Function approximation in bounded model checking

Daniil Stepanov
Saint Petersburg Polytechnic University, Russia
JetBrains Research
stepanov@kspt.icc.spbstu.ru

ABSTRACT

Bounded model checking (BMC) is one of the most interesting, relevant and promises method of software quality ensureness. One of the mainly BMC minuses is the interprocedural analysis is in all functions calls sides perfomed function inlining, which often makes analyze impossible because of difficulty and size of the resulting program. This work represents a method of solving this problem through function approximation is short description of function behavior which replaces full body in call sides. The prototype implementation in Borealis BMC has been evaluated on many programs and showed that our approach is able to give a correct summaries and these summaries have some positive effect on the analysis process.

Keywords

Bounded model checking, summaries, software quality, BMC improvements

1. INTRODUCTION

Software quality is very important aspect of software development because in our days software use in many areas of human life. Errors in modern software can lead to fatal consequences. Methods of software quality improving consists of programmers bugs fixing. Bounded model checking [5] is one of the most popular methods of software quality improving. In this method program represented as first order logic formula. Next this formula is sends in the SMT-solver, which makes conclusion about it's satisfiability. Thus considered full space of program states, aimed to find states, which brakes security properties, for example, outing bounds of array. One of the hardest questions in BMC is interprocedural analysis — how function calls mapped in the program state [11]. As default in BMC perfomed function inlining, which often makes analyze impossible because of size of the resulting program state.

This work represents a method of resolving interprocedural analysis problem in BMC through function approximation — short description of function behavior. A function approximation expressed in terms of its arguments and return value. Main advantage of using approximation is that they decrease the size of resulting formulae. Prototype is based on bounded model checker Borealis [1], but the method of approximation can be applied for another BMC tools. Evaluation results show that our approach is able to give some positive effect to BMC process.

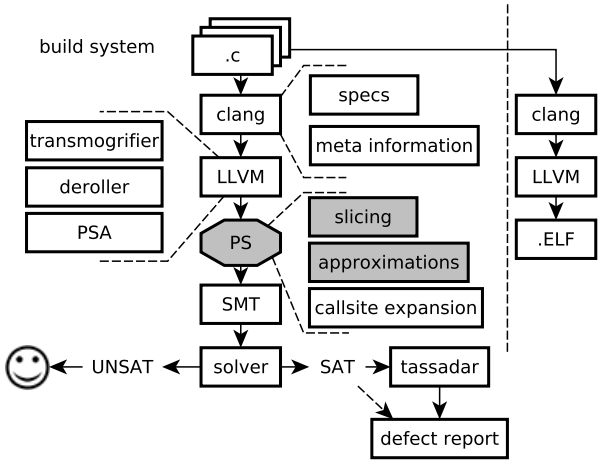
The rest of the paper is organized as follows. We lay the foundation for our work by introducing Borealis and bounded model checking in section 2. The process of taking functions approximations explained in section 3. We talk about preliminary evaluation results and related work in sections 5 and 6 respectively.

DOI:

2. BOREALIS BASICS

This work is based on Borealis bounded model checker [1], which used for C programming language. A large-scale overview of Borealis is shown in figure 1.

Figure 1: High-level scheme of Borealis bounded model checker



Borealis based on LLVM [12] framework, it works by converting a program representation to SMT formulae, which is then checked for possible satisfiability. Borealis uses its own program representation named Predicate State (PS). PS for given LLVM instruction corresponds to SMT formulae, which describes all possible program states in this in-

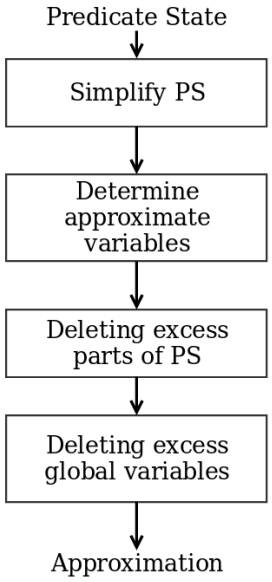
struction. PS for a given LLVM IR instruction corresponds to an SMT formula, which describes all possible program states at this instruction. A simplified description of PS format is shown in figure 2.

Functions approximations can be hand-crafted by the user in the form of source code or external annotations or external tools for component description. Also approximations can be extracted from source code using Craig interpolation or dynamic analyses. In this work we represented yet another approach to function approximation based on extraction from source code.

3. PREDICATE MINING

The overall algorithm of function approximation is show in figure 3

Figure 3: Function approximation scheme



In this work we focus on dataflow predicates, specifically, on predicates corresponding to conditional statements over function arguments. The premise is that, if a condition checks an argument before a function call, this condition may capture an interesting precondition of this function.

Simplistically, one can collect all relevant conditions by slicing the call site PS w.r.t. function arguments. However, as LLVM IR uses static single assignment form (SSA), each argument may be mapped to several LLVM IR variables, which greatly complicates slicing (because of the need to track possible dependencies between variables).

To sidestep this problem, we gain advantage of the boundness of the underlying program model used in BMC. As PS is fully unrolled and is, essentially, a directed acyclic graph (DAG) in SSA form, we can propagate the equalities between variables along the DAG, removing most of the temporary variables and placing the function arguments directly in conditionals. An example of this equality predicate substitution is shown in figure 4.

After slicing the substituted PS, we get a DAG containing all conditionals interesting w.r.t. some function call. Before saving these predicates for subsequent summarization, there is one additional step needed. Some conditionals in alternative DAG branches might form a complete group over the possible arguments, which does not contribute to the interesting preconditions and should be removed.

This is done using algorithm 1. We traverse the PS and remove all substates whose conditionals form a complete group. The check for a complete group is implemented using SMT solver as follows.

A set of conditionals $C = \{c_1, c_2, \ldots, c_n\}$ form a complete group, if for every possible argument value at least one of c_i is true. Therefore, if $\overline{c_1 \vee c_2 \vee \ldots c_n}$ is unsatisfiable, these conditionals form a complete group; otherwise, there is at least one argument value which falsifies all conditionals, and they should be retained.

Algorithm 1 PS compaction over complete conditionals

Input: S — sliced PS
Output: S' — compacted PS

```
S' ← empty
for all s ∈ S do
    c ← sliceConditions(s)
    if ¬isCompleteGroup(c) then
        S' ← S' + s
    end if
end for
```

The result of predicate mining is a set of compacted PS for every function call encountered during mining. These sets are stored in an external database[1], which is used to provide the aggregated information about interesting predicates over both the current and the previously analyzed programs.

4. CONTRACT SUMMARIZATION

To create a contract for a given function f, we need to summarize all interesting predicates mined previously for f. Let P be a set of unique predicates found for f, N be a mapping from predicates to their counts of occurencies. We summarize P and N into a contract in the following steps:

- Unsatisfiable core pruning
- Predicate weakening
- Frequent predicate collection

Let us describe these steps in more detail.

4.1 Unsatisfiable core pruning

Set P might contain predicates which contradict each other. There are two possible reasons for this: either these predicates do not form a precondition for f, or one of these predicates is a programmer's error. Unfortunately, there is no easy way of differentiating between these options; in our approach we decided to prune such predicates.

A naive approach to pruning would be to pairwise compare the predicates to find all contradictory pairs; however, this takes $O(|P|^2)$ calls to an SMT solver and, therefore, is very time-consuming. To speed-up the pruning, we use *unsatisfiable cores* — minimal subsets of predicates whose conjunctions are unsatisfiable — as they are easily obtainable from an SMT solver. An outline of unsatisfiable core pruning is shown in algorithm 2.

Unsatisfiable core pruning consists of two steps. The first step (lines 2–11) collects the maximal subset $UnsatCores$ of

[1]PS storage details are left outside the scope of this paper

Figure 2: Predicate state format

⟨*PredicateState*⟩ ::= `PredicateStateChain` head:⟨*PredicateState*⟩ tail:⟨*PredicateState*⟩
| `PredicateStateChoice` choices:⟨*ListOfPredicateStates*⟩
| `BasicPredicateState` data:⟨*ListOfPredicates*⟩

⟨*Predicate*⟩ ::= `AllocaPredicate` lhv:⟨*Term*⟩ numElems:⟨*Term*⟩ origNumElems:⟨*Term*⟩
| `DefaultSwitchCasePredicate` cond:⟨*Term*⟩ cases:⟨*ListOfTerms*⟩
| `EqualityPredicate` lhv:⟨*Term*⟩ rhv:⟨*Term*⟩
| `GlobalsPredicate` globals:⟨*ListOfTerms*⟩
| `InequalityPredicate` lhv:⟨*Term*⟩ rhv:⟨*Term*⟩
| `MallocPredicate` lhv:⟨*Term*⟩ numElems:⟨*Term*⟩ origNumElems:⟨*Term*⟩
| `SeqDataPredicate` base:⟨*Term*⟩ data:⟨*ListOfTerms*⟩
| `SeqDataZeroPredicate` base:⟨*Term*⟩ size:*UInt32*
| `StorePredicate` ptr:⟨*Term*⟩ value:⟨*Term*⟩
| `WriteBoundPredicate` ptr:⟨*Term*⟩ boundValue:⟨*Term*⟩
| `WritePropertyPredicate` propName:⟨*Term*⟩ ptr:⟨*Term*⟩ propValue:⟨*Term*⟩

⟨*Term*⟩ ::= `ArgumentTerm` idx:*UInt32* kind:⟨*ArgumentKind*⟩
| `ArgumentCountTerm`
| `AxiomTerm` term:⟨*Term*⟩ axiom:⟨*Term*⟩
| `BinaryTerm` op:⟨*BinaryOp*⟩ lhv:⟨*Term*⟩ rhv:⟨*Term*⟩
| `BoundTerm` term:⟨*Term*⟩
| `CastTerm` term:⟨*Term*⟩ signExtend:*Bool*
| `CmpTerm` op:⟨*CmpOp*⟩ lhv:⟨*Term*⟩ rhv:⟨*Term*⟩
| `ConstTerm`
| `FreeVarTerm`
| `GepTerm` base:⟨*Term*⟩ shifts:⟨*ListOfTerms*⟩ triviallyInbounds:*Bool*
| `LoadTerm` ptr:⟨*Term*⟩
| `ReadPropertyTerm` propName:⟨*Term*⟩ ptr:⟨*Term*⟩
| `ReturnPtrTerm` funcName:*String*
| `ReturnValueTerm` funcName:*String*
| `SignTerm` value:⟨*Term*⟩
| `TernaryTerm` cond:⟨*Term*⟩ tru:⟨*Term*⟩ fls:⟨*Term*⟩
| `UnaryTerm` op:⟨*UnaryOp*⟩ value:⟨*Term*⟩
| `ValueTerm` global:*Bool*
| `VarArgumentTerm` index:*UInt32*
| ...

⟨*ListOfPredicateStates*⟩ ::= ⟨*PredicateState*⟩ ⟨*ListOfPredicateStates*⟩ | ⟨*empty*⟩

⟨*ListOfPredicates*⟩ ::= ⟨*Predicate*⟩ ⟨*ListOfPredicates*⟩ | ⟨*empty*⟩

⟨*ListOfTerms*⟩ ::= ⟨*Term*⟩ ⟨*ListOfTerms*⟩ | ⟨*empty*⟩

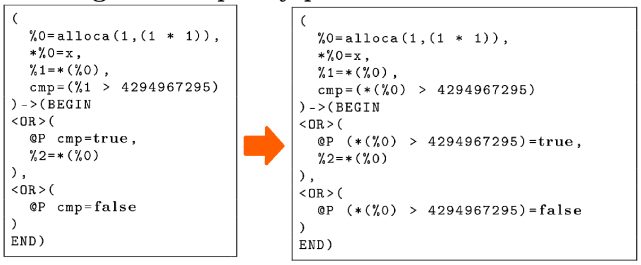
⟨*ArgumentKind*⟩ ::= `ANY` | `STRING`

⟨*CmpOp*⟩ ::= `EQ` | `NEQ` | `GT` | ...

⟨*BinaryOp*⟩ ::= `ADD` | `SUB` | `MUL` | ...

⟨*UnaryOp*⟩ ::= `NEG` | `NOT` | `BINARY_NOT`

Figure 4: Equality predicate substitution



Algorithm 2 Unsatisfiable core pruning

Input: P — set of unique predicates
Output: P — set of unsat-pruned unique predicates

1: $UnsatCores \leftarrow \emptyset$
2: **while** *true* **do**
3: $UC \leftarrow getUnsatCore(P)$
4: **if** $UC = \emptyset$ **then**
5: **break**
6: **end if**
7: $UnsatCores \leftarrow UnsatCores \cup UC$
8: **for all** $p \in UC$ **do**
9: $P \leftarrow P \backslash p$
10: **end for**
11: **end while**
12: **for all** $uc \in UnsatCores$ **do**
13: $UC \leftarrow getUnsatCore(P \cup \{uc\})$
14: $UnsatCores \leftarrow UnsatCores \cup UC$
15: **for all** $p \in UC$ **do**
16: $P \leftarrow P \backslash p$
17: **end for**
18: **end for**

unsatisfiable predicates via several calls to *getUnsatCore*. The second step (lines 12–18) reinforces *UnsatCores* with other contradictory predicates by finding additional unsatisfiable cores of P w.r.t. predicates from *UnsatCores*. After pruning, set P contains only non-contradictory predicates, which can be viewed as a raw contract for f.

4.2 Predicate weakening

A set P of non-contradictory predicates can be simplified by extracting a subset of weakest predicates[2]. We consider the classic definition of weakest precondition — p is weaker than q, if $q \implies p$. This check is implemented (yet again) by employing an SMT solver; if $\overline{q \implies p}$ is unsatisfiable, $q \implies p$ always holds, i.e., p is weaker than q. Algorithm 3 shows the predicate weakening process.

4.3 Frequent predicate collection

After collecting the weakest predicates for our function f, we are prepared for the final step in contract summarization. As traditionally adopted in specification mining, we filter out predicates unusual w.r.t. the analyzed software, i.e., predicates infrequently encountered in the source code. The algorithm is shown in 4.

Our algorithm is parametric in the minimal number X and percentage K of predicate occurences. By changing these parameters, we can (to some degree) control the precision

[2]as we are interested in [weakest] preconditions

Algorithm 3 Predicate weakening

Input: P — set of unique predicates
Input: N — mapping from predicates to counts
Output: P — set of weakest predicates
Output: N — updated mapping

for all $p, q \in P \times P : p \neq q$ **do**
if $isWeaker(p, q)$ **then**
$P \leftarrow P \backslash p$
$N(p) \leftarrow N(p) + N(q)$
end if
end for

Algorithm 4 Frequent predicate collection

Param: K — allowed percentage $0 \leq K \leq 1$
Param: X — allowed number of occurences $0 \leq X$
Input: P — set of weakest predicates
Input: N — mapping from predicates to counts
Input: T — total number of occurences
Output: C — function contract

$C \leftarrow \emptyset$
if $X > T$ **then**
return
end if
for all $p \in P$ **do**
if $K \leq N(p)/T$ **then**
$C \leftarrow C \cup p$
end if
end for

and recall of our contract mining. If $K \to 1$, the balance is shifted towards precision, as we consider a condition to be a contract only if it is encountered very frequently w.r.t. function calls. In the opposite case of $K \to 0$, we greatly increase the recall of our approach, but the probability of mining an erroneous contract is also higher (as we might capture a programmer's error as a required precondition). Parameter X has similar properties, but in the space of absolute (versus relative) values of the number of occurences.

To get the optimal results, one needs to find a good balance between these two parameters. However, at the moment we consider this problem as our future work.

5. EVALUATION

We implemented our approach as a plugin for Borealis bounded model checker and conducted some preliminary evaluation. We were interested in three kinds of datum: how many contracts we have mined, how good these contracts are and what influence they have on BMC.

As discussed in section 4.3, our algorithm has two tunable parameters, K and X. We decided to select the values biased towards precision: $K = 0.68$, $X = 3$.

We selected four C projects of different size and complexity for our evaluation: `beanstalkd`, `iputils`, `wrk` and `git`. We also mined an initial predicate database on more than 40 open-source projects from GitHub. The evaluation results are shown in table 1.

One can easily see from the evaluation results that the contract mining results are highly dependent on the analyzed project. If the project does not contain many mineable functions (e.g., `wrk`), our algorithm shows little to no influence on the analysis results. On the other hand, if the

project is big (e.g., `git`), it will most probably contain a lot of different function calls, and we are able to collect a lot of interesting contracts.

Table 1: Evaluation results

Project	SLOC	Contracts [3]	Violations[4]
`beanstalkd`	7.5k	10	6
`iputils`	12k	15	6
`wrk`	80k	6	4
`git`	340k	186	59

Some examples of the extracted contracts for `git`, `iputils` and `beanstalkd` projects are shown in figure 5. Most of the contracts are pretty simple (as we selected K and X with precision in mind) and may seem modest; but they were created in a fully automatic fashion and capture relevant safety preconditions, without any need for the user interaction. In our future work we plan to analyze the impact of the parameter selection on the extraction results in more detail.

6. RELATED WORK

There already exsits a large body of work on function summarization. It can be roughly divided in two parts: approaches based on software mining and on static analysis.

6.1 Software mining

A lot of approaches to function summarization are based on different flavours of software mining [19, 9, 8, 15, 18]. Many of the works are based on dynamic techniques, i.e., summarization is done using the information collected during program execution. For example, in [19] the authors propose to collect execution traces during testing and infer interesting properties from them. As the traces are often partial (due to the peculiarities of test development), the algorithm is reinforced with property template inference, which accounts for incomplete traces. The authors of [9] tackle the same problem from a different angle — traces are summarized using a machine learning technique, based on a generate-and-check loop. Their approach has also been successfully extended to concurrent and distributed systems [4].

Despite all the advantages of dynamic techniques, they all share one critical flaw — they need to run the program — which makes them ill-suited for use in program static analysis. A number of static approaches to summarization have been developed to overcome this [15, 18].

As mentioned before, our approach is most similar to [15], which also collects different control- and dataflow predicates from the source code. These predicates are then summarized using frequent subset and sequence mining to create the function specification. Our approach differs in that it employs the capabilities of modern SMT solvers for summarization, to make the resulting contracts more succinct and precise; for another thing, it is limited to dataflow predicates, as we still working on how to embed control-flow predicates in the BMC setting. The work of [18] is also similar to [15], but is built upon automata abstractions, which are used to summarize possible control-flow predicates in an efficient way. Dual to our approach, this work targets only temporal control-flow predicates, as they go best with automata-based abstractions.

[3]number of collected contracts

[4]number of contract violations found

6.2 Static analysis

There are several ways static analysis can be used in fuction summarization. A number of approaches are based on different kinds of predicate abstraction [10] — a technique where predicates are constructed on-the-go and refined as the analysis explores the program, i.e., the set of predicates may change depending on the analyzed property. In [16] counterexample-guided abstraction refinement (CEGAR [7]) is used to find a function contract, by iteratively refining the program state until a sufficient precondition is found. Other approaches use classic Hoare logic and infer the needed function safety conditions in this framework [3].

Some approaches employ bi-abduction [6], a very powerful technique which answers the following question: for some given E and B, find AF and F such that $E \wedge AF \implies B \wedge F$. If we fix E to be the environment and B to be the function called, formula AF gives us exactly the contract for B w.r.t E. An advantage of this technique is that function summarization can be done on a per-call-site basis, thus greatly increasing its precision. An example of this technique is [2], where it is successfully used to create function summaries for all unknown functions in a given program.

The interpolation approaches to function summarization are founded on Craig interpolation [13], which allows one to find an intermediate formulae I[5] for $B \implies Q$, such that $B \implies I$ and $I \implies Q$. This interpolant summarizes a function w.r.t. some property of interest, if we take the function body as B and our safety property as Q. This idea has been explored in [17], where Craig interpolation is used for function summarization in the classic BMC setting. It allows one to greatly reduce the size of the resulting SMT formulae, thus increasing the analysis performance. The original idea of interpolation in BMC was introduced in [13] and [14], to make BMC applicable to infinite-state programs, by exploiting its [Craig interpolation] generalization power. This technique is very powerful, and we may consider augmenting our approach with it.

7. CONCLUSION

In this paper we presented an approach to automatic contract mining for use in BMC tools. By embedding the mining problem into the BMC and exploiting the power of modern SMT solvers, our approach is able to extract succinct function contracts. The parametric nature allows one to vary the precision-recall tradeoff on a per-project basis.

We implemented the proposed approach in a prototype plugin for Borealis BMC tool and did some preliminary evaluation. The evaluation results show our approach to be applicable to real software and to improve the BMC results.

There are several things we would like to consider in our future work. First, we want to do a more extensive and thorough evaluation of our approach, and to also analyze its influence on the quality of BMC results in more detail. Second, we would like to try automating parameter selection via some flavour of computer-aided optimization, and to study their effect on precision and recall. Third, predicate pruning can be amplified to do a more in-depth analysis of unsatisfiable cores, resulting in increased pruning precision. It could also be improved by considering predicates in a per-call-site fashion, thus improving pruning granularity.

[5]called Craig interpolant

Figure 5: Examples of the extracted contracts

```
—— Project: git ——
—— Function: fgets ——
(@P (arg$2 == <nullptr>)=false)
—— Function: ungetc ——
(@P (arg$0 == 4294967295)=false)
—— Function: lseek64 ——
(@P (arg$0 < 0)=false)
—— Function: git_deflate_bound ——
(@P (arg$1 +> 1024)=true)
—— Function: get_next_line ——
(@P (arg$0 +< arg$1)=true)
—— Function: check_patch ——
(@P (arg$0 == <nullptr>)=false)
—— Function: write_branch_report ——
(
  @P (arg$1 == <nullptr>)=false,
  @P (arg$0 == <nullptr>)=false
)
```

```
—— Project: iputils ——
—— Function: ping4_run ——
(@P (arg$2 == <nullptr>)=false)
—— Function: ping6_run ——
(@P (arg$2 == <nullptr>)=false)
—— Function: probe_ttl ——
(@P (arg$0 < 0)=false)
—— Function: ioctl ——
(@P (arg$0 == 4294967295)=false)

—— Project: beanstalkd ——
—— Function readdir ——
(@P (arg$0 == <nullptr>)=false)
—— Function reply_job ——
(@P (arg$1 == <nullptr>)=false)
—— Function bind ——
(@P (arg$0 == 4294967295)=false)
—— Function store_job ——
(@P (arg$0 == <nullptr>)=false)
```

8. REFERENCES

[1] M. Akhin, M. Belyaev, and V. Itsykson. Software defect detection by combining bounded model checking and approximations of functions. *Automatic Control and Computer Sciences*, 48(7):389–397, 2014.

[2] A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. POPL'16, pages 789–801, 2016.

[3] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. PASTE'05, pages 82–87, 2005.

[4] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. ICSE'14, pages 468–479, 2014.

[5] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[6] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. POPL'09, pages 289–300, 2009.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, Sept. 2003.

[8] A. Danese, T. Ghasempouri, and G. Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. DATE'15, pages 67–72, 2015.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. ICSE'99, pages 213–224, 1999.

[10] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. POPL'02, pages 191–202, 2002.

[11] W. Landi. Undecidability of static analysis. *LOPLAS*, 1(4):323–337, 1992.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. CGO'04, pages 75–86, 2004.

[13] K. L. McMillan. Applications of Craig interpolants in model checking. TACAS'05, pages 1–12, 2005.

[14] K. L. McMillan. Lazy abstraction with interpolants. CAV'06, pages 123–136, 2006.

[15] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. PLDI'07, pages 123–134, 2007.

[16] M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. ESOP'13, pages 451–471, 2013.

[17] O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based function summaries in bounded model checking. HVC'11, pages 160–175, 2012.

[18] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. ISSTA'07, pages 174–184, 2007.

[19] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. ICSE'06, pages 282–291, 2006.