Int J Softw Tools Technol Transfer (2017) 19:517–534
DOI 10.1007/s10009-015-0405-y

TACAS 2013

Flexible SAT-based framework for incremental bounded upgrade checking

Grigory Fedyukovich[1] · Ondrej Sery[1,2] · Natasha Sharygina[1]

Published online: 20 October 2015

Abstract Software undergoes a myriad of minor changes along its lifecycle. Each evolved transformation of a program is expected to preserve important correctness and security properties, in particular confirmed by a software model checking tool. However, it may be extremely resource- and time-consuming to repeat entire model checking for each new version of the program. As a possible solution to this problem, we propose to conduct incremental analysis of a new program version by reusing efforts of bounded model checking of the previous program version. Our approach maintains over-approximations of the bounded program behaviors by means of function summaries derived using Craig interpolation. For each new version, these summaries are used to localize the scope of model checking. A benefit of this approach is that the cost of the upgrade checking depends on the change impact between the two versions. If the change impact is relatively small, then the incremental check can drastically outperform the model checking the new program version from scratch. We implemented the approach in scope of the SAT-based bounded model checker for C, eVolCheck. The evaluation of eVolCheck confirms that incremental changes can be verified efficiently for different classes of industrial programs.

✉ Grigory Fedyukovich
grigory.fedyukovich@gmail.com

Ondrej Sery
ondrej.sery@d3s.mff.cuni.cz

Natasha Sharygina
natasha.sharygina@usi.ch

1 Formal Verification Lab of the Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland

2 D3S, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

1 Introduction

To achieve stability, software must pass through a long chain of updates and bug fixes. Each new version is developed incrementally due to numerous reasons: (1) requirements change and have impact on the design and implementation; (2) errors are often discovered late in the design cycle and must be fixed; (3) software components are updated or substituted to adapt to architectural and requirement changes; just to name a few. An example of a program change is illustrated on Fig. 1. The increment operation is lifted from one function to another one.

The state of affairs is that each new version of a given program needs to preserve important safety and security properties. However, even a small change can have a profound impact on the program behavior, triggering an expensive re-verification of the whole program. In this article, we address the problem of efficient analysis a program after a change in symbolic verification techniques such as bounded model checking (BMC) [4]. There is a clear need for a technique that focuses on the incremental changes and takes advantage of the efforts already invested in the model checking of previous versions. The target of the approach is to avoid, when possible, re-analyzing the new program version from scratch and to reduce analysis only to the parts of the program which were affected by the change.

Bounded model checking has proven particularly successful in safety analysis of software and has been implemented in several tools, including CBMC [8], LLBMC [31], VeriSoft [25], and FunFrog [37]. While BMC assumes a loop-free approximation of the program, there are several recent techniques for transforming programs into loop-free programs which, if successful, do not sacrifice soundness or completeness of the verification results. Examples of such techniques include unwinding assertions [8], automatic

```
int g(int a, int b)
{
  if (a < b)
    return a;
  return a - b + 1;
}

int f (int a, int b
{
  return g (a, b);
}

main()
{
  int x = 0;
  int y = nondet();
  int z = nondet();

  if (y > 0)
    x = f(y, z);

  assert(x > 0);
}
```

(a) Original program

```
int g(int a, int b)
{
  if (a < b)
    return a;
  return a - b;
}

int f (int a, int b)
{
  return g (a, b) + 1;
}

main()
{
  int x = 0;
  int y = nondet();
  int z = nondet();

  if (y > 0)
    x = f(y, z);

  assert(x > 0);
}
```

(b) Modified program

Fig. 1 Two versions of the C program

detection of recursion depth [19], k-induction [14], and loop summarization [27]. While we believe that these techniques are to some extent compatible with the methods being discussed in this article, we leave this study for future work.

This article presents an incremental SAT-based BMC approach that uses function summarizations for local upgrade checks. It receives two versions of a program, an old and a new one, and a bound to be used to unwind the loops and recursive function calls in both program versions. We assume, both versions share the same set of assertions to be checked. Given that the old version satisfy the given assertions up to the predefined bound, the goal of the approach is to verify that the assertions hold in the new version as well.

The upgrade checking algorithm maintains so-called *summaries* that, in our case, over-approximate the bounded behavior of the functions, computed by means of Craig interpolation [39]. The core idea of the algorithm is to check if the old function summaries still over-approximate the bounded behavior of the corresponding functions in the new program version. If the check fails for some function call, it needs to be propagated by the calltree traversal to the caller of the function. If the check fails for the root of the calltree (i.e., the "main" function of the program), an error trace is computed and reported to the user as a witness to the violation of one of the given assertions. On the other side, if for each function call there exists an ancestor function with the valid old summary, then the new program version is safe up to the predefined bound. Finally, for functions whose old summaries are not valid any longer, the new summaries are synthesized using Craig interpolation.

The upgrade checking algorithm implements the refinement strategy for dealing with spurious behaviors that can be introduced during computation of the over-approximated summaries. The refinement procedure for upgrade checks builds on ideas of using various summary substitution scenarios [39]. We further extend it to simplify the summary validity checks by substituting the summaries of nested function calls which are already proven valid. Failures of such checks may be due to the use of summaries which are not accurate enough. In such case, the refinement is used to expand the involved function calls on demand.

We implemented the algorithm in an upgrade checker eVolCheck that uses the SAT solver and interpolator PeRIPLO [34]. We exploit the variety of settings for interpolation, incorporated in PeRIPLO, and make them accessible from the eVolCheck running interface. It allows not only achieving the primary BMC goals, but also studying an effect of different interpolants on the different classes of programs being analyzed. We evaluated this *flexible framework* on a set of industrial benchmarks. Our experimentation confirms that the incremental analysis of upgrades is often orders of magnitude faster than the analysis performed from scratch within the same unwinding bound.

In summary, the contributions of the article are as follows:

- It presents a fully automated technique for bounded model checking incremental upgrades. It is able to re-check all previously established properties and to detect newly introduced errors.
- It combines BMC with function summarization for *local* and *incremental* analysis of changes. The use of Craig interpolation to compute summaries allows capturing symbolically all bounded execution traces through the function and, together with the local per-function checks of the new algorithm, often results in a more efficient analysis than re-checking the code from scratch.
- It presents the tool eVolCheck that implements the upgrade checking algorithm and employs the flexibility of the interpolation algorithms provided by the underlying tool PeRIPLO. We report on successful evaluation of eVolCheck on industrial benchmarks.

Our approach relies on the following assumptions.

- Both old and new versions of the given program should be unrolled to the same number n of loop iterations and recursion depth. In this case, the approach guarantees to find a real counter-example (or prove its absence) in the resulted unrolling. There is no guarantee that an error, appeared in the program unrolled to the number $m > n$ will be found.
- The algorithm relies on the fact that the unrolled calltrees of both program versions are the same. Furthermore, each pair of function calls corresponding to the same node in the calltrees should have the same signature. These

requirement can, however, be relaxed if an extra matching of the function's input/output variables in the two versions is provided.
- The approach requires the interpolation algorithm to have a specific *tree-interpolation* property (to be defined in Sect. 2). As shown in [36], only interpolation algorithms that generate interpolants at least as strong as those given by Pudlák's algorithm [33] are guaranteed to satisfy this property.

The rest of the article is organized as follows. Section 2 defines the notation and presents background on function summarization in BMC (that originally appeared in [39]). Section 3 describes and justifies the upgrade checking algorithm (proposed in [38] and extended with the preprocessing step). Section 4 describes implementation of EVOLCHECK (that differs from the original one [18] by the use of PERIPLO). Section 5 presents new evaluation of EVOLCHECK and PERIPLO. Finally, Sect. 6 discusses the related work and Sect. 7 concludes the article.

2 Background

Craig interpolation [12] is a well-known abstraction technique widely used in Model Checking.

Definition 1 Given formulas A and B such that $A \wedge B$ is unsatisfiable, *Craig interpolant* of (A, B) is a formula I such that $A \implies I$, $I \wedge B$ is unsatisfiable, and I is defined over common alphabet to A and B.

For a pair of propositional quantifier-free formulas (A, B), such that $A \wedge B$ is unsatisfiable (denoted as $A \wedge B \implies$ *false*), an interpolant always exists [33]. A commonly used framework for computing the interpolant from a given refutation proof is the *labeled interpolation system* (LIS) [15], a generalization of several interpolation algorithms (including *Pudlák's algorithm*) parameterized by a *labeling function*. Given a labeling function and a proof, LIS uniquely determines the interpolant.

Classical interpolation can be generalized so the partitions of an unsatisfiable formula naturally correspond to a tree structure. Let $\Phi = \bigwedge_{i=1}^{n} \phi_i$. For $K \subseteq \{1 \ldots n\}$, a partitioning of Φ with respect to K allows representing Φ in the following form: $\Phi \equiv \Phi_K \wedge \overline{\Phi_K}$, where $\Phi_K = \bigwedge_{k \in K} \phi_k$ and $\overline{\Phi_K} = \bigwedge_{\ell \notin K} \phi_\ell$. An *interpolation system* is a function that, given Φ and K, returns an interpolant $I_{\Phi,K}$, a formula implied by Φ_K, inconsistent with $\overline{\Phi_K}$ and defined over the common language of Φ_K and $\overline{\Phi_K}$.

Now consider a tree $T = (V, E)$ with n nodes $V = \{1, \ldots, n\}$ imposed on Φ. Formally, a sequence of n interpolation systems has the T-tree interpolation property [23] iff for any i, $\bigwedge_{(i,j)\in E} I_{\Phi,K_j} \wedge \phi_i \implies I_{\Phi,K_i}$, where $K_i = \{j \mid i \sqsubseteq j\}$ and $i \sqsubseteq j$ iff node j is a descendant of node i in T. Notice that for the *root* node in T, $K_{root} = V$ and $I_{\Phi,V} = \mathit{false}$.

Interpolants are useful in various verification domains including refinement of predicate abstraction [24], and unbounded model checking [29] to name a few. The following outlines how interpolation is used for function summarization in BMC [39].

Definition 2 An *unwound program* for a bound ν is a tuple $P_\nu = (F, f_{main})$, such that F is a finite set of functions, $f_{main} \in F$ is an entry point and every loop and recursive call is unrolled (unwound) ν times.

In addition, we define a relation *child* $\subseteq F \times F$ which relates each function f to all the functions invoked by f. Relation *subtree* $\subseteq F \times F$ is a reflexive transitive closure of *child*. The set $\hat{F}$ denotes the finite set of unique function calls, with $\hat{f}_{main}$ being the implicit call to the program entry point. The relations *child* and *subtree* are naturally extended to $\hat{F}$, such that $\forall \hat{f}, \hat{g} \in \hat{F} : child(\hat{f}, \hat{g}) \implies child(f, g)$, and *subtree* is a reflexive transitive closure of the extended relation *child*.

Standard software bounded model checkers encode an unwound program to a special kind of logical formula, called BMC formula. Although the detailed construction of the BMC formula can be found in [8], we illustrate it on an example in Fig. 2. It encodes the original program from

```
x0 = 0;
y0 = nondet();
z0 = nondet();
if (y0 > 0) {
  f_a0 = y0;
  f_b0 = z0;

  // inline f
  g_a0 = f_a0;
  g_b1 = f_b0;

  // inline g
  if (g_a0 < g_b0)
    g_ret0 = g_a0;
  else
    g_ret1 = g_a0 - g_b0  + 1;
  g_ret2 = phi(g_ret0, g_ret1);
  // end inline g;

  f_ret0 = g_ret2;
  // end inline f

  x1 = f_ret0;
}
x2 = phi(x0, x1);
assert(x2 >= 0);
```

(a)

$$
\begin{aligned}
& x_0 = 0 \wedge \\
& y_0 = nondet_0 \wedge \\
& z_0 = nondet_1 \wedge \\
& f_{a_0} = y_0 \wedge \\
& f_{b_0} = z_0 \wedge \\
& g_{a_0} = f_{a_0} \wedge \\
& g_{b_0} = g_{b_0} \wedge \\
& g_{ret_0} = g_{a_0} \wedge \\
& g_{ret_1} = g_{a_0} - g_{b_0} + 1 \wedge \\
& (y_0 > 0 \wedge g_{a_0} < g_{b_0} \implies \\
& \quad g_{ret_2} = g_{ret_0}) \wedge \\
& (y_0 > 0 \wedge g_{a_0} \geq g_{b_0} \implies \\
& \quad g_{ret_2} = g_{ret_1}) \wedge \\
& f_{ret_0} = g_{ret_2} \wedge \\
& x_1 = f_{ret_0} \wedge \\
& (y_0 > 0 \implies x_2 = x_1) \wedge \\
& (y_0 \leq 0 \implies x_2 = x_0) \wedge \\
& x_2 < 0
\end{aligned}
$$

(b)

Fig. 2 BMC formula generation. **a** SSA form. **b** BMC formula

Fig. 1a. First, the unwound program is converted into the static single assignment (SSA) [13] form (Fig. 2a), where each variable is assigned at most once. A so-called phi-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. 2b) is constructed from the SSA form. Assignments are converted to equalities, path conditions are computed from branching conditions and used to encode phi-functions. Negation of the assertion expression guarded by its path condition (*true* in this case) is conjuncted with the BMC formula. If the resulting BMC formula is unsatisfiable then the assertion holds (for unwinding up to ν). In the other case, a satisfying assignment identifies an error trace.

2.1 Function summaries

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 $\mathbb{D}$ that assigns a domain (i.e., set of possible values) to every variable from $\mathbb{V}$.

Definition 3 For a function f, sequences of variables $args^f_{in} = \langle in_1, \dots, in_m \rangle$ and $args^f_{out} = \langle out_1, \dots, out_n \rangle$ 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, $args^f = \langle in_1, \dots, in_m, out_1, \dots, out_n \rangle$ denotes all the arguments of f. As a shortcut, we use $\mathbb{D}(f) = \mathbb{D}(in_1) \times \cdots \times \mathbb{D}(in_m) \times \mathbb{D}(out_1) \times \cdots \times \mathbb{D}(out_n)$.

In the following, we expect that an in–out argument (e.g., a parameter passed by reference) 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 all the tuples $\bar{v} = \langle v_1, \dots, v_{|args^f|} \rangle$ such that the function f called with the input values $\langle v_1, \dots, v_{|args^f_{in}|} \rangle$ can finish with the output values $\langle v_{|args^f_{in}|+1}, \dots, v_{|args^f|} \rangle$.

Note that Definition 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 Definition 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 all 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 up to ν.

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

Definition 7 (*Summary formula*) Let f be a function, ν a bound, σ_f a formula with free variables only from $args^f$, and S a relation induced by σ_f as $S = \{\bar{v} \in \mathbb{D}(f) \mid \sigma_f[\bar{v}/args^f] \models true\}$. If S is a summary of the function f and bound ν, then σ_f 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 corresponding to the subtree of a called function does not have to be created and converted to a part of the BMC formula. Moreover, the summary formula tends to be more compact. Considering the example in Fig. 2, using the summary formula $f_{a_0} > 0 \implies f_{ret_0} > 0$ for the function `f` yields the BMC formula in Fig. 3.

The important property of the resulting BMC formula is that if it is unsatisfiable (as in Fig. 3) then also the formula without summaries (as in Fig. 2b) 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. If ϕ' is unsatisfiable then ϕ is unsatisfiable as well.*

2.2 Interpolation-based summaries

Among different possible ways to obtain a summary formula, we consider 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 body of the function call $\hat{f}$. Moreover, the

$$\begin{aligned}
&x_0 = 0 \wedge\\
&y_0 = nondet_0 \wedge\\
&z_0 = nondet_1 \wedge\\
&f_{a_0} = y_0 \wedge\\
&f_{b_0} = z_0 \wedge\\
&\boldsymbol{(f_{a_0} > 0 \implies f_{ret_0} > 0)} \wedge\\
&x_1 = f_{ret_0} \wedge\\
&(y_0 > 0 \implies x_2 = x_1) \wedge\\
&(y_0 \leq 0 \implies x_2 = x_0) \wedge\\
&x_2 < 0
\end{aligned}$$

Fig. 3 BMC formula created after substituting the summary $f_{a_0} > 0 \implies f_{ret_0} > 0$ for function f

$$\begin{aligned}
&x_0 = 0 \wedge\\
&y_0 = nondet_0 \wedge\\
&z_0 = nondet_1 \wedge\\
&f_{a_0} = y_0 \wedge\\
&f_{b_0} = z_0 \wedge\\
&(y_0 > 0 \qquad callstart_{\hat{f}}) \wedge\\
&x_1 = f_{ret_0} \wedge\\
&(y_0 > 0 \implies x_2 = x_1) \wedge\\
&(y_0 \leq 0 \implies x_2 = x_0) \wedge\\
&(x_2 < 0 \qquad error_{\hat{f}_{main}})
\end{aligned}$$

(a)

$$\begin{aligned}
&g_{a_0} = f_{a_0} \wedge\\
&g_{b_0} = f_{b_0} \wedge\\
&(callstart_{\hat{f}} \qquad callstart_{\hat{g}}) \wedge\\
&(callstart_{\hat{f}} \implies f_{ret_0} = g_{ret_0})\\
&\\
&g_{ret_1} = g_{a_0} \wedge\\
&g_{ret_2} = g_{a_0} - g_{b_0} + 1 \wedge\\
&(callstart_{\hat{g}} \wedge g_{a_0} < g_{b_0} \implies\\
&\quad g_{ret_0} = g_{ret_1}) \wedge\\
&(callstart_{\hat{g}} \wedge g_{a_0} \geq g_{b_0} \implies\\
&\quad g_{ret_0} = g_{ret_2})
\end{aligned}$$

(b)

Fig. 4 Partitioned bounded model checking formula. **a** formula $\phi_{\hat{f}_{main}}$. **b** formulas $\phi_{\hat{f}}$ and $\phi_{\hat{g}}$

symbols of $\phi_{\hat{f}}$ shared with the rest of the formula correspond only to the input and output program variables.

The formula in classical BMC is generally constructed monolithically: all the function calls are inlined, and the variables from the calling context tend to leak into the formulas of the called function as a part of the path condition. For example in Fig. 2b, the variable y_0 from the calling context of the function f appears in the encoding of the body of the function g, called inside f. To achieve the desired form, we generate the parts of the formula corresponding to the individual functions in separation and bind them together using a boolean variable $callstart_{\hat{f}}$ for every function call. Intuitively, $callstart_{\hat{f}}$ evaluates to *true* iff the corresponding function call $\hat{f}$ is reached. Another special variable is $error_{\hat{f}}$, which is constrained to be *true* iff the function call $\hat{f}$ results in an error. Consequently, $error_{\hat{f}_{main}}$ encodes reachability of an error in the entire program. We call the resulting formula a *partitioned bounded model checking* (PBMC) formula.

Creation of a PBMC formula for the example from Fig. 2 is shown on Fig. 4. When the functions f and g are called, the corresponding variables $callstart_{\hat{f}}$ and $callstart_{\hat{g}}$ are *true*. Therefore, the formula of the calling context (Fig. 4a) makes it equivalent to the path condition of the call.

If the resulting PBMC formula is unsatisfiable, we compute multiple Craig interpolants from a single proof of unsatisfiability to get the set of function summaries. The details of the summarization procedure are given in Sect. 2.3.

2.3 Summarization algorithm

Algorithm 1 outlines the method for constructing function summaries in BMC.

PBMC formula construction (line 1).

The PBMC formula is created in the recursive method `CreateFormula` as follows.

$$\texttt{CreateFormula}(\hat{f}) \triangleq \phi_{\hat{f}} \wedge \bigwedge_{\hat{g}\in\hat{F}:child(\hat{f},\hat{g})} \texttt{CreateFormula}(\hat{g})$$

For a function call $\hat{f} \in \hat{F}$, the formula is constructed by conjunction of the partition $\phi_{\hat{f}}$ reflecting the body of the function and a separate partition for every nested function call. The logical formula $\phi_{\hat{f}}$ is constructed from the SSA form of the body of the function f. The bodies of the nested calls are encoded into separate logical formulas (using a recursive call to `CreateFormula`) and thus separate partitions in the resulting PBMC formula.

Summarization (line 6). If the PBMC formula is unsatisfiable, i.e., the program is safe, the algorithm proceeds with interpolation. The function summaries are constructed as interpolants from a proof of unsatisfiability of the PBMC formula. In order to generate the interpolant, for each function call $\hat{f}$ the PBMC formula is split into two parts. First, $\phi^{subtree}_{\hat{f}}$ corresponds to the partitions representing the function call $\hat{f}$ and all the nested functions. Second, $\phi^{env}_{\hat{f}}$ corresponds to the context of the call $\hat{f}$, i.e., to the rest of the encoded program.

$$\phi^{subtree}_{\hat{f}} \triangleq \bigwedge_{\hat{g}\in\hat{F}:subtree(\hat{f},\hat{g})} \phi_{\hat{g}}$$

$$\phi^{env}_{\hat{f}} \triangleq error_{\hat{f}_{main}} \wedge \bigwedge_{\hat{h}\in\hat{F}:\neg subtree(\hat{f},\hat{h})} \phi_{\hat{h}}$$

Therefore, for each function call $\hat{f}$, the `Interpolate` method splits the PBMC formula into $A \equiv \phi^{subtree}_{\hat{f}}$ and $B \equiv \phi^{env}_{\hat{f}}$ and synthesizes an interpolant $I_{\hat{f}}$ for (A, B), such that $I_{\hat{f}}$ satisfies Definition 1. Throughout the article, we will refer to $I_{\hat{f}}$ as a *summary* of function call $\hat{f}$.

The generated interpolants are associated with the function calls by a mapping $\sigma : \hat{F} \implies \mathbb{S}$, i.e., $\sigma(\hat{f}) = I_{\hat{f}}$.

Algorithm 1: Function summarization in BMC

```
Input: Unwound program: $P_\nu = (F, f_{main})$ with function calls $\hat{F}$
Output: Verification result: {SAFE, UNSAFE}, summaries
        mapping: $\sigma$
Data: PBMC formula: $\phi$, refutation: proof
1 $\phi \leftarrow$ CreateFormula($\hat{f}_{main}$) $\wedge$ $error_{\hat{f}_{main}}$;
2 result, proof $\leftarrow$ Solve($\phi$) ;            // run SAT solver
3 if result = SAT then
4     return UNSAFE, $\varnothing$;
5 foreach $\hat{f} \in \hat{F}$ do                 // extract summaries
6     $\sigma(\hat{f}) \leftarrow$ Interpolate(proof, $\hat{f}$);
7 return SAFE, $\sigma$;
```

Example Assume the program on Fig. 1a is verified by Algorithm 1. It is encoded into the PBMC formula (Fig. 4) and the PBMC formula is bit-blasted and passed to a SAT solver. After the SAT solver proved unsatisfiability, the proof is used to create the following summaries of the functions `main`, `f`, `g`:

$$\sigma(\texttt{main}) = x_2 > 0 \tag{1}$$

$$\sigma(\texttt{f}) = (f_{a_0} > 0) \implies (f_{ret_0} > 0) \tag{2}$$

$$\sigma(\texttt{g}) = (g_{a_0} > 0) \implies (g_{ret_0} > 0) \tag{3}$$

Note that the summary of function `main` is expressed over variable x_2, the only common one between the body of the function and the assertion expression. The summaries of functions `f` and `g` are expressed over their input and output variables respectively (i.e., the common language of the function and its caller). Furthermore, formulas (2) and (3) are semantically equivalent, but syntactically different.

Refinement. When the same program is being verified again (e.g., with respect to a different assertion), the exact function calls can be substituted by the summaries constructed before. In this case, the method `CreateFormula` of Algorithm 1 is replaced by the following:

$$\texttt{CreateFormula}(\hat{f}) \triangleq \phi_{\hat{f}} \wedge \left(\bigwedge_{\hat{g} \in \hat{F} : child(\hat{f}, \hat{g}) \wedge \Omega(\hat{g}) = inline} \texttt{CreateFormula}(\hat{g}) \right) \left(\bigwedge_{\hat{g} \in \hat{F} : child(\hat{f}, \hat{g}) \wedge \Omega(\hat{g}) = sum} \sigma(\hat{g}) \right)$$

where a *substitution scenario* $\Omega : \hat{F} \implies \{inline, sum, havoc\}$ determines how each function call should be handled. Initially, Ω depends on the existence of function summaries. If a summary of a function $\hat{f}$ exists, it substitutes the body, i.e., $\Omega(\hat{f}) = sum$. If not, $\hat{f}$ is either represented precisely, i.e., $\Omega(\hat{f}) = inline$ (*eager* scenario), or abstracted away, i.e., $\Omega(\hat{f}) = havoc$ (*lazy* scenario).

If the PBMC formula constructed using Ω is satisfiable, it may be due to too coarse summaries. Refinement, first, identifies which summaries affect satisfiability of the PBMC formula. This is done by analyzing the occurrence of summaries along an error trace: the corresponding $callstart_{\hat{f}}$ variable should be evaluated to *true*. Second, the *refined* substitution scenario Ω' is constructed from Ω by assigning those function calls to *inline*. Finally, Algorithm 1 should proceed to the next iteration using Ω'. If no summary is identified for refinement, the error is real.

3 Upgrade checking

This section describes our solution to the upgrade checking problem, the incremental summary-based BMC algorithm. As an input, the algorithm takes two versions of the program, old and new, and the function summaries of the old version. If the old version or its function summaries are not available (e.g., for the new version of the program), a bootstrapping verification (Algorithm 1) run is needed to analyze the entire new version of the program and to generate the summaries, which are then maintained during the incremental runs.

The incremental upgrade check is performed in two phases. First the two versions are compared at the syntactical level. This allows identification of functions that were modified and whose summaries need rechecking (or they even do not exist yet). An additional output of this phase is an updated mapping σ, which maps function calls in the new version to the old summaries.

For example, Fig. 5a depicts an output of the preprocessing, i.e., a calltree of a new version with two changed function calls (gray fill). Their summaries need rechecking. In this case, all function calls are mapped to the corresponding old summaries (i.e., functions were possibly removed or modified, but not added). Summaries of all the function calls marked by a question mark may yet be found invalid. Although the code of the corresponding functions may be unchanged, some of their descendant functions were changed and may eventually lead to invalidation of the ancestor's summary.

In the second phase, the actual upgrade check is performed. Starting from the bottom of the calltree, summaries of all functions marked as changed are rechecked. That is, a local check is performed to show that the corresponding summary is still a valid over-approximation of the function's behavior. If successful, the summary is still valid and the change (e.g., rightmost node in Fig. 5b) does not affect correctness of the new version. If the check fails, the summary is invalid for the new version and the check needs to be propagated to the caller, towards the root of the calltree (Fig. 5b,c).

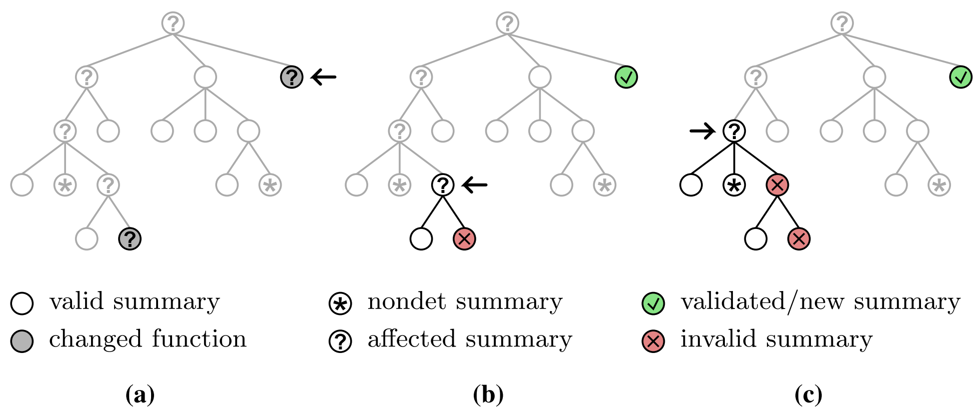


Fig. 5 Progress of the upgrade checking algorithm; the *faded parts* of the calltree were not yet analyzed by the algorithm

When the check fails for the root of the calltree (i.e., program entry point $\hat{f}_{main}$), a real error is identified and reported to the user.

In the rest of the section, we present this basic algorithm in more detail, describe its optimization with a refinement loop and prove its correctness. Note that we describe the upgrade checking algorithm instantiated in the context of BMC. However, the algorithm is more general and can be applied in other approaches relying on over-approximative function summaries.

3.1 Basic upgrade checking algorithm

We proceed by presenting the basic upgrade checking algorithm (Algorithm 2). As an input, Algorithm 2 takes the unwound program together with the old and new versions of the SSA form for each function call, and a mapping σ from the function calls in the new version to the summaries from the old version. We denote the domain of a mapping by *dom*. If $dom(\sigma) = \emptyset$, the algorithm is inapplicable, and instead, the bootstrapping (Algorithm 1) should be run. Thus, we assume that $dom(\sigma)$ contains at least a summary for $\hat{f}_{main}$.

The algorithm starts with computing a set *changed* that collects the function calls corresponding to the functions, on which the old and the new versions disagree (line 1). In our implementation, we syntactically compare the corresponding SSA forms.

The algorithm maintains a worklist *WL* of function calls that require rechecking. Initially, *WL* is populated by the elements of the previously computed set *changed* (line 2). Then the algorithm repeatedly removes a function call $\hat{f}$ from *WL* and attempts to check validity of the corresponding summary in the new version. Note that the algorithm picks $\hat{f}$ so that no function call in the subtree of $\hat{f}$ occurs in *WL* (line 5). This ensures that summaries in the subtree of $\hat{f}$ were already analyzed (shown either valid or invalid).

The actual summary validity check occurs on lines 8–9. First, the PBMC formula encoding the subtree of $\hat{f}$ (with respect to the new version of the SSA form) is constructed and stored as ϕ. Then, conjunction of ϕ with a negated summary of $\hat{f}$ is passed to a SAT solver for the satisfiability check. If unsatisfiable, the summary is still a valid over-approximation of the function's behavior. Here, the algorithm obtains a proof of unsatisfiability which is used later to create new summaries to replace the invalid or missing ones (lines 11–12). If satisfiable, the summary is not valid for the new version (line 13). In this case, either a real error is identified (lines 15–16) or the check is propagated to the function caller (line 18).

Note that if the chosen function call $\hat{f}$ has no summary, the check is propagated to the caller immediately (line 14) and the summary of $\hat{f}$ is created later when the check succeeds for some ancestor function call of $\hat{f}$.

Assuming the SAT solver returns, the algorithm always terminates with either *SAFE* or *UNSAFE* value. Creation of each PBMC formula terminates because they operate on the already unwound program. The algorithm terminates with *SAFE* result (line 19) when all function calls requiring rechecking were analyzed (line 4). Either all the summaries possibly affected by the program change are immediately shown to be still valid over-approximations (see Fig. 6a) or some are invalid but the propagation stops at a certain level of the calltree and new valid summaries are generated (see Fig. 6b). The algorithm terminates with *UNSAFE* result (line 16), when the check propagates to the calltree root, $\hat{f}_{main}$, and fails (see Fig. 6c). In this case, a real error is encountered and reported to the user.

Example As a demonstration of the upgrade checking algorithm, consider the upgraded version of the running example in Fig. 1b. It is created by lifting an increment operator one level up on the calltree (i.e, from function `g` to its caller, function `f`). Our approach first checks if $\sigma(\texttt{g})$ still over-approximates `g`.

In our example, the summary check of $\sigma(\texttt{g})$ does not succeed. As a witness of this, one can chose equal positive numbers to be the values for the function parameters. For example, if $g_{a_0} = g_{b_0}$ then $g_{ret_0} = 0$ which contradicts the formula (3).

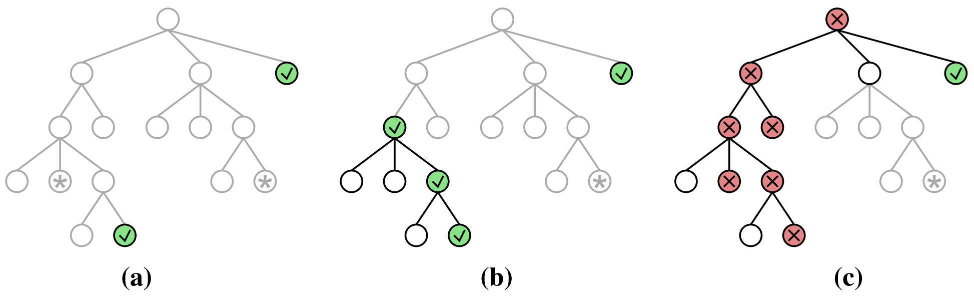


Fig. 6 Sample outcomes of Algorithm 2; analyzing the *faded parts* of calltree is not required to decide safety of the upgrade

Algorithm 2: Upgrade checking algorithm

```
Input: Unwound program: $P_v = (F, f_{main})$ with function calls $\hat{F}$, SSA forms of both versions of $P_v$: $code_{old} : \hat{F} \to$ SSA and $code_{new} : \hat{F} \to$ SSA, summaries mapping: $\sigma : \hat{F} \to \mathbb{S}$
Output: Verification result: {SAFE, UNSAFE}, actualized summaries mapping: $\sigma$
Data: temporary sets of function calls: $changed$, $WL \subseteq \hat{F}$, PBMC formula: $\phi$, set of invalid summaries: $invalid \subseteq \mathbb{S}$, refutation: $proof$
1  $changed \leftarrow \{\hat{f} \mid \hat{f} \in \hat{F}$, s.t. $code_{old}(\hat{f}) \not\equiv code_{new}(\hat{f})\}$;
2  $WL \leftarrow changed$;
3  $invalid \leftarrow \varnothing$;
4  while ($WL \neq \varnothing$) do
5      choose $\hat{f} \in WL$, s.t. $\forall \hat{g} \in WL : \neg subtree(\hat{f}, \hat{g})$;
6      $WL \leftarrow WL \setminus \{\hat{f}\}$;
7      if ($\hat{f} \in dom(\sigma)$) then
8          $\phi \leftarrow$ CreateFormula($\hat{f}$);
9          $result, proof \leftarrow$ Solve($\phi \wedge \neg\sigma(\hat{f})$);
10         if ($result$ = UNSAT) then
11             for ($\hat{g} \in \hat{F} : subtree(\hat{f}, \hat{g}) \wedge$
                       ($\hat{g} \notin dom(\sigma) \vee \sigma(\hat{g}) \in invalid$)) do
12                 $\sigma(\hat{g}) \leftarrow$ Interpolate($proof, \hat{g}$);
13         else $invalid \leftarrow invalid \cup \{\sigma(\hat{f})\}$;
14     if ($\hat{f} \notin dom(\sigma) \vee \sigma(\hat{f}) \in invalid$) then
15         if ($\hat{f} = \hat{f}_{main}$) then
16             return UNSAFE, $\varnothing$;          // real error found
17         else
18             $WL \leftarrow WL \cup \{parent(\hat{f})\}$;   // check the parent
19 return SAFE, $\sigma$;                    // new version is safe
```

Then the algorithm proceeds with checking validity of $\sigma(\mathtt{f})$ and proves it. During this check, the new behavior of $\mathtt{g}$ was encoded and its old summary (3) was not used. Given the proof of validity of $\sigma(\mathtt{f})$, we apply interpolation and update $\sigma(\mathtt{g})$ as follows:

$$\sigma(\mathtt{g}) = (g_{a_0} > 0) \implies (g_{ret_0} \geq 0) \tag{4}$$

Since there was no change in function `main`, the upgrade checking terminates. The updated summary (4) is going to be stored instead of (3) and used when another program version arrives.

Further discussion. The presented approach can also be viewed as a technique for reusing refutation proofs from one unsatisfiable propositional formula to speed up SAT checks for a slightly different formula. For instance, a large number of tools and algorithms within verification and program analysis make many quick calls to SAT solvers. A large number of these calls are often similar. However, to apply the idea of Algorithm 2 to this setting, an additional input (a tree T that describes relationships between the subformulas) should be provided. In the scope of model checking, T naturally corresponds to the program calltree, but in a general setting, it could be difficult to discover it. We leave this task for a future work.

3.2 Optimization and refinement

To optimize the upgrade check, old function summaries can be used to abstract away the function calls. Consider the validity check of a summary of a function call $\hat{f}$. Suppose there exists a function call $\hat{g}$ in the subtree of $\hat{f}$ together with its old summary, already shown valid. Then this summary can be substituted for $\hat{g}$, while constructing the PBMC formula of $\hat{f}$ (line 8). This way, only a part of the subtree of $\hat{f}$ needs to be traversed and the PBMC formula ϕ can be substantially smaller compared to the encoding of the entire subtree.

If the resulting formula is satisfiable, it can be either due to a real violation of the summary being checked or due to too coarse summaries used to substitute some of the nested function calls. In our upgrade checking algorithm, this is handled in a similar way as in the refinement of the standalone verification by analyzing the satisfying assignment. The set of summaries used along the counter-example is identified. Then it is further restricted by dependency analysis to only those possibly affecting the validity. Every summary in the set is marked as *inline* in the next iteration. If the set is empty, the check fails and the summary is shown invalid. This refinement loop (replacing lines 8–9 in Algorithm 2) iterates until validity of the summary is decided.

This optimization does not affect termination of the algorithm (in each step at least one of the summaries is refined).

Regarding complexity, in the worst case scenario, i.e., when a major change occurs, the entire subtree is refined one summary at a time for each node of the calltree. This may result in a number of SAT solver calls quadratic in the size of the calltree, where the last call is as complex as the verification of the entire program from scratch. This article focuses on incremental changes and thus for most cases there is no need for the complete calltree traversal. Moreover, the quadratic number of calls can be easily mitigated by limiting the refinement laziness using a threshold on the number of refinement steps and disabling this optimization when the threshold is exceeded.

3.3 Correctness

This section proves the correctness of the upgrade checking algorithm, i.e., given an unwinding bound ν, the algorithm always terminates with the correct answer with respect to ν. Note that throughout this section, program safety is with respect to the unwinding bound ν provided by the user.[1] Also, we use $\sigma_{\hat{f}}$ as a shortcut for $\sigma(\hat{f})$. The key insight for the correctness is that after each successful run of Algorithm 2 (i.e., when *SAFE* is returned), the following two properties are maintained.

$$error_{\hat{f}_{main}} \wedge \sigma_{\hat{f}_{main}} \implies false \tag{5}$$

Given each function call $\hat{f}$ and its children calls $\hat{g}_1, \ldots, \hat{g}_n$:

$$\sigma_{\hat{g}_1} \wedge \cdots \wedge \sigma_{\hat{g}_n} \wedge \phi_{\hat{f}} \implies \sigma_{\hat{f}} \tag{6}$$

The following theorem is required to prove the correctness of Algorithm 2. It considers the tree-interpolation property of interpolants generated from the same resolution proof using Pudlák's algorithm [33].

Theorem 1 *Let $X_1 \wedge \cdots \wedge X_n \wedge Y \wedge Z$ be an unsatisfiable formula and let $I_{X_1}, \ldots, I_{X_n}$, and I_{XY} be Craig interpolants for pairs $(X_1, X_2 \wedge \cdots \wedge X_n \wedge Y \wedge Z), \ldots, (X_n, X_1 \wedge \cdots \wedge X_{n-1} \wedge Y \wedge Z)$, and $(X_1 \wedge \cdots \wedge X_n \wedge Y, Z)$ respectively, derived using Pudlák's algorithm over a resolution proof $\mathbb{P}$. Then $(I_{X_1} \wedge \cdots \wedge I_{X_n} \wedge Y) \implies I_{XY}$.*

We will first state and prove a version of Theorem 1 limited to two partitions and then generalize.

Lemma 2 *Let $X \wedge Y \wedge Z$ be an unsatisfiable formula and let I_X, I_Y, and I_{XY} be Craig interpolants for pairs $(X, Y \wedge Z)$, $(Y, X \wedge Z)$, and $(X \wedge Y, Z)$, respectively, derived using Pudlák's algorithm over a resolution proof $\mathbb{P}$. Then $(I_X \wedge I_Y) \implies I_{XY}$.*

[1] We expect the same ν for the old and new version. To ensure correctness, if the user increases the bound for a specific loop, the corresponding function has to be handled as if modified.

Table 1 Variable classes; a, b: x occurs only in A, resp. B, ab: x occurs in both A and B

x in	Class of x for partial interpolant		
	I_X	I_Y	I_{XY}
X	a	b	a
Y	b	a	a
Z	b	b	b
$X+Y$	ab	ab	a
$X+Z$	ab	b	ab
$Y+Z$	b	ab	ab
$X+Y+Z$	ab	ab	ab

Proof By structural induction over the resolution proof, we show that $(I_X \wedge I_Y) \implies I_{XY}$ for all partial interpolants at all nodes of the proof $\mathbb{P}$. As a base case, there is a clause C and we need to consider three cases: $C \in X$, $C \in Y$, and $C \in Z$. When $C \in X$, we have $(false \wedge true) \implies false$, which holds. The case $C \in Y$ is symmetric. When $C \in Z$, we have $(true \wedge true) \implies true$, which again obviously holds.

As an inductive step, we have a node $C_1 \vee C_2$ representing resolution over a variable x with parent nodes $x \vee C_1$ and $\neg x \vee C_2$. From the inductive hypothesis, we have partial interpolants I^1_X, I^1_Y, and I^1_{XY} for the node $x \vee C_1$ so that $(I^1_X \wedge I^1_Y) \implies I^1_{XY}$ and partial interpolants I^2_X, I^2_Y, and I^2_{XY} for the node $\neg x \vee C_2$ so that $(I^2_X \wedge I^2_Y) \implies I^2_{XY}$. We need to consider the different cases of coloring of x based on its occurrence in different subsets of the parts of the formula $X \wedge Y \wedge Z$. The cases are summarized in Table 1.

In case $x \in X$, we have:

$$I_X \equiv I^1_X \vee I^2_X, \quad I_Y \equiv I^1_Y \wedge I^2_Y$$
$$I_{XY} \equiv I^1_{XY} \vee I^2_{XY}$$

Using the inductive hypothesis, we have $((I^1_X \vee I^2_X) \wedge I^1_Y \wedge I^2_Y) \implies (I^1_{XY} \vee I^2_{XY})$, which is the required claim $(I_X \wedge I_Y) \implies I_{XY}$. The case $x \in Y$ is symmetric.

In case $x \in Z$, we have:

$$I_X \equiv I^1_X \wedge I^2_X, \quad I_Y \equiv I^1_Y \wedge I^2_Y$$
$$I_{XY} \equiv I^1_{XY} \wedge I^2_{XY}$$

Using the inductive hypothesis, we have $(I^1_X \wedge I^2_X \wedge I^1_Y \wedge I^2_Y) \implies (I^1_{XY} \wedge I^2_{XY})$, which is the required claim $(I_X \wedge I_Y) \implies I_{XY}$.

In case $x \in X+Y+Z$, using $sel(x, S, T)$ as a shortcut for $(x \vee S) \wedge (\neg x \vee T)$, we get:

$$I_X \equiv sel(x, I^1_X, I^2_X), \quad I_Y \equiv sel(x, I^1_Y, I^2_Y)$$
$$I_{XY} \equiv sel(x, I^1_{XY}, I^2_{XY})$$

Using the inductive hypothesis and considering both possible values of x, we have $(sel(x, I^1_X, I^2_X) \wedge sel(x, I^1_Y, I^2_Y)) \implies sel(x, I^1_{XY}, I^2_{XY})$, which is the required claim $(I_X \wedge I_Y) \implies I_{XY}$. The other cases where $x \in X + Y$ or $x \in X + Z$ or $x \in Y + Z$ are subsumed by this case as $(P \wedge Q) \implies sel(x, P, Q) \implies (P \vee Q)$. Structural induction yields $(I_X \wedge I_Y) \implies I_{XY}$ for the root of the proof tree and for the final interpolants. □

When we apply the result of Lemma 2 iteratively, we obtain a generalized form for cases using multiple interpolants mixed with original parts of the formula, i.e., a proof of Theorem 1.

Proof By iterative application of Lemma 2, we get $(I_{X_1} \wedge \cdots \wedge I_{X_n} \wedge I_Y) \implies I_{XY}$, where I_Y is Craig interpolant for the pair $(Y, X_1 \wedge \cdots \wedge X_n \wedge Z)$ derived using Pudlák's algorithm over the resolution proof $\mathbb{P}$. Using $Y \implies I_Y$, we obtain the claim $(I_{X_1} \wedge \cdots \wedge I_{X_n} \wedge Y) \implies I_{XY}$. □

In the following two lemmas, we first show that properties (5, 6) hold after an initial whole-program check. Then we show that the properties are maintained between individual successful upgrade checks.

Lemma 3 *After an initial whole-program check, the properties (5, 6) hold over the calltree annotated by the generated interpolants.*

Proof Recall that the summaries are constructed only when the program is safe. In other words, $error_{\hat{f}_{main}} \wedge \phi^{subtree}_{\hat{f}_{main}} \implies false$. Thus, by definition of interpolation, $error_{\hat{f}_{main}} \wedge I_{\hat{f}_{main}}$ is obviously unsatisfiable, i.e., the property (5) holds. The property (6) follows from Theorem 1. It suffices to choose $X_i \equiv \phi^{subtree}_{\hat{g}_i}$ for $i \in 1..n$, $Y \equiv \phi_{\hat{f}}$, and $Z \equiv \phi^{env}_{\hat{f}}$. □

Lemma 4 *Properties (5, 6) are reestablished whenever the upgrade checking algorithm successfully finishes (SAFE is returned).*

Proof Property (5) could be affected only when the summary of $\hat{f}_{main}$ is recomputed (line 10). However, this happens only when we are checking the root of the tree and, at the same time, the check succeeds (line 12). Therefore, by definition of interpolation, the property (5) is maintained.

If Algorithm 2 successfully finishes, then each function call $\hat{f}$ with an invalidated summary must have been assigned a new summary $\sigma_{\hat{f}}$ (line 12) when some of its ancestors $\hat{h}$ passed the summary validity check (line 10). Otherwise, the invalidation would propagate to the root of the calltree and eventually produce an *UNSAFE* result. Therefore, it suffices to show that the newly generated interpolants satisfy the property (6). For this purpose, we can use the same argument as in the proof of Lemma 3, again relying on Theorem 1. Note that if any already valid summaries are used in the summary validity check, we keep those (see condition on line 11) instead of generating new ones. This is sound as we know that $\sigma_{\hat{g}_i} \implies I_{X_i}$, which is consistent with our claim. Analogically, we also keep the old summary $\sigma_{\hat{h}}$ for the root of the subtree that passed the check and caused generation of the new summaries. This is sound as $I_{\hat{h}} \implies \sigma_{\hat{h}}$ is implied by the summary validity check. □

We now show that the properties (5, 6) are strong enough to show that the whole program is safe.

Theorem 2 *When the program calltree annotated by interpolants satisfies the properties (5, 6), then $error_{\hat{f}_{main}} \wedge \phi^{subtree}_{\hat{f}_{main}} \implies false$ (i.e., the whole program is safe).*

Proof Property (5) yields $error_{\hat{f}_{main}} \wedge \sigma_{\hat{f}_{main}} \implies false$. Repeated application of the property (6) to substitute all interpolants on the right hand side yields the claim $error_{\hat{f}_{main}} \wedge \phi^{subtree}_{\hat{f}_{main}} \implies false$. □

We proved correctness of the upgrade checking algorithm in the context of bounded model checking and interpolation-based function summaries. The upgrade checking algorithm, however, is not bound to this context and can be employed also in other verification approaches based on over-approximative function summaries (including the use of other interpolation algorithms). The key ingredient of the correctness proof, property (6), has to be ensured for the particular application.

4 eVolCheck tool architecture

We developed a bounded model checker eVolCheck, which focuses on incremental verification of software written in C. It handles an external summary database to store its outputs, which will be given as inputs for verification of each consequent version. eVolCheck communicates with FunFrog BMC [37] for bootstrapping (to create function summaries of the original code) and exploits its interface with the PeRIPLO solver [34] to solve a PBMC formula, encoded propositionally, and to generate effective interpolants using different algorithms. Altogether, the tool implements two major tasks: syntactic difference check, and the actual upgrade check. eVolCheck binaries, benchmarks used for evaluation, a tutorial explaining how to use eVolCheck and explanation of the most important parameters are available online for other researchers.[2]

This section focuses on the actual implementation of the eVolCheck tool, including an Eclipse plug-in, which facilitates its use, together with details of its industrial and academic applications.

[2] http://verify.inf.usi.ch/evolcheck.

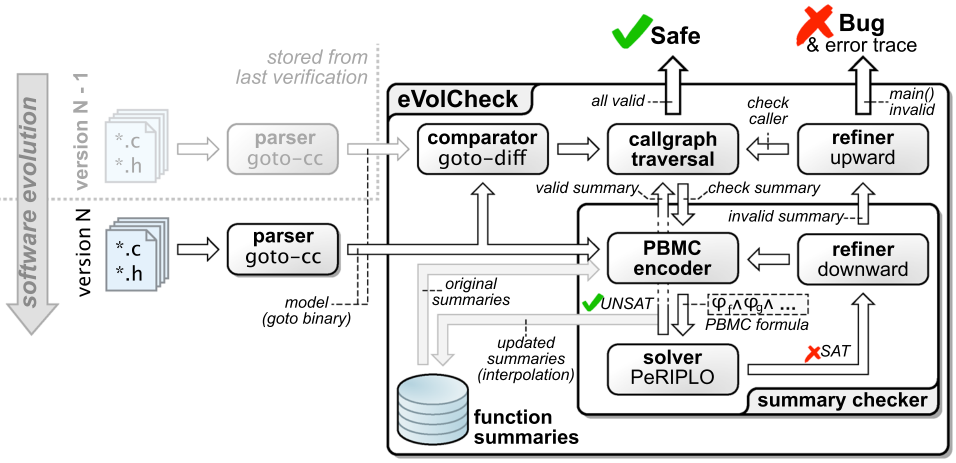


Fig. 7 EVOLCHECK architecture overview

4.1 Tool architecture

This section presents the architecture of the EVOLCHECK tool as depicted in Fig. 7. The tool uses the GOTO- CC compiler provided by the CPROVER framework.[3] The GOTO- CC compiler produces an input model of the source code of C program (called *goto-binary*) suitable for automated analysis. Each version of the analyzed software is compiled using GOTO- CC separately.

eVolCheck. The EVOLCHECK tool itself consists of a comparator, a calltree traversal, an upward refiner and a summary checker. The comparator identifies the changed functions calls. Note that if a function call was newly introduced or removed (i.e., the structure of the calltree is changed), it is considered as change in the parent function call. The calltree traversal attempts to check summaries of all the modified function calls bottom–up. The upward refiner identifies the parent function call to be rechecked when a summary check fails. The summary checker performs the actual check of a function call against its summary. In turn, it consists of a PBMC encoder that takes care of unwinding loops and recursion, SSA-generation and bit-blasting, a solver wrapper that takes care of communication with the solver and the interpolator (PERIPLO), and a downward refiner that identifies nested functions to be refined when a summary check fails possibly due to too coarse summaries. Additionally, there are two optional optimizations in EVOLCHECK, namely slicing and summary optimization. The first can reduce the size of the SSA form using slicing with respect to variables irrelevant to the assertions being checked. The second can compare the existent summaries for the same function and the same bound, and keep the more precise one.

Goto-diff. For comparing the two models, of the previous and the newly upgraded versions, we implemented a tool called GOTO- DIFF. The tool accepts two goto-binary models and analyzes them function by function. the longest common sub-sequence algorithm is used to match the preserved instructions and to identify the changed ones.

It is crucial that GOTO- DIFF works on the level of the models rather than on the level of the source files. This way, it is able to distinguish some of the inconsequential changes in the code. Examples include changes in the order of function declarations and definitions, text changes in comments and white spaces, and simpler cases of refactoring. These changes are usually reported as semantic changes by the purely syntactic comparators (e.g., the standard diff tool). Moreover, as GOTO- DIFF works on the goto-binary models (i.e., after the C pre-processors) it correctly interprets also changes in the pre-processor macros.

Solver and interpolation engine. As mentioned in Sect. 3, to guarantee correctness of the upgrade check, EVOLCHECK requires a solver that is able to generate multiple interpolants with the tree-interpolation property from a single satisfiability query. For this reason, we use the interpolating solver, PERIPLO, which performs proof-logging and provides API for convenient specification of different partitionings of the given formula corresponding to the functions in the calltree. Currently, for using PERIPLO, the PBMC formulas are bit-blasted to the propositional level. As a result, EVOLCHECK provides bit-precise reasoning.

EVOLCHECK has been designed to work with any interpolating solvers (We used in some of our experiments OPENSMT [6], for example). The advantage of using PERIPLO, which we experimented with, is its ability to adjust interpolation by (1) proof manipulation and compression [3,5,11,20,35], (2) variation of the strength of the interpolants, by choosing the labeled interpolation systems [15]. The architecture PERIPLO is depicted on Fig. 8.

Eclipse plug-in. In order to make the tool as user-friendly as possible, we integrated EVOLCHECK in the ECLIPSE development environment in the form of a plug-in. For a user, developing a program using the ECLIPSE environment, the

[3] http://www.cprover.org.

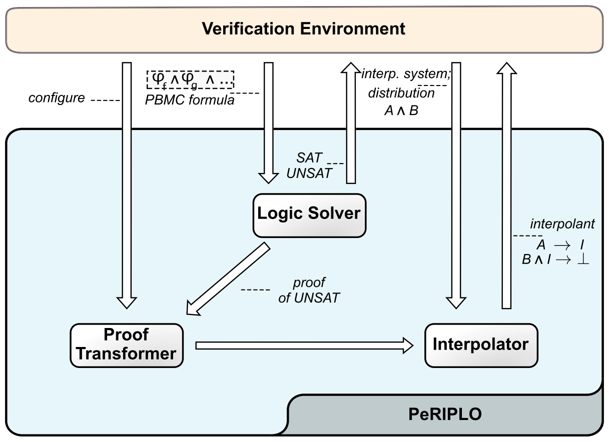


Fig. 8 PeRIPLO architecture overview

EVolCheck plug-in makes it possible to verify changes as part of the development flow for each version of the code. If the version history of the program is empty, the bootstrapping (initial verification) is performed first. Otherwise, EVolCheck verifies the program with respect to the last safe version. Graphical capabilities of Eclipse contain a variety of helpers, allowing configuration of the verification environment.

The plug-in is developed using plug-in development environment (PDE), a tool-set to create, develop, test, debug, build and deploy Eclipse plug-ins. It is built as an external jar-file, which is loaded together with Eclipse. The plug-in follows the paradigm of Debugging components, and provides the separate perspective, containing a view of the source code, highlighted lines, reported by goto- diff, visualization of the error traces and change impact, computed for each upgrade checking of the program.

At the low level, the plug-in delegates the verification tasks to the corresponding command line tools goto- cc, goto-diff and EVolCheck. It maintains a database and external file storage to keep goto-binaries, summaries and other meta-data of each version of each program verified earlier.

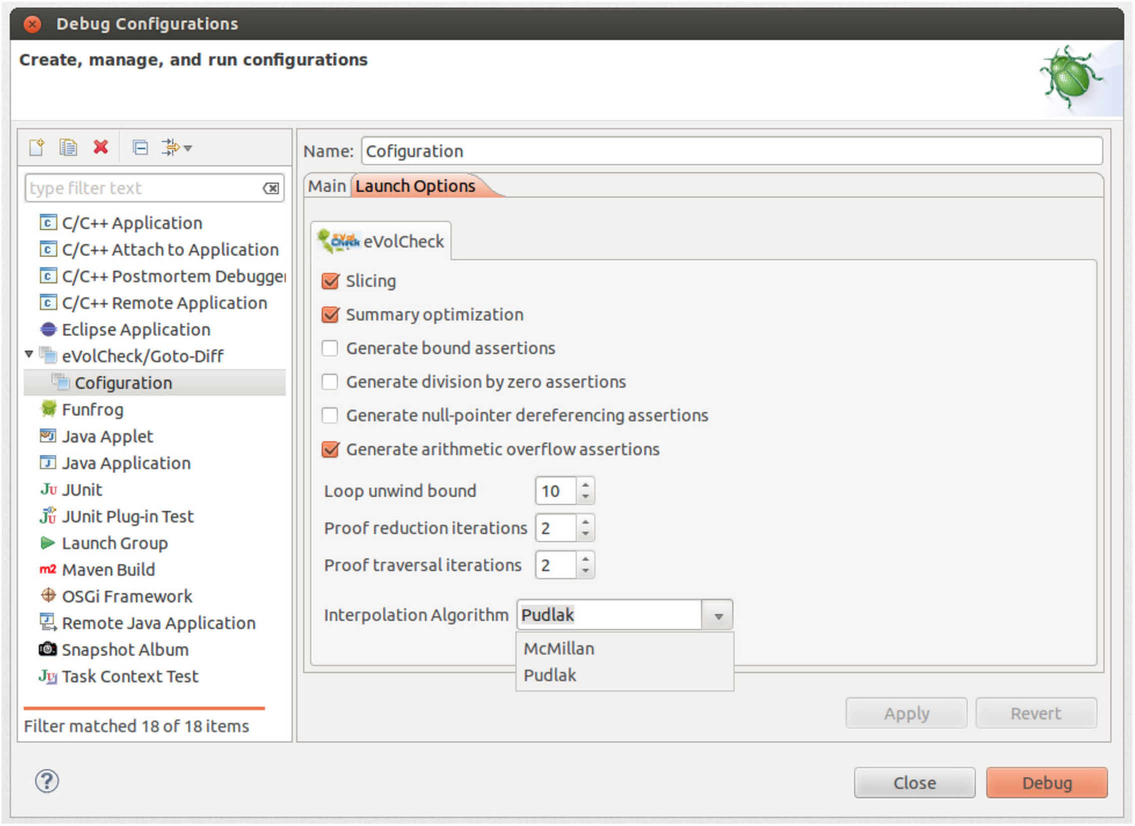
1. The user develops a current version of the program. In order to provide verification condition, the assertions should be placed in the code or generated automatically by the tool. The examples of such assertions are division by zero, null-pointers dereferencing, array out-of-bounds checks.
2. The user opens the *debug configurations* window and chooses the file(s) to be checked and specifies the verification parameters (Fig. 9): unwinding bound, automatic generation of assertions and interpolation strategy. Our plugin then automatically creates the model (goto-binary) from the selected source files and keeps working with it.
3. The plug-in searches for the last safe version of the current program (goto-binary created from the same selection of source files and the same unwinding number). If no such a version is found, it performs the initial bootstrapping check. Otherwise, plug-in restores the summaries and outdated goto-binary from the subsidiary storage. EVolCheck then identifies the modified code by comparing calltrees for both the current and the previous versions. The modified lines of code are marked (Fig. 10) for the user review. Note that modified code may also contain some new assertions, manually or automatically inserted. These assertions will be also considered in the next step.
4. Then the localized upgrade check is performed. If it is unsuccessful, the plug-in reports violation to the user and provides an error trace (Fig. 11). The user can traverse the error trace line by line in the original code and see the valuation of all variables in all states along the error trace. If desired, the user fixes the reported errors and continues from Step 3.
5. In case of successful verification, the positive result is reported (Fig. 12). The plug-in stores the set of valid and new summaries and the goto-binary in the subsidiary storage. In addition, graphical visualization of the change impact in the form of a coloured calltree is available (Fig. 13).

Default parameters. By default, the tool runs without slicing, without summary optimization, does not automatically generate assertions and uses the Pudlák's algorithm for interpolation without proof reduction.

5 Evaluation

To demonstrate the applicability and advantages of EVolCheck, we provide evaluation details of several test cases. `p2p_`*n* were provided by industrial partners for which the changes were extracted from the project repositories. `diskperf_`*n*, `floppy_`*n*, `kbfiltr_`*n* were derived from Windows device driver library. The changes (with different level of impact, from adding an irrelevant line of code to moving a part of functionality between functions) were introduced manually there. The rest of the benchmarks are crafted programs with arithmetic computations. Pre-processing the code with the goto- cc tool generated a collection of goto-binaries that were then processed with EVolCheck focusing the validation to particular functional sub-projects.

Table 2 represents results of the experiments. Each benchmark is shown in a separate row, which summarizes statistics about the initial verification and verification of an upgrade.



Name: Cofiguration

- [✓] Slicing
- [✓] Summary optimization
- [ ] Generate bound assertions
- [ ] Generate division by zero assertions
- [ ] Generate null-pointer dereferencing assertions
- [✓] Generate arithmetic overflow assertions

Loop unwind bound: 10
Proof reduction iterations: 2
Proof traversal iterations: 2
Interpolation Algorithm: Pudlak (McMillan, Pudlak)

Fig. 9 EVOLCHECK configuration window

$Inst(\#)$ measures the original source code size as a number of instructions in the goto-binary (this is the more accurate parameter than the usual "lines of code" since it does not contain declarations of the variables, empty lines of code and so on). $Total(\#)$ represents the number of function calls in the program, and $Diff(\#)$ is the number of changed function calls, identified by GOTO- DIFF.

Time (in seconds) for running GOTO- DIFF $[Diff(s)]$ and for generation of the interpolants $[Itp(s)]$ represents the computational overhead of the upgrade checking procedure, and included to the total running time $[Total(s)]$ of EVOLCHECK. Note that interpolation can not be performed for the buggy upgrades (marked as "err"), for which the corresponded PBMC formula is satisfiable; or for the identical upgrades (marked as "id"), for which GOTO- DIFF returned empty set of changed function calls.

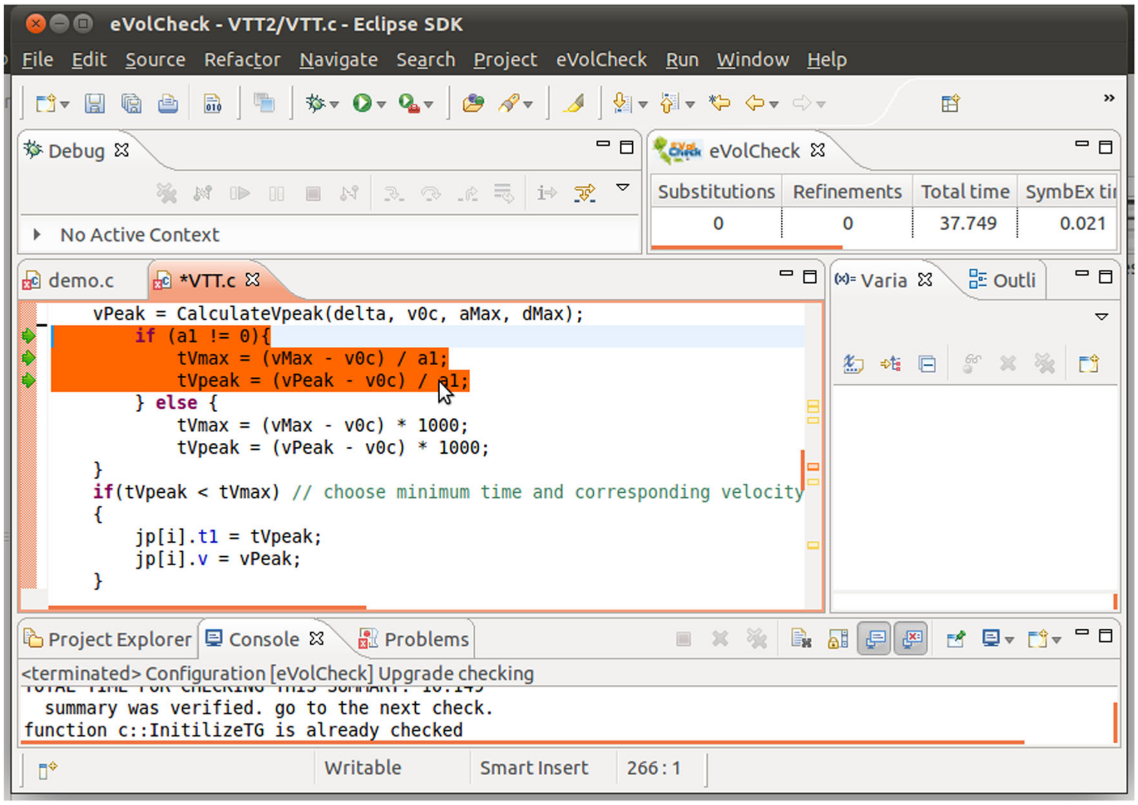
To show advantages of EVOLCHECK, for each change we calculated the speedup (*Speedup*) of the upgrade check versus standalone verification of the changed code from scratch, performed only for the sake of comparison and thus not shown in the table. Finally, the number of invalidated summaries (due to the change) is listed in $Inv(\#)$ column.

Discussion. Our evaluation demonstrates good performance of EVOLCHECK. In particular, the experiments show high effect of upgrade checking for safe upgrades since they result in a small number of refinements (both, upward and downward). Moreover, if the changed function are located deeper in the calltree, this generally leads to a small number of invalidated summaries, as witnessed by the ratio $Inv(\#)/Total(\#)$. For example, consider the case, where summaries of all changed functions were proven valid.

EVOLCHECK is less efficient in case of upgrades, which affect a large amount of function calls located on the different levels of the calltree. When the upgrade introduced a bug, it will cause an increasing amount of upward refinements. However, sometimes even a single buggy change introduced in a higher level of the calltree, might be verified efficiently.

In classical model checking, confirming the absence of bugs is usually more expensive (since it requires the full state-space search) than detecting the bugs (where the search can be terminated once a counter-example is detected). On the contrary, EVOLCHECK is less time- and resource-demanding in proving the absence of bugs. Thus, it might make sense to run EVOLCHECK and a classical model checker (e.g., FUNFROG [17,19,37]) in parallel, and terminate both processes whenever one of them returned a result.

The use of GOTO- DIFF has been particularly useful since it managed to detect test cases with small syntactic changes that

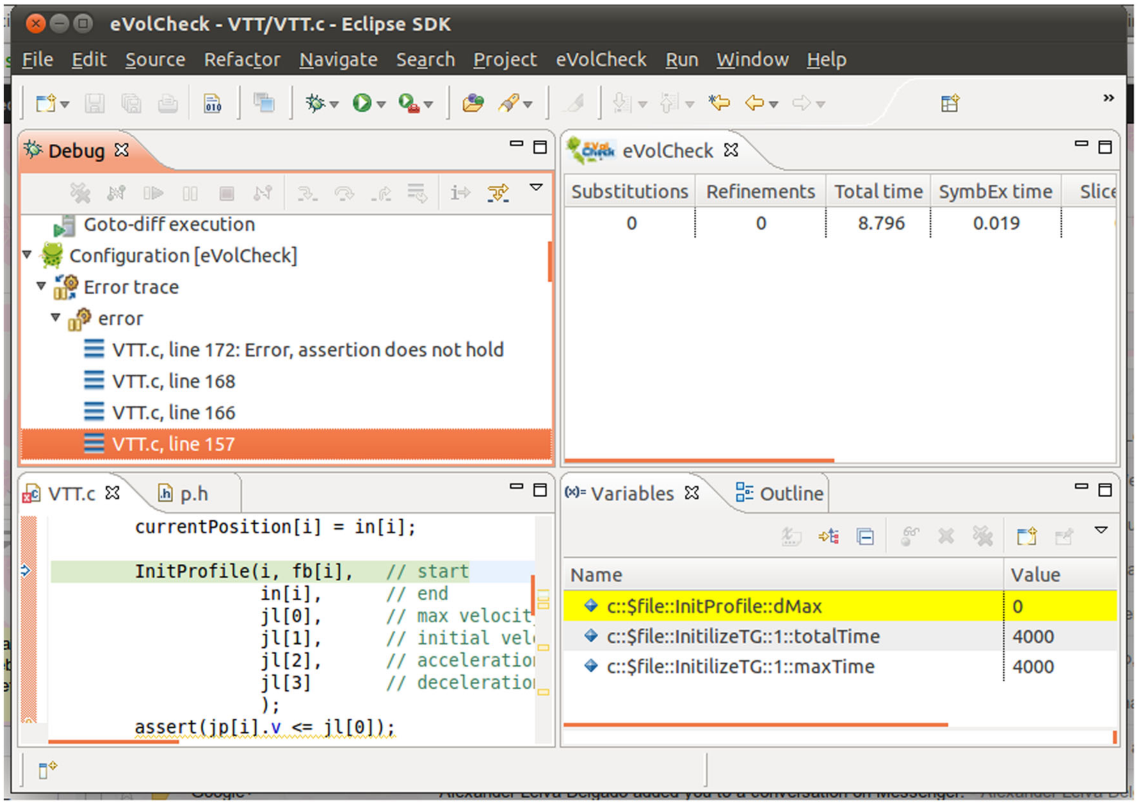

eVolCheck - VTT2/VTT.c - Eclipse SDK
File Edit Source Refactor Navigate Search Project eVolCheck Run Window Help
Debug
No Active Context
eVolCheck
demo.c
*VTT.c
Varia
Outli
Project Explorer
Console
Problems
Writable
Smart Insert
266 : 1

Substitutions	Refinements	Total time	SymbEx ti
0	0	37.749	0.021

```
    vPeak = CalculateVpeak(delta, v0c, aMax, dMax);
        if (a1 != 0){
            tVmax = (vMax - v0c) / a1;
            tVpeak = (vPeak - v0c) / a1;
        } else {
            tVmax = (vMax - v0c) * 1000;
            tVpeak = (vPeak - v0c) * 1000;
    }
    if(tVpeak < tVmax) // choose minimum time and corresponding velocity
    {
        jp[i].t1 = tVpeak;
        jp[i].v = vPeak;
    }
```

```
<terminated> Configuration [eVolCheck] Upgrade checking
  summary was verified. go to the next check.
function c::InitilizeTG is already checked
```

Fig. 10 EVOLCHECK invokes GOTO- DIFF (changed lines are *highlighted*)


eVolCheck - VTT/VTT.c - Eclipse SDK
File Edit Source Refactor Navigate Search Project eVolCheck Run Window Help
Debug
eVolCheck
VTT.c
p.h
Variables
Outline

```
Goto-diff execution
▼ Configuration [eVolCheck]
  ▼ Error trace
    ▼ error
        VTT.c, line 172: Error, assertion does not hold
        VTT.c, line 168
        VTT.c, line 166
        VTT.c, line 157
```

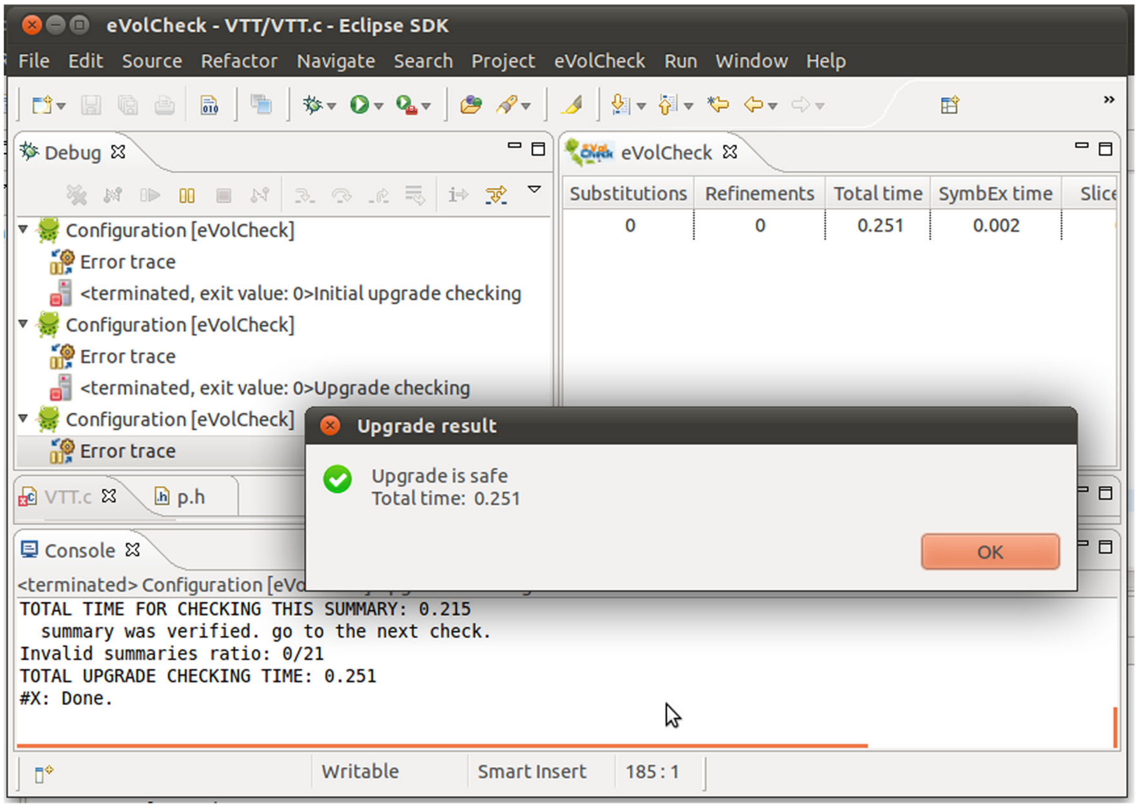
Substitutions	Refinements	Total time	SymbEx time	Slice
0	0	8.796	0.019	

```
        currentPosition[i] = in[i];

        InitProfile(i, fb[i],   // start
                    in[i],      // end
                    jl[0],      // max velocit
                    jl[1],      // initial vel
                    jl[2],      // acceleratio
                    jl[3]       // deceleratio
                    );
        assert(jp[i].v <= jl[0]);
```

Name	Value
c::$file::InitProfile::dMax	0
c::$file::InitilizeTG::1::totalTime	4000
c::$file::InitilizeTG::1::maxTime	4000

Fig. 11 EVOLCHECK error trace


eVolCheck - VTT/VTT.c - Eclipse SDK
File Edit Source Refactor Navigate Search Project eVolCheck Run Window Help
Debug
eVolCheck
VTT.c
p.h
Console
Upgrade result
OK
Writable
Smart Insert
185 : 1

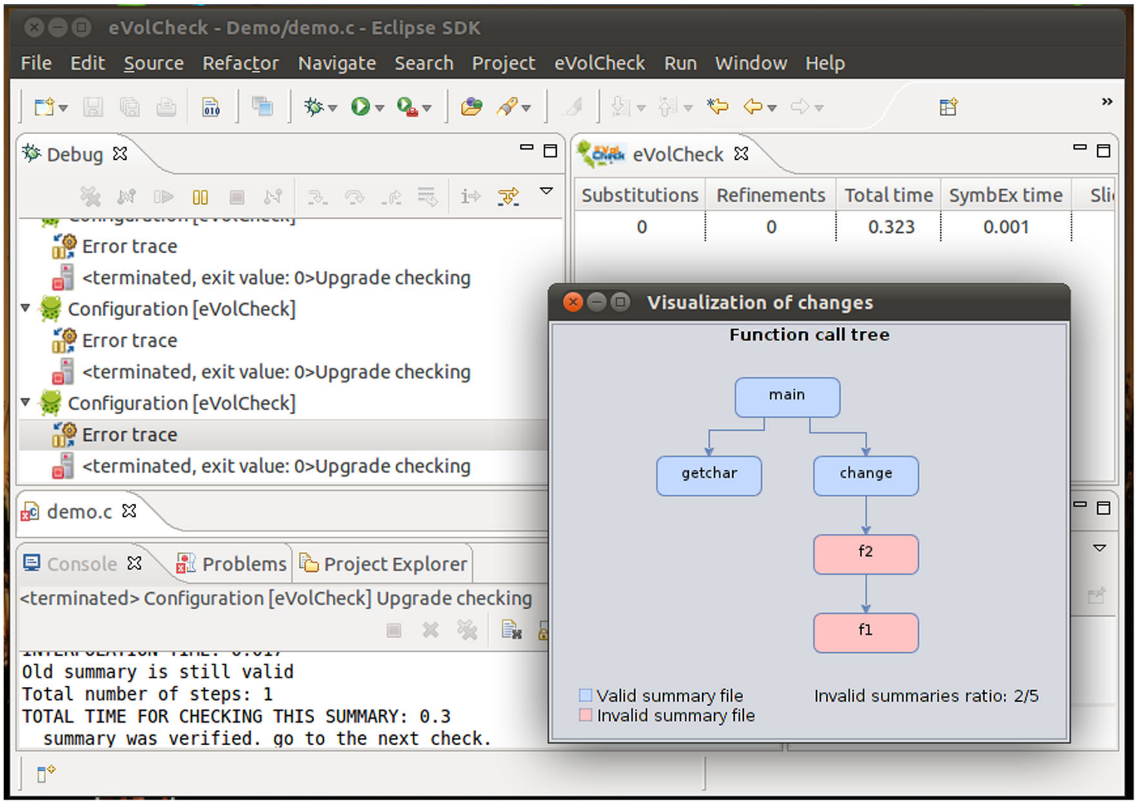
Substitutions	Refinements	Total time	SymbEx time	Slice
0	0	0.251	0.002	

Configuration [eVolCheck]
Error trace
<terminated, exit value: 0>Initial upgrade checking
Configuration [eVolCheck]
Error trace
<terminated, exit value: 0>Upgrade checking
Configuration [eVolCheck]
Error trace

Upgrade is safe
Total time: 0.251

```
<terminated> Configuration [eVo
TOTAL TIME FOR CHECKING THIS SUMMARY: 0.215
  summary was verified. go to the next check.
Invalid summaries ratio: 0/21
TOTAL UPGRADE CHECKING TIME: 0.251
#X: Done.
```

Fig. 12 EVOLCHECK successful verification report



Substitutions	Refinements	Total time	SymbEx time	Sli
0	0	0.323	0.001	

Error trace
<terminated, exit value: 0>Upgrade checking
Configuration [eVolCheck]
Error trace
<terminated, exit value: 0>Upgrade checking
Configuration [eVolCheck]
Error trace
<terminated, exit value: 0>Upgrade checking

<terminated> Configuration [eVolCheck] Upgrade checking

```
Old summary is still valid
Total number of steps: 1
TOTAL TIME FOR CHECKING THIS SUMMARY: 0.3
  summary was verified. go to the next check.
```

🟦 Valid summary file
🟥 Invalid summary file
Invalid summaries ratio: 2/5

Fig. 13 EVOLCHECK change impact

Table 2 eVolCheck experimental evaluation

Benchmark		Bootstrapping		Upgrade check						
Name	Inst (#)	Itp (s)	Total (s)	Diff (#)	Diff (s)	Itp (s)	Total (s)	Inv (#)	Total (#)	Speedup (×)
`floppy 1`	2434	1.455	556.231	21	1.188	0	1.304	0	187	223.21
`p2p 1`	276	0.633	76.884	0	0.018	id	0.018	0	8	4271.33
`p2p 3`	358	0.498	40.618	1	0.02	0.277	10.453	0	20	3.88
`arith 36`	60	20.047	40.53	2	0.001	5.997	7.663	2	5	5.29
`arith 31`	51	12.134	33.043	1	0.001	1.119	1.509	1	4	21.88
`kbfiltr 1`	1024	0.369	31.828	1	0.072	0.004	0.113	0	56	172.04
`kbfiltr 1`	1024	0.371	31.813	2	0.071	0.004	0.24	1	56	102.29
`life 1`	118	3.137	30.9	2	0.004	err	18.757	4	30	1.65
`arith 2`	64	8.123	26.121	2	0.002	0.52	0.927	2	5	28.12
`arith 6`	78	9.914	22.287	3	0.001	2.791	4.227	3	6	5.27
`arith 20`	70	9.83	22.125	3	0.002	3.478	4.607	3	6	4.80
`arith 24`	61	8.844	21.234	2	0.001	17.898	33.008	3	5	0.64
`arith 19`	61	5.561	21.159	1	0.002	0.434	0.571	2	5	36.93
`euler 1`	85	0.742	19.439	1	0.001	0.147	0.678	1	11	28.63
`diskperf 1`	538	0.449	19.301	1	0.027	0.014	0.183	0	19	91.91
`diskperf 2`	535	0.447	19.134	1	0.026	0.259	11.326	2	19	1.69
`diskperf 3`	523	0.4	19.017	2	0.025	0.285	11.298	2	19	1.68
`arith 7`	100	1.518	12.784	3	0.001	2.847	14.881	7	9	0.86
`p2p 2`	355	0.493	6.595	0	0.02	id	0.02	0	9	329.75
`floppy 3`	323	0.161	3.677	1	0.029	0.003	0.07	0	18	37.14
`floppy 2`	320	0.16	3.675	1	0.028	0.003	0.07	0	18	37.50
`diskperf 1`	1880	0.124	3.149	1	0.114	0.001	0.151	1	5	11.88
`floppy 4`	322	0.088	2.127	2	0.028	0	0.101	0	7	16.49
`floppy 5`	330	0.089	1.895	5	0.028	0.082	2.041	0	7	0.92

did not require running the main eVolCheck procedures. For example, in `p2p 1/2`, the comparator proved that the models are identical (however, the source code might still be different), so no further checking was needed.

As expected, in the majority of the experiments, the localized upgrade check provided by eVolCheck outperforms the verification from scratch, which is indicated by *speedup* > 1. Moreover, in many instances (usually on large industrial cases) the speedup is large, which demonstrates good efficiency and usefulness of the tool.

In the future, we plan to conduct a case study where the incremental upgrade checking is applied to a realistic line of revisions (e.g., several dozens of successive program versions from a repository) for a given project. Unfortunately, so far we were not provided with such long sequences of project files.

5.1 Effect of proof compression

Small interpolants have a strong impact on the performance in upgrade checking. Table 3 provides statistics for the two interpolation systems of Pudlák (P) [33] and McMillan (M) [29]. *#Invalid summaries* denotes the total number of summaries that were proven invalid with respect to program updates. $Avg|I|$ and $Time(s)$ indicate the average size of interpolants by means of number of logical connectives and the average verification time over all the benchmarks. Finally, $C_{\text{Time}}/V_{\text{Time}}$ *ratio* estimates the average portion of compression time to verification time.

Table 3 eVolCheck experimental evaluation with PeRIPLO proof compression

No compression	M	P
#Invalid summaries	49	49
Avg $\|I\|$	174959.2	173385.67
Time (s)	1699.73	1716.81
Compression	**M**	**P**
#Invalid summaries	44	44
Avg $\|I\|$	9466.22	10420.37
Time (s)	1014.82	991.25
$C_{\text{Time}}/V_{\text{Time}}$ ratio	0.18	0.16

As proven in [15], interpolants generated by M are logically stronger than the ones generated by P. But in our experimentation, the interpolant strength does not affect verification process significantly. On the other hand, the results show that the size of interpolants seems to have definitely an overall greater impact than interpolant strength. Verification time is principally determined by the size of the summaries. The interesting research direction, which is left as future work, would investigate how the proof compression affects the strength of interpolation (i.e., why compression reduces the amount of invalidated summaries, as can be seen from the table).

6 Related work

The area of software upgrade checking is not as studied as model checking of standalone programs. The idea of an upgrade check that reuses information learned during analysis of the previous program version was employed in [7]. The authors consider the problem of substitutability of updated components of a system. Their algorithm is based on inclusion of behaviors and uses a CEGAR loop [9] combining over- and under-approximations of the component behaviors. First, a containment check is performed, i.e., it is checked that every behavior of the old component occurs also in the new one. Second, they use a learning-based assume-guarantee reasoning algorithm to check compatibility, i.e., that the new component satisfies a given property when the old component does. When compared, our approach focuses on real low-level properties of code expressed as assertions rather than abstract inclusion of behaviors. The use of interpolants also appears to be a more practical approach as compared to the application of learning regular languages techniques employed in [7].

The authors of [21] study effects of code changes on function summaries used in dynamic test generation (also referred to as white-box fuzzing). White-box fuzzing consists of running a program while simultaneously executing the program symbolically in order to gather constraints on inputs from conditional statements encountered along the execution. The particular goal of [21] is to identify summaries that were affected by the change and cannot be used to analyze the new version. Then the actual testing is performed using the preserved summaries. Due to reliance on testing, this approach suffers of *path explosion* problem, i.e., infeasibility to cover all program paths. In contrast, we obtain and manipulate function summaries only symbolically, thus allowing to encode all paths into a single formula.

Another group of related work aims at equivalence checking and regression verification of programs [10,16,22,26,32,40]. Strichman et al [22] employ bounded model checking to prove partial equivalence of programs. As in our algorithm, their method starts with the syntactic difference check that identifies the set of modified functions. Then it also traverses the calltree bottom–up, and separately checks equivalence between the old and the new versions of the function, while all the nested calls are abstracted using the same uninterpreted function. Differential Symbolic Execution [32] and further established Change Impact Analysis [2] attempts to show equivalence of two versions of code using symbolic execution or to compute a behavioral delta when not equivalent. Clarke et al. [10] checks equivalence of a Verilog circuit and a C program through encoding and solving quantifier-free SAT formula.

Since finding a solution to the more general problem of checking absolute equivalence is hard, there is a group of techniques [16,40] that check partial (or relative) equivalence with respect to some safety property (or a set of properties). The approach of [28], implemented in the tool SymDiff [26], decides conditional partial equivalence, i.e., equivalence under certain input constraints. Moreover, SymDiff allows extraction of these constraints and reports them to the user. The technique from the article also belongs to this group, but it is the only one that uses BMC and Craig interpolation.

Last group of related work includes approaches using interpolation-based function summaries (such as [1,30]). Although these do not consider upgrade checking, we believe that our incremental algorithm may be instantiated in their context similar to how we instantiated it in the context of [39].

7 Conclusion

We presented an upgrade checking algorithm using interpolation-based function summaries. Instead of running bounded model checking on the entire new version of a program, the modified functions are first compared against their over-approximative summaries from the old version. If this local check succeeds, the upgrade is safe. We proved that the proposed algorithm is sound, if the summaries are generated from the same proof using the original Pudlák's algorithm. Experimental evaluation using our prototype implementation supports our intuition about ability to check system upgrades locally and demonstrates that the algorithm significantly speeds up checking programs with incremental changes.

Acknowledgements We thank the following people for their valuable contribution during the work on this paper: Murillo Miranda Cristina Maria for her implementation work on the Eclipse plug-in; Antti Hyvärinen for his numerous suggestions to improve the approach; Leonardo Alt and Simone Fulvio Rollini for developing and evaluating PeRIPLO; Michael Tautschnig for his help with CProver and goto-cc adjustments; the PINCETTE project (http://www.pincette-project.eu) validators for assistance with the industrial test cases; USI students for their feedback on the eVolCheck usage and the anonymous reviewers for their constructive comments, which helped to improve the article.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Whale: an interpolation-based algorithm for inter-procedural verification. In: VMCAI. LNCS, vol. 7148, pp. 39–55. Springer, Berlin (2012)
2. Backes, J.D., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impact summaries. In: SPIN. LNCS, vol. 7976, pp. 99–116. Springer, Berlin (2013)
3. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-time reductions of resolution proofs. In: HVC. LNCS, vol. 5394, pp. 114–128. Springer, Berlin (2008)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. LNCS, vol. 1579, pp. 193–207. Springer, Berlin (1999)
5. Bruttomesso, R., Rollini, S., Sharygina, N., Tsitovich, A.: Flexible interpolation with local proof transformations. In: ICCAD, pp. 770–777 (2010)
6. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: TACAS. LNCS, vol. 6015, pp. 150–153. Springer, Berlin (2010)
7. Chaki, S., Clarke, E., Sharygina, N., Sinha, N.: Dynamic component substitutability analysis. In: FM. LNCS, vol. 3582, pp. 512–528. Springer, Berlin (2005)
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer, Berlin (2004)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. LNCS, vol. 1855, pp. 154–169. Springer, Berlin (2000)
10. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC, pp. 368–371. ACM, New York (2003)
11. Cotton, S.: Two techniques for minimizing resolution proofs. In: SAT. LNCS, vol. 6175, pp. 306–312. Springer, Berlin (2010)
12. Craig, W.: Three uses of the Herbrand–Gentzen theorem in relating model theory and proof theory. J. Symb. Logic 269–285 (1957)
13. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F.: An efficient method of computing static single assignment form. In: POPL, pp. 25–35. ACM, New York (1989)
14. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: TACAS. LNCS, vol. 6015, pp. 280–295. Springer, Berlin (2010)
15. D’Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: VMCAI. LNCS, vol. 5944, pp. 129–145. Springer, Berlin (2010)
16. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Incremental verification of compiler optimizations. In: NFM. LNCS, vol. 8430, pp. 300–306. Springer, Berlin (2014)
17. Fedyukovich, G., D’Iddio, A.C., Hyvärinen, A.E.J., Sharygina, N.: Symbolic detection of assertion dependencies for bounded model checking. In: FASE. LNCS, vol. 9033, pp. 186–201. Springer, Berlin (2015)
18. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: incremental upgrade checker for C. In: TACAS. LNCS, vol. 7795, pp. 292–307. Springer, Berlin (2013)
19. Fedyukovich, G., Sharygina, N.: Towards completeness in bounded model checking through automatic recursion depth detection. In: SBMF. LNCS, vol. 8941, pp. 96–112. Springer, Berlin (2014)
20. Fontaine, P., Merz, S., Paleo, B.W.: Compression of propositional resolution proofs via partial regularization. In: CADE. LNCS, vol. 6803, pp. 237–251. Springer, Berlin (2011)
21. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically validating must summaries for incremental compositional dynamic test generation. In: SAS. LNCS, vol. 6887. Springer, Berlin (2011)
22. Godlin, B., Strichman, O.: Regression verification. In: DAC, pp. 466–471. ACM, New York (2009)
23. Gurfinkel, A., Rollini, S., Sharygina, N.: Interpolation properties and SAT-based model checking. In: ATVA. LNCS, vol. 8172, pp. 255–271. Springer, Berlin (2013)
24. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM, New York (2004)
25. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. Theor. Comput. Sci. **404**(3), 256–274 (2008)
26. Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Conditional equivalence. Tech. Rep. MSR-TR-2010-119, Microsoft Research (2010)
27. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using state and transition invariants. Form. Methods Syst. Des. **42**(3), 221–261 (2013)
28. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: FSE, pp. 345–355. ACM, New York (2013)
29. McMillan, K.L.: Interpolation and SAT-based model checking. In: CAV. LNCS, vol. 2725, pp. 1–13. Springer, Berlin (2003)
30. McMillan, K.L.: Lazy annotation for program testing and verification. In: CAV. LNCS, vol. 6174, pp. 104–118. Springer, Berlin (2010)
31. Merz, F., Falke, S., Sinz, C.: LLBMC: bounded model checking of C and C++ programs using a compiler IR. In: VSTTE. LNCS, vol. 7152, pp. 146–161. Springer, Berlin (2012)
32. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: FSE, pp. 226–237. ACM, New York (2008)
33. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. **62**(3), 981–998 (1997)
34. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: a framework for producing effective interpolants in SAT-based software verification. In: LPAR. LNCS, vol. 8312, pp. 683–693. Springer, Berlin (2013)
35. Rollini, S.F., Bruttomesso, R., Sharygina, N.: An efficient and flexible approach to resolution proof reduction. In: HVC. LNCS, vol. 6504, pp. 182–196. Springer, Berlin (2010)
36. Rollini, S.F., Sery, O., Sharygina, N.: Leveraging interpolant strength in model checking. In: CAV. LNCS, vol. 7358, pp. 193–209. Springer, Berlin (2012)
37. Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: bounded model checking with interpolation-based function summarization. In: ATVA. LNCS, vol. 7561, pp. 203–207. Springer, Berlin (2012)
38. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: FMCAD, pp. 114–121. IEEE (2012)
39. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: HVC. LNCS, vol. 7261, pp. 160–175. Springer, Berlin (2012)
40. Yang, G., Khurshid, S., Person, S., Rungta, N.: Property differencing for incremental checking. In: ICSE, pp. 1059–1070. ACM, New York (2014)