Interpolation-based Function Summaries in Bounded Model Checking*

Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina

Formal Verification Lab, University of Lugano, Switzerland
{ondrej.sery,grigory.fedyukovich,natasha.sharygina}@usi.ch
http://verify.inf.usi.ch/

Abstract During model checking of software against various specifications, it is often the case that the same parts of the program have to be modeled/verified multiple times. To reduce the overall verification effort, this paper proposes a new technique that extracts function summaries after the initial successful verification run, and then uses them for more efficient subsequent analysis of the other specifications. Function summaries are computed as over-approximations using Craig interpolation, a mechanism which is well-known to preserve the most relevant information, and thus tend to be a good substitute for the functions that were examined in the previous verification runs. In our summarization-based verification approach, the spurious behaviors introduced as a side effect of the over-approximation, are ruled out automatically by means of the counter-example guided refinement of the function summaries. We implemented interpolation-based summarization in our FunFrog tool, and compared it with several state-of-the-art software model checking tools. Our experiments demonstrate the feasibility of the new technique and confirm its advantages on the large programs.

1 Introduction

Model checking is a popular technique for automated analysis of software. Due to the state explosion problem, it is usually infeasible to analyze a whole program in a single run starting from its entry point (e.g., the `main` function). Instead, the problem is often modularized and a model checker is used to exhaustively explore portions of the program for different properties. Typically, this means that the same code (e.g., same functions) of the original program is used in multiple model checker runs and it is analyzed multiple times. We observe that significant savings can be achieved if information concerning the already analyzed code is reused in the subsequent runs of the model checker.

We present a technique for extracting and reusing information about the already analyzed code, in the form of function summaries. The novelty of our work lies in the use of Craig interpolation [8] to extract function summaries

* This work is partially supported by the European Community under the call FP7-ICT-2009-5 — project PINCETTE 257647.

after a successful verification run. An interpolant-based function summary is an over-approximation of the actual function behavior and it symbolically captures all execution traces through the function. Since interpolants tend to contain mostly the relevant information, the computed function summaries are more compact than a precise representation of the function, and thus result in the overall verification efficiency gain. We prove that no errors are missed due to the use of the interpolation-based summaries. On the other hand, when spurious errors occur as a side-effect of over-approximation, our approach uses a counter-example guided strategy to automatically refine summaries.

The implementation of the proposed technique, the FunFrog tool, is based on the CBMC bounded model checker [6]. We use propositional encoding to get bit-precise reasoning. However, our approach is general and works also with SMT encodings for which an interpolation algorithm exists. To evaluate the new approach, we compared running times of FunFrog with the state-of-the-art model checkers CBMC, SATABS [7], and CPAchecker [4] on various benchmarks. The experimental results demonstrate feasibility and advantages of our approach.

2 Preliminaries

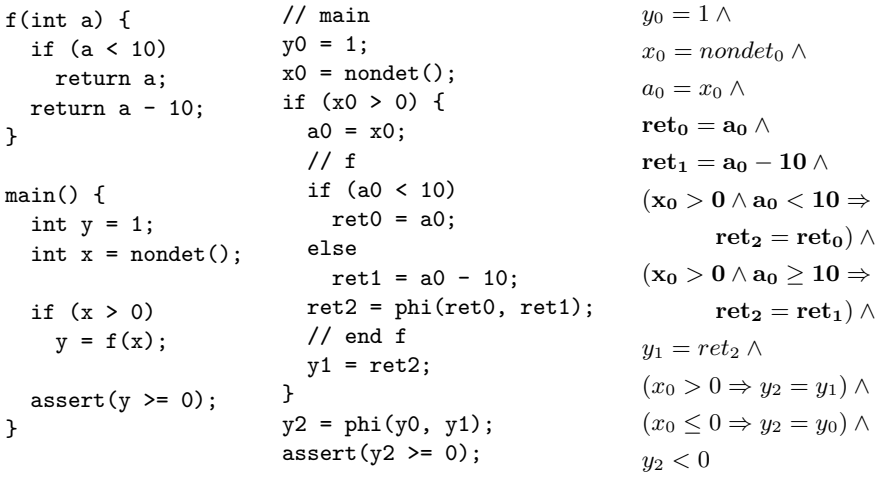
As customary in model checking, we use an adapted definition of interpolation:

Definition 1 (Craig interpolation). *Let A and B be formulas and $A \wedge B$ be unsatisfiable.* Craig interpolant *of (A, B) is a formula I such that $A \rightarrow I$, $I \wedge B$ is unsatisfiable, and I contains only free variables common to A and B.*

For an unsatisfiable pair of formulas (A, B), an interpolant always exists [8]. For many theories, an interpolant can be constructed from a proof of unsatisfiablity [19]. In this work, interpolation is used to extract function summaries in the context of bounded model checking (BMC). Therefore, for the sake of simplicity but without a loss of generality, the paper refers to unwound programs without loops and recursion as an input of the summarization algorithm. Intuitively, such a program is created from the original one by unwinding all loops and recursive calls by the given number (bound). Note that in our implementation the unwinding is performed on-the-fly when needed.

Definition 2. *An* unwound program *for a bound ν is a tuple $P_\nu = (F, f_{main})$, s.t. F is a finite set of functions and $f_{main} \in F$ is an entry point.*

We use relations $child, subtree \subseteq F \times F$, where *child* relates each function f to all the functions invoked by f, and *subtree* is a transitive closure of *child*. In addition, we use $\hat{F}$ to denote the finite set of unique function calls, i.e., function call with a unique combination of a call stack, a program location, and a target function (denoted by $target : \hat{F} \rightarrow F$). $\hat{F}$ corresponds to the invocation tree of the unwound program. By $\hat{f}_{main}$ we denote the implicit call of the program entry point and $target(\hat{f}_{main}) = f_{main}$. We extend the relations *child* and *subtree* to $\hat{F}$ in a natural way, s.t. $\forall \hat{f}, \hat{g} \in \hat{F} : child(\hat{f}, \hat{g}) \rightarrow child(target(\hat{f}), target(\hat{g}))$ and *subtree* is a transitive closure of the extended relation *child*.



(a) C code (b) SSA form (c) BMC formula

Figure 1: BMC formula generation

Standard BMC of software encodes an unwound program to a BMC formula in a way illustrated in Fig. 1 (more details on the encoding can be found in [6]). First, the unwound program is converted into the SSA form (Fig. 1b), where each variable is assigned at most once. A so called ϕ-function is used to merge values from different control-flow paths. Functions are expanded in the call site as if being inlined. Then a BMC formula (Fig. 1c) is constructed from the SSA form. Assignments are converted to equalities, path conditions are computed from branching conditions and used to encode ϕ-functions. Negation of the assertion condition guarded by its path condition (*true* in this case) is conjuncted with the BMC formula. The resulting BMC formula is unsatisfiable if the assertion holds. In the other case, a satisfying assignment identifies an error trace.

3 Function Summaries

This section first defines function summaries as a means to over-approximate functions in BMC. Then it shows how interpolation can be used as a way to extract function summaries after a successful verification run. Finally, it presents a BMC algorithm extended with the interpolation-based function summarization.

A function summary relates input and output arguments of a function. Therefore, a notion of arguments of a function is necessary. For this purpose, we expect to have a set of program variables $\mathbb{V}$ and a domain function $\mathbb{D}$ which assigns a domain (i.e., set of possible values) to every variable from $\mathbb{V}$.

Definition 3. *For a function f, sequences of variables $args^f_{in} = \langle in_1, \ldots, in_m \rangle$ and $args^f_{out} = \langle out_1, \ldots, out_n \rangle$ denote the input and output arguments of f,*

where $in_i, out_j \in \mathbb{V}$ *for* $1 \leq i \leq m$ *and* $1 \leq j \leq n$*. In addition,* $args^f = \langle in_1, \ldots, in_m, out_1, \ldots, out_n \rangle$ *denotes all the arguments of* f*. As a shortcut, we use* $\mathbb{D}(f) = \mathbb{D}(in_1) \times \ldots \times \mathbb{D}(in_m) \times \mathbb{D}(out_1) \times \ldots \times \mathbb{D}(out_n)$*.*

In the following, we expect that functions do not have other than input and output arguments, which include also the return value. Note that an in-out argument[1] is split into one input and one output argument. Similarly, a global variable accessed by a function is rewritten into the corresponding input or/and output argument, depending on the mode of access (i.e., read or/and write).

Precise behavior of a function can be defined as a relation over values of input and output arguments of the function as follows.

Definition 4 (Relational Representation). *Let* f *be a function, then the relation* $R^f \subseteq \mathbb{D}(f)$ *is the* relational representation *of the function* f*, if* R^f *contains exactly all the tuples* $\bar{v} = \langle v_1, \ldots, v_{|args^f|} \rangle$ *such that the function* f *called with the input values* $\langle v_1, \ldots, v_{|args^f_{in}|} \rangle$ *can finish with the output values* $\langle v_{|args^f_{in}|+1}, \ldots, v_{|args^f|} \rangle$*.*

Note that Def. 4 admits multiple combinations of values of the output arguments for the same combination of values of the input arguments. This is useful to model nondeterministic behavior, and for abstraction of the precise behavior of a function. In this work, the summaries are applied in BMC. For this reason, the rest of the text will be restricted to the following bounded version of Def. 4.

Definition 5 (Bounded Relational Representation). *Let* f *be a function and* ν *be a bound, then the relation* $R^f_\nu \subseteq R^f$ *is the* bounded relational representation *of the function* f*, if* R^f_ν *contains only the tuples representing computations with all loops and recursive calls unwound up to* ν *times.*

Then a summary of a function is an over-approximation of the precise behavior of the given function under the given bound. In other words, a summary captures all the behaviors of the function and possibly more.

Definition 6 (Summary). *Let* f *be a function and* ν *be a bound, then a relation* S *such that* $R^f_\nu \subseteq S \subseteq \mathbb{D}(f)$ *is a* summary *of the function* f*.*

The relational view on a function behavior is intuitive but impractical for implementation. Typically, these relations are captured by means of logical formulas. Def. 7 makes a connection between these two views.

Definition 7 (Summary Formula). *Let* f *be a function,* ν *a bound,* σ *a formula with free variables only from* $args^f$*, and* S *a relation induced by* σ *as* $S = \{\bar{v} \in \mathbb{D}(f) \mid \sigma[\bar{v}/args^f] \models true\}$*. If* S *is a summary of the function* f *and bound* ν*, then* σ *is a* summary formula *of the function* f *and bound* ν*.*

A summary formula of a function can be directly used during construction of the BMC formula to represent a function call. This way, the part of the SSA form

[1] E.g., a parameter passed by reference

$$
\begin{array}{ll}
y_0 = 1 \wedge & y_1 = ret_0 \wedge \\
x_0 = nondet_0 \wedge & (x_0 > 0 \Rightarrow y_2 = y_1) \wedge \\
a_0 = x_0 \wedge & (x_0 \leq 0 \Rightarrow y_2 = y_0) \wedge \\
(\mathbf{a_0} > \mathbf{0} \Rightarrow \mathbf{ret_0} > \mathbf{0}) \wedge & y_2 < 0
\end{array}
$$

Figure 2: BMC formula created using summary $a > 0 \Rightarrow ret > 0$ for function f

corresponding to the called function does not have to be created and converted to a part of the BMC formula. Moreover, the summary formula tends to be smaller. Of course, the arguments have to be assigned the correct SSA version. Considering the example in Fig. 1, using the summary formula $a > 0 \Rightarrow ret > 0$ for the function f yields the BMC formula in Fig. 2.

The important property of the resulting BMC formula is that if it is unsatisfiable (as in Fig. 2) then also the formula without summaries (in Fig. 1c) is unsatisfiable. Therefore, no errors are missed due to the use of summaries.

Lemma 1. *Let ϕ be a BMC formula of an unwound program P for a given bound ν, and let ϕ' be a BMC formula of P and ν, with some function calls substituted by the corresponding summary formulas bounded by ν', $\nu' \geq \nu$. If ϕ' is unsatisfiable then ϕ is unsatisfiable as well.*

Proof. Without loss of generality, suppose that there is only one summary formula σ_f substituted in ϕ' for a call to a function f. If multiple summary formulas are substituted, we can apply the following reasoning for all of them.

For a contradiction, suppose that ϕ' is unsatisfiable and ϕ is satisfiable. From the satisfying assignment of ϕ, we get values $\langle v_1, \ldots, v_{|args^f|} \rangle$ of the arguments to the call to the function f. Assuming correctness of construction of the BMC formula ϕ, the function f given the input arguments $\langle v_1, \ldots, v_{|args^f_{in}|} \rangle$ can finish with the output arguments $\langle v_{|args^f_{in}|+1}, \ldots, v_{|args^f|} \rangle$ and with all loops and recursive calls unwound at most ν times. Therefore, by definition of the summary formula, the values $\langle v_1, \ldots, v_{|args^f|} \rangle$ also satisfy σ_f. Since the rest of the formulas ϕ and ϕ' is the same, the satisfying assignment of ϕ is also a satisfying assignment of ϕ' (up to SSA version renaming). □

3.1 Interpolation-based Summaries

There may be multiple ways to obtain a summary formula. In this section, we present a way to extract summary formulas using Craig interpolation. To use interpolation, the BMC formula ϕ should have the form $\bigwedge_{\hat{f} \in \hat{F}} \phi_{\hat{f}}$ such that every $\phi_{\hat{f}}$ symbolically represents the function f, a target of the call $\hat{f}$. Moreover, the symbols of $\phi_{\hat{f}}$ shared with the rest of the formula are only the elements of $args^f$.

Note that the BMC formula is generally not in this form. Variables from the calling context tend to leak into the formulas of the called function as a part

$y_0 = 1 \wedge$
$x_0 = nondet_0 \wedge$
$a_0 = x_0 \wedge$
$\mathbf{x_0 > 0 \Leftrightarrow callstart_{\hat{f}}} \wedge$ (1)
$y_1 = ret_0 \wedge$
$(x_0 > 0 \Rightarrow y_2 = y_1) \wedge$
$(x_0 \leq 0 \Rightarrow y_2 = y_0) \wedge$
$(\mathbf{callend_{\hat{f}}} \vee \mathbf{x_0 \leq 0}) \wedge \mathbf{y_2 < 0}$ (2)

(a) formula $\phi_{\hat{f}_{main}}$

$ret_1 = a_0 \wedge$
$ret_2 = a_0 - 10 \wedge$
$(\mathbf{callstart_{\hat{f}}} \wedge a_0 < 10 \Rightarrow ret_0 = ret_1) \wedge$ (3)
$(\mathbf{callstart_{\hat{f}}} \wedge a_0 \geq 10 \Rightarrow ret_0 = ret_2) \wedge$ (4)
$(\mathbf{callend_{\hat{f}}} \Rightarrow \mathbf{callstart_{\hat{f}}})$ (5)

(b) formula $\phi_{\hat{f}}$

Figure 3: Partitioned bounded model checking formula

of the path condition. For example in Fig. 1c, the variable x_0 from the calling context of the function f appears in the bold part, which represents f itself. To achieve the desired form, we generate the parts of the formula corresponding to the individual functions in separation and bind them together using two boolean variables for every function call: $callstart_{\hat{f}}$ and $callend_{\hat{f}}$. We call the resulting formula a *partitioned bounded model checking* (PBMC) formula.

Fig. 3 demonstrates creation of a PBMC formula for the example from Fig. 1. Intuitively, $callstart_{\hat{f}}$ should be *true* when the corresponding function call is reached. Therefore, the formula of the calling context (Fig. 3a) makes it equivalent to the path condition of the call (1). The $callend_{\hat{f}}$ variable is *true* if the call returns. It is conjuncted with the path condition so it occurs in the guard of the assertion check (2). In the called function (Fig. 3b), $callstart_{\hat{f}}$ is taken as the initial path condition, and thus it appears in the expanded ϕ-function (3, 4). The value of $callend_{\hat{f}}$ is derived from the path conditions[2] at function exit points (5). The two helper variables are added to the set of function arguments $args^f$. Therefore, the variables shared between the individual formulas $\phi_{\hat{f}}$ and the rest of the PBMC formula (here $\phi_{\hat{f}_{main}}$) are only the variables from $args^f$.

If the resulting PBMC formula is unsatisfiable, we compute multiple Craig interpolants from a single proof of unsatisfiability to get function summaries.

Definition 8 (Interpolant summary formula). *Let $\hat{f}$ be a function call of an unwound program P, ν a bound, and $\phi \equiv \bigwedge_{\hat{g} \in \hat{F}} \phi_{\hat{g}}$ an unsatisfiable PBMC formula for P. Furthermore, let $I_\nu^{\hat{f}}$ be a Craig interpolant of (A, B) such that $A \equiv \bigwedge_{\hat{g} \in \hat{F}: subtree(\hat{f}, \hat{g})} \phi_{\hat{g}}$, and $B \equiv \bigwedge_{\hat{h} \in \hat{F}: \neg subtree(\hat{f}, \hat{h})} \phi_{\hat{h}}$. Then the interpolant $I_\nu^{\hat{f}}$ is an* interpolant summary formula.

Of course, an important property of the interpolant summary formula is that it is indeed a summary formula from Def. 7.

[2] Note that the implication may be more complicated, e.g., if the function can exit the program or if it contains user assumptions that prune some computational paths.

Lemma 2. *The interpolant $I_\nu^{\hat{f}}$ constructed by Def. 8 is a summary formula for the function f and the bound ν.*

Proof. By definition of Craig interpolation, the only free variables of $I_\nu^{\hat{f}}$ are from $args^f$. Moreover, we know that $A \Rightarrow I_\nu^{\hat{f}}$ and that A represents the call $\hat{f}$ with all function invocations within it. By construction of A and the PBMC formula ϕ, every tuple of values $\bar{v} \in R_\nu^f$ defines a partial valuation of A that can be extended to a satisfying valuation of A. Therefore by $A \Rightarrow I_\nu^{\hat{f}}$, all these partial valuations satisfy $I_\nu^{\hat{f}}$ as well. The relation S induced by the satisfying valuations of $I_\nu^{\hat{f}}$ thus satisfies $R_\nu^f \subseteq S \subseteq \mathbb{D}(f)$. □

Another useful property of the interpolant summary formula is that $I_\nu^{\hat{f}} \wedge B$ is unsatisfiable (by Def. 1). In other words, the interpolant summary formula contains all the necessary information for showing that the program under analysis is safe with respect to the property being analyzed. Since the interpolant is created from a proof of unsatisfiablity of $A \wedge B$, it tends to contain only the relevant part and thus be smaller than A. An important consequence is that the interpolant summary formulas can be used to abstract function calls in BMC without missing errors that are reachable within the given bound.

Theorem 1. *Let ϕ be a BMC formula of an unwound program P for a given bound ν and let ϕ' be a BMC formula of P and ν, with some function calls substituted by the corresponding interpolant summary formulas bounded by ν', $\nu' \geq \nu$. If ϕ' is unsatisfiable then ϕ is unsatisfiable as well.*

Proof. The proof directly follows from Lemmas 1 and 2. □

3.2 Algorithm

An overview of the BMC algorithm for creation of the PBMC formula and extraction of interpolant summaries is depicted in Alg. 1. First, the algorithm creates the PBMC formula. It takes one function at a time and creates the corresponding part of the formula (line 12) using the SSA encoding as sketched in Section 2. The difference lies in handling of function calls. When available, function summaries (line 8) are used instead of processing the function body (`ApplySummary` maps the symbols in the summary to the correct SSA version). Otherwise, the function is queued for later processing (line 10). In both cases, a glue part of the formula, which reserves the argument SSA versions and generates the $callstart_{\hat{f}}$ and $callend_{\hat{f}}$ bindings as described above, is created (line 6).

Having the PBMC formula, the algorithm calls a SAT or SMT solver. In the case of a successful verification (UNSAT answer), the algorithm extracts new function summaries (line 18-24). For many functions, the summary is just a trivial *true* formula, which means that the function is not relevant for validity of the property being verified. Note that the function `StoreSummary` (line 23) also does a simple filtering, i.e., if there are multiple summaries for a single function, it checks that none of them implies any other. Though this means a quadratic number of solver calls in general, in our experience, the actual cost is very small.

Algorithm 1: BMC algorithm with summary application and extraction.

Input: Unwound program $P_\nu = (F, f_{main})$ with function calls $\hat{F}$
Output: Verification result: $\{SAFE, UNSAFE\}$
Data: D: queued function calls, ϕ: PBMC formula

```
1  D ← {f̂_main}, φ ← true ;                                  // (1) formula creation
2  while D ≠ ∅ do
3      choose f̂ ∈ D, and D ← D \ {f̂};
4      φ_f̂ ← true;
5      foreach ĝ s.t. child(f̂, ĝ) do
6          φ_f̂ ← φ_f̂ ∧ ReserveArguments(ĝ);
7          if HasSummary(ĝ) then
8              φ_f̂ ← φ_f̂ ∧ ApplySummary(ĝ) ;                  // apply summaries
9          else
10             D ← D ∪ {ĝ} ;                                  // process ĝ later
11     end
12     φ_f̂ ← φ_f̂ ∧ CreateFormula(f̂);
13     φ ← φ ∧ φ_f̂
14 end
15 result ← Solve(φ) ;                                       // (2) run solver
16 if result = SAT then
17     return UNSAFE;
18 foreach f̂ ∈ F̂ do                                         // (3) extract summaries
19     A ← ⋀_{ĝ∈F̂:subtree(f̂,ĝ)} φ_ĝ;
20     B ← ⋀_{ĥ∈F̂:¬subtree(f̂,ĥ)} φ_ĥ;
21     I_f̂ ← Interpolate(A, B);
22     if I_f̂ ≠ true then
23         StoreSummary(I_f̂);
24 end
25 return SAFE;
```

4 Refinement

When the PBMC formula is satisfiable, the BMC algorithm reports an error (line 17 of Alg. 1), which can be either a real or a spurious violation since function summaries are computed using over-approximation. This section introduces an algorithm that iteratively refines the PBMC formula until either a real error is found or an unsatisfiable PBMC formula is detected. The refinement algorithm uses the generalized version of Alg. 1 that can be executed with a specified level of approximation.

Definition 9. *A* substitution scenario *for function calls is a function* $\Omega : \hat{F} \to \{inline, sum, havoc\}$.

For each function call, a substitution scenario determines a level of approximation as one of the following three options: *inline* when it processes the whole

function body; *sum* when it substitutes the call by an existing summary, and *havoc* when it treats the call as a nondeterministic function. The *havoc* option abstracts from the call; it is equivalent to using a summary formula *true*. To employ these options, we replace lines 7-10 of Alg. 1 by the following code:

```
7  switch Ω(ĝ) do
8      case sum:  φ_f̂ ← φ_f̂ ∧ ApplySummary(ĝ) ; // apply summaries
9      case inline:  D ← D ∪ {ĝ} ;              // process ĝ later
10     case havoc:  skip;          // treat ĝ nondeterministically
11 endsw
```

For example, a substitution scenario that makes the generalized algorithm equivalent to Alg. 1 looks as follows:

$$\Omega_0(\hat{g}) = \begin{cases} sum, & \text{if } \texttt{HasSummary}(\hat{g}) = true \\ inline, & \text{otherwise} \end{cases}$$

The substitution scenario used as the initial approximation is called *initial scenario* and denoted as Ω_0. The above initial scenario is *eager*, since it eagerly processes bodies of functions without available summaries. Alternatively, one can use a *lazy* initial scenario to treat functions without available summaries as nondeterministic ones (by replacing the *inline* with *havoc* case). This results in a smaller initial PBMC formula and leaves identification of the important function calls to the refinement loop, possibly resulting in more refinement iterations.

When a substitution scenario Ω_i leads to a satisfiable PBMC formula, a *refinement strategy* either shows that the error is real or looks for another substitution scenario Ω_{i+1}. In the latter case, Ω_{i+1} represents a tighter approximation, i.e., it refines Ω_i.

Definition 10. *Given two substitution scenarios Ω_1, Ω_2, we say that Ω_2* refines *Ω_1, if $\forall \hat{f} \in \hat{F} : \Omega_1(\hat{f}) = inline \rightarrow \Omega_2(\hat{f}) = inline$, and $\exists \hat{g} \in \hat{F} : \Omega_1(\hat{g}) \neq inline \wedge \Omega_2(\hat{g}) = inline$.*

Note, that due to a finite size of $\hat{F}$, the refinement loop terminates independently from the refinement strategy (i.e., the choice of Ω_{i+1}). Rephrasing Def. 10, we have $\{\hat{f} \in \hat{F} \mid \Omega_i(\hat{f}) = inline\} \subset \{\hat{f} \in \hat{F} \mid \Omega_{i+1}(\hat{f}) = inline\} \subseteq \hat{F}$. Therefore, the sequence of sets $\{\hat{f} \in \hat{F} \mid \Omega_i(\hat{f}) = inline\}$ grows strictly monotonically while being bounded by $\hat{F}$. If the refinement loop reaches a substituting scenario $\Omega_\top$ such that $\forall \hat{f} \in \hat{F} : \Omega_\top(\hat{f}) = inline$, the generalized algorithm using $\Omega_\top$ is equivalent to BMC without summarization, thus yielding the same precise answer. In the following, we call $\Omega_\top$ the *supreme scenario*.

Counter-example guided refinement. We propose a refinement strategy based on analysis of an error trace. When refining a substitution scenario Ω_i, the counter-example guided refinement strategy refines the function calls that (1) are substituted by a summary or havoced in Ω_i and (2) are on the error trace corresponding to the given satisfying assignment of the current PBMC formula and (3) do influence validity of the assertion being analyzed.

The second point is deduced from the satisfying assignment of the PBMC formula. By construction of the PBMC formula, a variable $callstart_{\hat{f}}$ is valuated

to true, if and only if the satisfying assignment represents a trace that includes the function call $\hat{f}$. Therefore, all function calls for which the *callstart* variable is assigned *true* are suspected. The third point is decided based on a path-sensitive dependency analysis over the SSA form. As a result, only the function calls that actually influence validity of the assertion are marked *inline* in Ω_{i+1}. If no such function call exists, the error trace is real and it is reported to a user.

$$\Omega_{i+1}(\hat{g}) = \begin{cases} \mathit{inline}, \text{ if } \Omega_i(\hat{g}) \neq \mathit{inline} \wedge \mathit{callstart}_{\hat{g}} = \mathit{true} \wedge \texttt{InfluenceProp}(\hat{g}) \\ \Omega_i(\hat{g}), \text{ otherwise} \end{cases}$$

Note that we do not explicitly test whether the error trace is feasible. The error trace can be simulated exactly, where summaries are not used. However, a summary hides precise paths inside the substituted function and only the inputs and outputs of the functions are preserved in the satisfying assignment. Thus all the possible paths through the function would have to be considered to see whether this combination of inputs and outputs is indeed possible. This becomes costly for summaries of large functions and the advantage of having a simple abstraction might be lost.

For experimentation purposes, we define another simplistic refinement strategy, a *greedy* one. When the PBMC formula corresponding to the chosen initial scenario Ω_0 is satisfiable, the greedy strategy simply refines Ω_0 directly to the supreme scenario $\Omega_\top$. This way, the greedy strategy fallbacks to the standard BMC when the approximation is too coarse to prove the assertion being verified.

5 Evaluation

We implemented the interpolation-based function summarization and refinement in a tool called FunFrog, extending the CBMC model checker. Currently, there is a limited support for pointers. The OpenSMT solver [5] is used both for satisfiability checks and interpolation. Note that OpenSMT is used as a SAT solver, which gives us bit-precise reasoning[3]. FunFrog and the benchmarks used for its evaluation are available for other researchers[4].

We run FunFrog on industrial benchmarks to show that it works correctly for real-life purposes, and on artificial programs (`artN`), to stress-test the implementation. Three benchmarks are taken from the Versicec[5] suite (`verisecN`), small string manipulating programs. The most interesting benchmarks (`kbfiltrN`, `diskperfN`, `floppyN`) are taken from [18], which are three Windows device drivers, each of which contains user defined assertions. All the assertions hold, i.e., FunFrog may generate and reuse summaries.

To evaluate FunFrog, we compared it with CBMC (v3.9), SATABS (v2.4 with Cadence SMV v10-11-02p46), and CPAchecker (rev3901). SATABS and CPAchecker are CEGAR-based model checkers of C. Being a bounded model

[3] Specialized SAT solvers without proof construction generally outperform OpenSMT in the satisfiability checks though they lack the interpolant generation features.

[4] `http://verify.inf.usi.ch/funfrog`

[5] `se.cs.toronto.edu/index.php/Verisec_Suite`

benchmark			FunFrog		CBMC	SATABS	CPAchecker
name	#assert	LoC	total time	itp. time			
`verisec1`	2	63	0.020	0.002	0.003	1.004	2.851
`verisec2`	2	101	0.515	0.005	0.016	0.003	0.896
`verisec3`	2	81	0.093	0.004	0.011	TO	1.91
`art1`	2	242	1.731	0.034	0.280	534.8	65.37
`art2`	2	63	3.327	0.030	0.408	881.2	WA
`art3`	4	120	1.811	0.076	2.112	TO	WA
`kbfiltr1`	8	12253	6.718	0.003	5.742	106.457	WA
`kbfiltr2`	5	12253	2.665	0.008	3.702	13.002	WA
`diskperf1`	9	6321	5.284	0.037	20.309	433.74	15.045
`diskperf2`	4	6321	43.486	2.005	11.620	1064.2	24.849
`floppy1`	2	10259	2.196	0.001	18.028	2735.4	15.246
`floppy2`	4	10259	2.283	0.003	53.801	1402.1	47.891
`floppy3`	11	10259	45.073	0.006	99.512	2208.5	97.436

Table 1: Verification times (sec.) of FunFrog, CBMC, SATABS, and CPAchecker. Number of assertions and lines of code in the benchmarks and interpolation time for FunFrog are shown. WA: wrong answer, TO: 1 hour timeout exceeded

checker, CBMC is the closest tool to compare with. We used the same bounds for FunFrog and CBMC, sufficient to traverse the state space of the benchmarks. In order to make the results comparable, we manually unwound the benchmarks to represent the same verification task for SATABS and CPAchecker.

We expected reusability of the interpolation-based function summaries to be sensitive to the mutual relevance of the assertions in the code. Therefore, for the large benchmarks (`kbfiltrN`, `diskperfN`, `floppyN`), we experimented with multiple groups of assertions with a different level of mutual relevance, ignoring the other assertions.

The experimental results are captured in Table 1. The timings are in seconds and denote the whole verification process[6]. FunFrog performs very favorably on the larger benchmarks as it outperforms all other tools. In some cases, it outperforms CBMC, the second best tool, by an order of magnitude. However, as may be expected, the benefit is not general for all assertions. Clearly, when a set of unrelated assertions is checked, the generated function summaries are not reusable (see `diskperf2`). In this case, a number of refinement steps is needed to construct a precise approximation (which is close to the supreme substitution scenario). Since CBMC creates the full BMC formula right away without the iterative refinement, it outperforms FunFrog on this benchmark. Even in this case, running time of FunFrog is still comparable to the other tools.

As expected, FunFrog is less competitive on the small benchmarks, as it, for example, is outperformed by CBMC. We identified several reasons for this. First, there is a rather small number of function calls in these benchmarks. Thus the

[6] On some benchmarks, the simplified handling of pointers and the known implementation bug prevent CPAchecker from producing the correct results. We reported the problems to the developers of CPAchecker.

benefit of function summarization is smaller compared to the overhead of using a slower solver and the extra work on partitioning the formula. Second, CBMC can prove trivially holding assertions using only constant propagation, which is currently not implemented in FunFrog.

Notably, the overhead of our technique is small as the actual interpolation time is very low (itp. time in Table 1). Still, some cost (not measured) is hidden in the need to create an unsatisfiability proof, which hinders the solver.

Comparison of refinement strategies. Table 2 compares verification times for different combinations of refinement strategies and initial scenarios. Due to their realistic size, only the benchmarks from [18] are considered.

We note that the counter-example guided refinement strategy (noted as CEG to prevent confusion with classical CEGAR) is better or comparable to the greedy one on almost all the benchmarks. The exception (`diskperf2`) is the case where CBMC outperforms FunFrog. In this case, the number of refinement iterations is quite large due to too coarse summaries. In general, however, the number of refinement iterations needed is small. Therefore, we conclude that the counter-example guided refinement strategy performs well.

Based on the experiments, neither initial scenario is universally better. Despite winning in some cases, the lazy initial scenario performs very poor on some others. The eager initial scenario tends to perform consistently well in general, even though loosing sometimes. Therefore, the eager initial scenario combined with counter-example based refinement strategy is a safer, less volatile choice.

6 Related Work

Function summaries date back to Hoare [13]. Now it is commonly used in program analysis to achieve scalable interprocedural analysis [1,9]. Each function is processed only once, its summary is created and applied for other calls of the function. To get more fine grained summaries, multiple summaries may be created for different input conditions [9]. In BMC, state exploration of the unwound symbolically encoded program is left to a SAT/SMT solver. Thus, the program analysis approaches using fixpoint computation are not directly applicable.

Another domain of function summaries is model checking of pushdown systems (PDS). Here the most related work is [2] proposing a method to create function summaries for bounded model checking of PDS using a QBF solver. As admitted in [2], QBF queries constitute a major bottleneck. In our case, we extract multiple function summaries from a single proof of unsatisfiability of a BMC formula, which is inexpensive in comparison.

Less frequently, the idea of function summaries is used in concolic execution [10] and explicit-state model checking [20]. For example, the model checker Zing records explicit summaries as a set of tuples of explicit input and output values that were observed on an execution trace during state space traversal [20]. Summaries used in Zing also contain lock-related information necessary for checking concurrent software. In contrast, in FunFrog each summary symbolically defines an over-approximation of *all* explicit execution traces through

benchmark (assertion)		lazy/greedy		eager/greedy		lazy/CEG		eager/CEG	
		time	#RI	time	#RI	time	#RI	time	#RI
kbfiltr1	(1/5)	0.637	1	0.448	0	0.639	1	0.446	0
kbfiltr1	(2/5)	0.650	1	0.975	1	0.187	1	0.801	1
kbfiltr1	(3/5)	0.115	0	0.468	0	0.133	0	0.466	0
kbfiltr1	(4/5)	0.124	0	0.493	0	0.121	0	0.444	0
kbfiltr1	(5/5)	0.120	0	0.501	0	0.114	0	0.508	0
kbfiltr2	(1/8)	1.141	1	1.029	0	1.133	1	1.042	0
kbfiltr2	(2/8)	0.595	1	0.908	1	0.058	1	0.811	1
kbfiltr2	(3/8)	0.036	0	0.251	0	0.061	1	0.525	1
kbfiltr2	(4/8)	0.634	1	0.518	0	0.104	1	0.864	1
kbfiltr2	(5/8)	0.693	1	0.491	0	1.170	2	0.927	1
kbfiltr2	(6/8)	0.074	0	0.294	0	0.101	1	0.597	1
kbfiltr2	(7/8)	0.662	1	0.491	0	0.653	2	0.865	1
kbfiltr2	(8/8)	0.586	1	0.849	1	0.601	2	1.087	2
diskperf1	(1/9)	1.686	1	1.493	0	1.727	1	1.462	0
diskperf1	(2/9)	0.061	0	0.370	0	0.055	0	0.372	0
diskperf1	(3/9)	0.060	0	0.379	0	0.055	0	0.369	0
diskperf1	(4/9)	0.031	0	1.287	0	0.028	0	0.252	0
diskperf1	(5/9)	0.044	0	1.222	0	0.034	0	1.310	0
diskperf1	(6/9)	0.060	0	0.343	0	0.065	0	0.335	0
diskperf1	(7/9)	1.545	1	0.523	0	1.544	2	0.535	0
diskperf1	(8/9)	0.075	0	0.385	0	0.073	0	0.376	0
diskperf1	(9/9)	0.030	0	0.267	0	0.029	0	0.273	0
diskperf2	(1/4)	6.917	1	1.110	0	7.032	1	1.129	0
diskperf2	(2/4)	5.397	1	6.007	1	8.631	11	6.075	1
diskperf2	(3/4)	5.660	1	5.267	1	9.630	11	20.025	10
diskperf2	(4/4)	8.910	1	9.084	1	13.713	12	16.257	10
floppy1	(1/4)	0.284	0	1.346	0	0.285	0	1.334	0
floppy1	(2/4)	0.281	0	0.472	0	0.287	0	0.477	0
floppy1	(3/4)	149.401	1	0.485	0	151.163	1	0.509	0
floppy1	(4/4)	31.510	1	0.504	0	31.017	1	0.503	0
floppy2	(1/2)	155.741	1	1.088	0	154.174	1	1.080	0
floppy2	(2/2)	161.395	1	1.111	0	39.875	1	1.116	0
floppy3	(1/11)	160.323	1	6.549	0	161.108	1	6.508	0
floppy3	(2/11)	159.470	1	148.587	1	39.213	1	12.956	1
floppy3	(3/11)	144.195	1	1.747	0	9.346	1	2.238	0
floppy3	(4/11)	160.492	1	1.739	0	9.345	1	2.254	0
floppy3	(5/11)	0.328	0	1.904	0	9.355	1	2.354	0
floppy3	(6/11)	0.312	0	1.835	0	0.294	0	2.339	0
floppy3	(7/11)	0.313	0	1.870	0	0.291	0	2.393	0
floppy3	(8/11)	0.320	0	2.716	0	9.401	1	3.452	0
floppy3	(9/11)	166.772	1	2.877	0	9.402	1	4.246	0
floppy3	(10/11)	0.313	0	2.840	0	9.493	1	4.180	0
floppy3	(11/11)	0.149	0	1.537	0	2.335	1	2.153	0

Table 2: Verification times (sec.) and refinement iterations (RI) per assertion

a function, but currently without concurrency related data.

Craig interpolation [8] is commonly used as a means of abstraction in model checking [16]. It was used to speed up convergence of BMC by iterative over-approximation of transition relation [15]. In the scope of predicate abstraction, interpolation was used [12] to derive new predicates in the abstraction refinement phase of CEGAR-based tools (e.g., Blast [3], CPAchecker [4]). In these tools, sets of predicates are extracted from interpolants of the formulas corresponding to a prefix and suffix of a spurious error trace. This results in predicates associated to program locations along the spurious error trace yielding a more fine grained abstraction [12]. The authors also propose reordering of a path formula to generate interpolants with local variables suitable for inter-procedural analysis. Focusing on predicates, these works ignore the boolean structure of interpolants. Others observed that interpolation can be directly used to create an inductive sequence of interpolants [17,21]. The authors of [17] envision interpolation-based function summaries that would be derived from a single spurious error trace. Unfortunately, the idea is not further refined. In comparison, our function summaries are derived from the whole BMC formula and thus they contain information about all the paths through the function. Moreover, we provide a refinement strategy to deal with too weak summaries. In [11], the authors extend the idea of finding an inductive sequence of interpolants [17] to programs with function calls and recursion. The work does not refine the idea of function summaries in any way.

Lazy annotation [18] also uses function summaries. It extends symbolic execution to remember a reason for infeasibility of an execution path, i.e., a blocking annotation. Blocking annotations are used to reject other execution paths as early as possible. Compared to our technique, lazy annotation uses interpolation to derive and propagate the blocking annotations backwards for every program instruction. If the annotation is to be propagated across a function call, a function summary merging blocking annotations from all paths through the function is generated and stored for a later use. Our technique uses interpolation on the whole BMC formula and creates one function summary from one interpolant.

7 Conclusion

This paper presented a new technique to speed up BMC by means of extracting and reusing over-approximating function summaries. Our function summaries are extracted from a successful verification run using Craig interpolation which symbolically captures all execution traces through the function. We provided a counter-example guided refinement strategy to automatically refine spurious behaviors which are possible due to over-approximation. The new approach was implemented in our tool FunFrog whose application to various benchmarks demonstrated feasibility and advantages of our approach. Although the presented technique is not strictly limited to BMC, it requires combining with another technique for dealing with loops and recursion. We are therefore investigating possible connection with an engine for loop invariant detection, e.g., the one used in LoopFrog [14]. Nevertheless, we restrict the presentation in this paper

to the BMC case that is also covered by the implementation in FunFrog. Another restriction of our technique is that it is defined for sequential programs. We believe that the generated summaries may be extended by a locking related information in a similar way as in [20]. However, this is left for a future work.

References

1. Babic, D., Hu, A.J.: Calysto: scalable and precise extended static checking. In: Int. Conference on Software Engineering (ICSE '08). pp. 211–220. ACM (2008)
2. Basler, G., Kroening, D., Weissenbacher, G.: SAT-based summarization for Boolean programs. In: SPIN Workshop (SPIN '07). pp. 131–148. LNCS (2007)
3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast: Applications to software engineering. Int. J. STTT 9, 505–525 (2007)
4. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Computer Aided Verification (CAV '11). pp. 184–190. LNCS (2011)
5. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Tools and Alg. for Con. and Anal. of Sys. (TACAS '10). pp. 150–153. LNCS (2010)
6. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Alg. for Con. and Anal. of Sys. (TACAS '04). pp. 168–176. LNCS (2004)
7. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based Predicate Abstraction for ANSI-C. In: Tools and Alg. for Con. and Anal. of Sys. (TACAS '05). pp. 570–574. LNCS (2005)
8. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. of Symbolic Logic pp. 269–285 (1957)
9. Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: Symposium on OS Principles (SOSP '03). pp. 237–252. ACM (2003)
10. Godefroid, P.: Compositional dynamic test generation. In: Principles of Prog. Languages (POPL '07). pp. 47–54. ACM (2007)
11. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Principles of Prog. Languages (POPL '10). pp. 471–482. ACM (2010)
12. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Principles of Prog. Languages (POPL '04). pp. 232–244. ACM (2004)
13. Hoare, C.: Procedures and parameters: An axiomatic approach. Symposium on Semantics of Algorithmic Languages pp. 102–116 (1971)
14. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loopfrog: A Static Analyzer for ANSI-C Programs. In: Automated Software Engineering (ASE '09). pp. 668–670. IEEE (2009)
15. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Computer Aided Verification (CAV '03). pp. 1–13. LNCS (2003)
16. McMillan, K.L.: Applications of Craig Interpolation in Model Checking. In: Tools and Alg. for Con. and Anal. of Sys. (TACAS '05). pp. 1–12. LNCS (2005)
17. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification (CAV '06). pp. 123–136. LNCS (2006)
18. McMillan, K.L.: Lazy annotation for program testing and verification. In: Computer Aided Verification (CAV' 10). pp. 104–118. LNCS (2010)
19. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic 62(3), 981–998 (1997)
20. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs. In: Principles of Prog. Languages (POPL '04). pp. 245–255. ACM (2004)
21. Weissenbacher, G.: Program analysis with interpolants. PhD thesis, Oxford (2010)