. Daikon has been used in conjunction with static analysis tools such

as ESC/Java to infer properties and specifications for subsequent verification or user consumption [8], and to achieve both scale and automation of theorem proving [22].

Gulavani et.al. [23] propose a combination of DART-like dynamic and predicate-based static reasoning, called the Synergy algorithm. Test executions are used to refine an abstraction of the program, which in turn generates new test cases attempting to violate the given assertion. The abstraction and refinement strategies differ from our approach, which focuses on summarizing functions.

Kroening et.al. [24] proposed loop summarization using abstract transformers for checking assertions using CBMC. They abstract program loops with respect to a given abstract interpretation. Similar to our technique, they instantiate candidate invariants from a template and check which invariant holds over the loop, except that we use Daikon to guess invariants. Leaping counterexamples provide diagnostic information to user but can include spurious ones as they do not refine the abstraction. In contrast, our approach refines the abstraction to eliminate spurious counterexamples.

Taghdiri [10] proposed an approach to statically abstract procedure specifications. The initial over-approximation is refined using spurious counterexamples returned by a SAT solver. Unsat cores reported by the solver while concretizing (spurious) traces are conjuncted with the existing specification, to iteratively strengthen the specification. Our approach differs in the way that, the abstraction is refined both by blocking (a family of) spurious counterexamples and increasing precision by adding new pre-post pairs.

6 Conclusion and Future work

We have presented a CEGAR-based technique that combines model checking with dynamic analysis. The central idea is to use dynamically inferred invariants to separate observed spurious behaviour, refining the abstraction. The technique is sound, compositional and scales better than some existing tools as seen from experiments.

A limitation of our approach is that we need to execute code and it's fragments, which requires a large infrastructure support. Analysis of code that manipulates complex files or network data, for instance, may require some effort for automating test executions.

As part of the future work, we envision the following:

- Our technique can be generalized to include different reasoning techniques for the decomposed code fragments. In particular, we would like to use static analysis methods to obtain sound postconditions for dynamically inferred preconditions.
- Daikon templates can be extended to infer suitable invariants over different domains. We would like to extend our approach to more challenging areas like heap analysis and concurrency.

References

1. Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: Whitebox fuzz testing in production. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press (2013) 122–131
2. Lakhotia, K., McMinn, P., Harman, M.: Automated test data generation for coverage: Haven't we solved this problem yet? In: Testing: Academic and Industrial Conference-Practice and Research Techniques, 2009. TAIC PART'09., IEEE (2009) 95–104
3. Beyer, D.: Competition on software verification. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2012) 504–524
4. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM **53**(2) (2010) 66–75
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does astrée scale up? Formal Methods in System Design **35**(3) (2009) 229–264
6. Graf, S., Saïdi, H.: Construction of abstract state graphs with pvs. In: CAV. (1997) 72–83
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169
8. Nimmer, J.W., Ernst, M.D.: Invariant inference for static checking. In: SIGSOFT FSE. (2002) 11–20
9. Polikarpova, N., Ciupa, I., Meyer, B.: A comparative study of programmer-written and automatically inferred contracts. In: ISSTA. (2009) 93–104
10. Taghdiri, M.: Inferring specifications to detect errors in code. In: ASE. (2004) 144–153
11. Yeolekar, A., Unadkat, D., Agarwal, V., Kumar, S., Venkatesh, R.: Scaling model checking for test generation using dynamic inference. In: International Conference on Software Testing, Verification and Validation (ICST 2013), IEEE (2013)
12. 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) (2007) 35–45
13. Ernst, M., et al.: The daikon invariant detector. URL: http://pag.lcs.mit.edu/daikon
14. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: TACAS. (2004) 168–176
15. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. PLDI '12, ACM (2012) 181–192
16. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos benchmarks. URL: https://es.fbk.eu/tools/kratos/index.php?n=Main.Benchmarks
17. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design (FMSD) **25** (September–November 2004) 105–127
18. Yuan, J., Shen, J., Abraham, J.A., Aziz, A.: On combining formal and informal verification. In: CAV. (1997) 376–387
19. Shacham, O., Sagiv, M., Schuster, A.: Scaling model checking of dataraces using dynamic information. J. Parallel Distrib. Comput. **67**(5) (2007) 536–550

20. Kroening, D., Groce, A., Clarke, E.: Counterexample guided abstraction refinement via program execution. In: Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods, Springer (2004) 224–238
21. Gunter, E.L., Peled, D.: Model checking, testing and verification working together. Formal Aspects of Computing **17** (2005) 201–221
22. Win, T., Ernst, M.: Verifying distributed algorithms via dynamic analysis and theorem proving. (2002)
23. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. SIGSOFT '06/FSE-14, New York, NY, USA, ACM (2006) 117–127
24. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: ATVA. (2008) 111–125