Incremental Upgrade Checking by Means of Interpolation-based Function Summaries

Ondrej Sery*† Grigory Fedyukovich* Natasha Sharygina*
*Formal Verification Lab, University of Lugano, Switzerland
†D3S, Faculty of Mathematics and Physics, Charles University, Czech Rep.

***Abstract*—During its evolution, a typical software/hardware design undergoes a myriad of small changes. However, it is extremely costly to verify each new version from scratch. As a remedy to this problem, we propose to use function summaries to enable incremental verification of the evolving systems. During the evolution, our approach maintains function summaries derived using Craig's interpolation. For each new version, these summaries are used to perform a local incremental check. Benefit of this approach is that the cost of the check depends on the extent of the change between the two versions and can be performed cheaply for incremental changes without resorting to re-verification of the entire system. Our implementation and experimentation in the context of the bounded model checking for C confirms that incremental changes can be verified efficiently for different classes of industrial programs.**

I. Introduction

Software and hardware designs are usually not written all at once, but are built 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 removed; 3) software components are updated or substituted to adapt to architectural and requirement changes; just to name a few. Changes are done frequently during the lifetime of many products and can introduce errors that were not present in the old versions, or expose errors that were present before but did not get exposed. The state of the affairs is that the correctness of the system has to be re-validated from scratch after any (even minor) change. Often the cost of this validation dominates costs of the products.

Currently, re-validation mostly relies on the execution of extensive test suits, which is inherently not exhaustive; fault localization is mainly manual and driven by experts' knowledge of the system; fault fixing often introduces new faults that are hard to detect and remove. To address this problem, this paper presents a new fully automated approach that extends formal verification by model checking to the problem of validation of system upgrades. The new technique focuses on the incremental changes and takes advantage of the effort already invested in the verification of previous versions. The target of our approach is to avoid (when possible) re-validation of the new system and to reduce analysis only to the parts of the system which were affected by the change.

The advantages of model checking are often shaded by its high consumption of computational resources (known as the state-space explosion problem). Many efficient complexity reduction algorithms have been developed to cope with this problem among which the representative approaches are symbolic verification such as Bounded Model Checking (BMC) [1], and different types of automated abstraction (predicate abstraction [2], interpolation-based reasoning [3], function summarization [4], [5], [6], [7], etc.). Most state-of-the-art model checking tools implement some (or combinations) of these methods in order to deal with complex designs. Notably, combinations of such techniques are known to be crucial for combating the high complexity of verification.

This paper presents a solution to the upgrade checking problem that extends the existing efficient techniques known to work well for standalone verification to the problem of analysis of system changes. In particular, it presents an incremental bounded model checking approach that uses function summarizations for local upgrade checks. The upgrade checking algorithm maintains program function summaries (i.e., over-approximations of the actual behavior of the functions, in our case computed by means of Graig interpolation [7]) and when a new version arrives, it checks if the summaries of the modified functions are still valid over-approximations. This is a local and cheap check. If it succeeds, the upgrade is safe with respect to both the preserved and newly added behaviors (we assume the upgrade was cleaned off the previously detected bugs). If not, the check is propagated by the call tree traversal to the caller of the modified function. As soon as the safety is established, new summaries are generated using Craig interpolation for all the functions with invalid summaries. If the check fails for the call tree root (the main function of the program), an error trace is created and reported to the user as a witness to the violation.

The upgrade checking algorithm implements the refinement strategy for dealing with spurious behaviors which 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 [7], [8] and extends it to 1) handle summaries of nested function calls and 2) consequently to use them to further simplify the validity checks of the upgraded functions summaries. Failures of such checks may be due to the use of too weak summaries, in which case, the refinement is used to expand the involved function calls on demand.

We developed a prototype implementation of the proposed algorithm and evaluated it using a set of industrial benchmarks.

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

Our experimentation confirms that the incremental analysis of upgrades containing incremental changes is often orders of magnitude faster then analysis preformed from scratch.

Although we implemented the proposed upgrade checking algorithm in the context of bounded model checking, the algorithm itself is more general and can be employed in other contexts, where over-approximative function summaries are used. For example, the WHALE approach [6] designed for standalone verification could be easily extended to incremental upgrade checking using our algorithm.

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

- It presents a fully automated model-checking-based technique for verification of incremental upgrades. It is able to re-validate all previously established properties and to detect newly introduced errors.
- It efficiently combines bounded model checking with function summarization for *local* and *incremental* analysis of changes. The use of Craig interpolation to compute summaries allows capturing symbolically all execution traces through the function and, together, with the local per-function checks of the new algorithm, results in the efficient analysis procedure.
- It reports on the prototype implementation of the new technique and its validation on industrial benchmarks.

The rest of the paper is organized as follow. Sect. II defines the notation and presents background on function summarization in BMC. Sect. III presents the new upgrade checking algorithm and proves its correctness. Sect. IV describes implementation and evaluation of the approach. Sect. V discusses the related work and Sect. VI concludes the paper.

II. Background and Previous Work

Craig Interpolation [9] is a popular abstraction technique widely used in Model Checking. Given a pair of formulas (A, B), *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 [9]. As shown in [10], an interpolant can be constructed from a proof of unsatisfiability by an algorithm referred as *Pudák's algorithm*. Although other algorithms exist, we will focus on Pudlák's throughout the paper. Interpolants are useful in various verification gambits including refinement of predicate abstraction [4], and bounded model checking [3] to name a few. The following outlines how interpolation is used for function summarization in BMC [7].

BMC is aimed at searching for errors in a program within the given number (bound) of loop iterations and recursion depth. First, it unwinds the program according to the bound. Second, it constructs the Static Single Assignment (SSA) form of the program, supplies it with the negated property to be checked, and encodes it into a logical formula, a *BMC formula*. The formula is satisfiable if and only if an error is reachable in the unwound program. If the formula is satisfiable, a satisfying assignment identifies a trace leading to an error. If unsatisfiable, the program is safe.

Standard BMC constructs a monolithic BMC formula where all function calls are inlined. In order to make interpolation applicable for extraction of function summaries, we construct BMC formula so that each function call is represented by a separate conjunct, and call it a *partitioned BMC* (PBMC) formula. To describe construction of PBMC formula in more details, we use the notion of an unwound program in terms of its call tree.

An *unwound program* for a bound ν is a tuple $P_\nu = (F, f_{main})$, s.t. 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 transitive closure of *child*. $\hat{F}$ 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 $\hat{F}$, s.t. $\forall \hat{f}, \hat{g} \in \hat{F} : child(\hat{f}, \hat{g}) \rightarrow child(f, g)$, and *subtree* is a transitive closure of the extended relation $child$. A *summary* of a function is a relation over its input and output variables, which over-approximates the precise behavior of the given function. This means that a summary contains all possible behaviors of the function (under the given bound ν) and possibly more. We use $\mathbb{S}$ to denote the set of all summaries.

Algorithm 1 summarizes the method for construction of function summaries in BMC. There are two major differences from the standard BMC algorithm that should be pointed out. First, the PBMC formula is constructed as a conjunction of parts representing individual functions. Second, function summaries are extracted using interpolation for every individual part of the PBMC formula.

PBMC formula construction (line 1). The PBMC formula is constructed 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. In addition, $\phi_{\hat{f}}$ contains special propositional symbols to bind the individual partitions together. An example of such a symbol is $error_{\hat{f}}$, which is constrained to be true if and only if the function call $\hat{f}$ results in an error. Consequently, $error_{\hat{f}_{main}}$ encodes reachability of an error in the entire program.

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

Algorithm 1: Function summarization in BMC [7]

Input: Unwound program $P_\nu = (F, f_{main})$ with function calls $\hat{F}$

Output: Verification result: {*SAFE, UNSAFE*}, mapping of function calls to their summaries $summaries$

Data: ϕ: PBMC formula

```
1 φ ← CreateFormula(f̂_main) ∧ error_{f̂_main};
2 result, proof ← Solve(φ) ;   // run SAT-solver
3 if result = SAT then
4     return UNSAFE;
5 foreach f̂ ∈ F̂ do            // extract summaries
6     summaries(f̂) ← Interpolate(proof, f̂);
7 end
8 return SAFE;
```

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 separates the PBMC formula into $A \equiv \phi^{subtree}_{\hat{f}}$ and $B \equiv \phi^{env}_{\hat{f}}$ and generates an interpolant $I_{\hat{f}}$ for the pair (A, B). Such interpolant $I_{\hat{f}}$ is a summary for the function f. The generated interpolants are associated with the function calls by a mapping[1] *summaries*: $\hat{F} \to \mathbb{S}$, i.e., $summaries(\hat{f}) = I_{\hat{f}}$.

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

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

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

If the resulting formula 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, determined by a satisfying assignment and by dependency analysis. Second, the *refined* substitution scenario Ω' is constructed from Ω by mapping the function calls corresponding to the identified summaries to *inline*. Then, the next iteration of the algorithm is run using Ω'. If no summary is identified for refinement, the error is real.

III. Upgrade Checking

This section describes our solution to the upgrade checking problem, the incremental summary-based model checking algorithm. As an input, the algorithm takes two versions of the system, 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 initial version of the system), a bootstrapping verification run is needed to analyze the entire new version of the system and to generate the summaries, which are then maintained during the incremental runs.

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

For example, Figure 1-a depicts an output of the preprocessing, i.e., a call tree 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 call tree, summaries of all functions marked as changed are rechecked. That is, a cheap 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 (i.e., rightmost node in Figure 1-b) 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 call tree (Figure 1-b,c). When the check fails for the root of the call tree (i.e., program entry point $\hat{f}_{main}$), a real error is identified and reported to the user. The following first presents this basic algorithm in more details and then describes its optimization with a refinement loop and proves its correctness. Note that we will describe the upgrade checking algorithm instantiated

[1]Here, we consider only a single summary per a function call for the sake of simplicity. This still means multiple summaries per a single function called multiple times. Our prototype implementation does not have this restriction.

Algorithm 2: Upgrade checking algorithm

Input: Unwound program $P_\nu = (F, f_{main})$ with function calls $\hat{F}$, mapping $summaries : \hat{F} \to \mathbb{S}$, set $changed \subseteq \hat{F}$
Output: Verification result: {*SAFE, UNSAFE*}
Data: $D \subseteq \hat{F}$: function calls to recheck, ϕ: PBMC formula, $invalid \subseteq \mathbb{S}$: set of invalid summaries

1 $D \leftarrow \{\hat{f} \mid \hat{f} \in changed\}$, $invalid \leftarrow \emptyset$;
2 **while** $D \neq \emptyset$ **do**
3 choose $\hat{f} \in D$, s.t. $\forall \hat{h} \in D : \neg subtree(\hat{f}, \hat{h})$;
4 $D \leftarrow D \setminus \{\hat{f}\}$;
5 **if** $\hat{f} \in dom(summaries)$ **then**
6 $\phi \leftarrow$ `CreateFormula`$(\hat{f})$;
7 $result, proof \leftarrow$ `Solve`$(\phi \wedge \neg summaries(\hat{f}))$;
8 **if** $result =$ UNSAT **then**
9 **for** $\hat{g} \in \hat{F} : subtree(\hat{f}, \hat{g}) \wedge (\hat{g} \notin dom(summaries) \vee summaries(\hat{g}) \in invalid)$ **do**
10 $summaries(\hat{g}) \leftarrow$ `Interpolate`$(proof, \hat{g})$;
11 **end**
12 **continue**;
13 **end**
14 $invalid \leftarrow invalid \cup \{summaries(\hat{f})\}$;
15 **end**
16 **if** $\hat{f} = \hat{f}_{main}$ **then**
17 **return** *UNSAFE*; `// real error found`
18 $D \leftarrow D \cup \{parent(\hat{f})\}$; `// check parent`
19 **end**
20 **return** *SAFE*; `// system is safe`

in the context of bounded model checking. However, the algorithm is more general and can be applied in other approaches relying on over-approximative function summaries.

A. Basic Algorithm

We proceed by presenting the basic upgrade checking algorithm (Alg. 2). As an input, Alg. 2 takes the unwound new version of the program, a mapping *summaries* from the function calls in the new version to the summaries from the old version, and a set *changed* marking the function calls corresponding to the functions that were changed or added in the new version (as an output of the preprocessing).

The algorithm keeps a set D of function calls that require rechecking. Initially, this set contains all the function calls marked by *changed* (line 1). Then the algorithm repeatedly removes a function call $\hat{f}$ from D 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 D (line 3). 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 6, 7. First, the PBMC formula encoding the subtree of $\hat{f}$ is constructed and stored as ϕ. Then, conjunction of ϕ with a negated summary of $\hat{f}$ is passed to a 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 (line 9-11). If satisfiable, there is a combination of inputs and outputs of the function f that is not covered by its original summary, thus the summary is not valid for the new version (line 14). In this case, either a real error is identified (lines 16, 17) or the check is propagated to the function caller (line 18).

Note that if the chosen function call $\hat{f}$ has no summary, e.g., due to being a newly added function, the check is propagated to the caller immediately (condition at line 5) and the summary of $\hat{f}$ is created later when the check succeeds for an ancestor function call.

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 20) when all function calls requiring rechecking were analyzed (line 2). Either all the summaries possibly affected by the program change are immediately shown to be still valid over-approximations (see Figure 2-a) or some are invalid but the propagation stops at a certain level of the call tree and new valid summaries are generated (see Figure 2-b). The algorithm terminates with UNSAFE result (lines 18), when the check propagates to the call tree root, $\hat{f}_{main}$, and fails (see Figure 2-c). In this case, a real error is encountered and reported to the user.

B. 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 6). 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 SAT, 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 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, check fails and the summary is shown invalid. This refinement loop (replacing lines 6, 7 in Alg. 2) iterates until validity of the summary is decided.

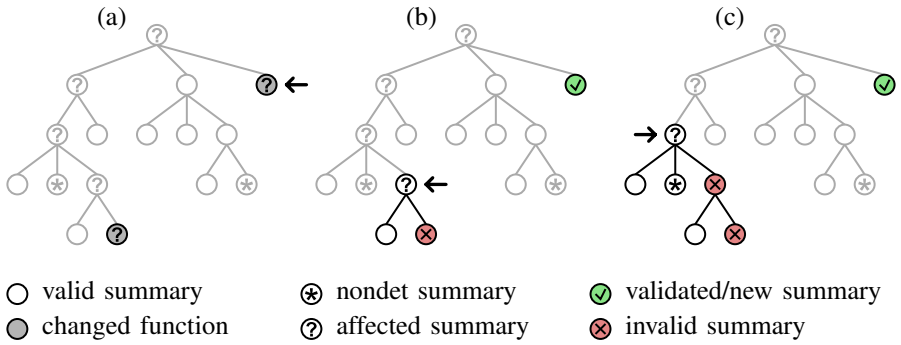

Figure 1: Progress of the upgrade checking algorithm; the faded parts of the call tree were not yet analyzed by the algorithm

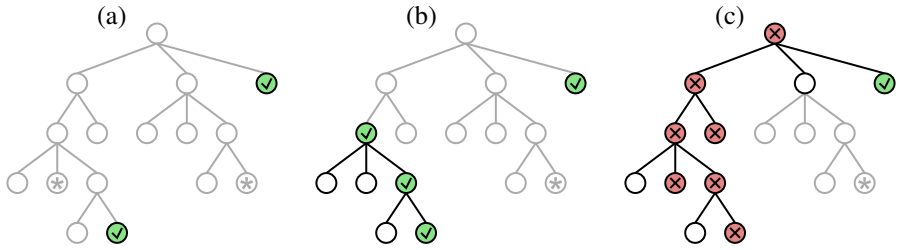

Figure 2: Sample outcomes of Alg. 2; analyzing the faded parts of call tree is not required to decide safety of the upgrade

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 call tree. This may result in a number of solver calls quadratic in the size of the call tree, where the last call is as complex as the verification of the entire program from scratch. This paper focuses on incremental changes and thus for most cases there is no need for the complete call graph 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.

C. Correctness

This section demonstrates the correctness of the upgrade checking algorithm, i.e., given an unwinding bound ν, the algorithm always terminates with the correct answer w.r.t. ν. Note that throughout this section, program safety is understood considering the bound ν[2]. Also, we use $\sigma_{\hat{f}}$ as a shortcut for *summaries*$(\hat{f})$. The key insight for the correctness is that after each successful run of Alg. 2 (i.e., when *SAFE* is returned), the following two properties are maintained.

$$error_{\hat{f}_{main}} \wedge \sigma_{\hat{f}_{main}} \rightarrow \bot \tag{1}$$

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

$$\sigma_{\hat{g}_1} \wedge \ldots \wedge \sigma_{\hat{g}_n} \wedge \phi_{\hat{f}} \rightarrow \sigma_{\hat{f}} \tag{2}$$

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

The following theorem is required to prove the correctness of Alg. 2. It considers properties of interpolants (also known as *tree interpolants*) generated from the same resolution proof using Pudlák's algorithm [10].

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

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

Lemma 1. *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 $(I_X \wedge I_Y) \rightarrow I_{XY}$.*

Proof: By structural induction over the resolution proof, we show that $(I_X \wedge I_Y) \rightarrow I_{XY}$ 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 $(false \wedge true) \rightarrow false$, which holds. The case $C \in Y$ is symmetric. When $C \in Z$, we have $(true \wedge true) \rightarrow true$, 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 $(I^1_X \wedge I^1_Y) \rightarrow I^1_{XY}$ and partial interpolants I^2_X, I^2_Y, and I^2_{XY} for the node $\neg x \vee C_2$ so that $(I^2_X \wedge I^2_Y) \rightarrow I^2_{XY}$. We need to consider the different cases of coloring of x based on its occurrence in different

Table I: 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

subsets of the parts of the formula $X \wedge Y \wedge Z$. The cases are summarized in Table I. In case $x \in X$, we have:

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

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

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

$$I_X \equiv I_X^1 \wedge I_X^2, \quad I_Y \equiv I_Y^1 \wedge I_Y^2$$
$$I_{XY} \equiv I_{XY}^1 \wedge I_{XY}^2$$

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

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:

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

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

When we apply the result of Lemma 1 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 1, we get $(I_{X_1} \wedge \ldots \wedge I_{X_n} \wedge I_Y) \rightarrow I_{XY}$, where I_Y is Craig interpolant for the pair $(Y, X_1 \wedge \ldots \wedge X_n \wedge Z)$ derived using Pudlák's algorithm over the resolution proof $\mathbb{P}$. Using $Y \rightarrow I_Y$, we obtain the claim $(I_{X_1} \wedge \ldots \wedge I_{X_n} \wedge Y) \rightarrow I_{XY}$. ∎

In the following two lemmas, we first show that the properties (1, 2) hold after an initial whole program check. Then we show that the properties are maintained between the individual successful upgrade checks.

Lemma 2. *After an initial whole-program check, the properties (1, 2) hold over the call tree annotated by the generated interpolants.*

Proof: Recall that the summaries are constructed only when the program is safe. In other words, $error_{\hat{f}_{main}} \wedge \phi^{subtree}_{\hat{f}_{main}} \rightarrow \bot$. Thus, by definition of interpolation, $error_{\hat{f}_{main}} \wedge I_{\hat{f}_{main}}$ is obviously unsatisfiable, i.e., the property (1) holds. The property (2) 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 3. *The properties (1, 2) are reestablished whenever the upgrade checking algorithm successfully finishes (SAFE is returned).*

Proof: The property (1) could be affected only when the summary of $\hat{f}_{main}$ is recomputed (line 8). However, this happens only when we are checking the root of the tree and, at the same time, the check succeeds (line 10). Therefore, by definition of interpolation, the property (1) is maintained.

If Alg. 2 successfully finishes, then each function call $\hat{f}$ with an invalidated summary must have been assigned a new summary $\sigma_{\hat{f}}$ (line 10) when some of its ancestors $\hat{h}$ passed the summary validity check (line 8). Otherwise, the invalidation would propagate to the root of the call tree and eventually produce an *UNSAFE* result. Therefore, it suffices to show that the newly generated interpolants satisfy the property (2). For this purpose, we can use the same argument as in the proof of Lemma 2, 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 9) instead of generating new ones. This is sound as we know that $\sigma_{\hat{g}_i} \rightarrow I_{X_i}$, 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 $I_{\hat{h}} \rightarrow \sigma_{\hat{h}}$ is implied by the summary validity check. ∎

We now show that the properties (1, 2) are strong enough to show that the entire program is safe.

Theorem 2. *When the program call tree annotated by interpolants satisfies the properties (1, 2), then* $\text{error}_{\hat{f}_{main}} \wedge \phi^{subtree}_{\hat{f}_{main}} \rightarrow \bot$ *(i.e., the entire program is safe).*

Proof: The property (1) yields $error_{\hat{f}_{main}} \wedge \sigma_{\hat{f}_{main}} \rightarrow \bot$. Repeated application of the property (2) to substitute all interpolants on the right hand side yields the claim $error_{\hat{f}_{main}} \wedge \phi^{subtree}_{\hat{f}_{main}} \rightarrow \bot$. ∎

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, the property (2), has to be ensured for the particular application.

IV. Evaluation

We implemented a prototype, eVolCheck, of the upgrade checking algorithm for incremental verification. It performs the checks of upgrades using outputs of the previous check and provides its own outputs to the next one. The required input is function summaries of the previous version. eVolCheck communicates with FunFrog [7] for bootstrapping (to create

function summaries of the original code) and exploits its interface with the OpenSMT solver [11] to solve a PBMC formula, encoded propositionally, and to generate interpolants. Altogether, the tool implements two major tasks: syntactic difference check, and the actual upgrade check.

For the first task, we implemented a syntactic difference tool called `goto-diff`. First, `goto-diff` extracts intermediate representations of a pair of (old and new) programs expressed in simple statements (assignments, guards, gotos, function calls) and constructs a so called `goto-binary`. For this, we use the `goto-cc`[3] verification front-end. Since, `goto-binary` is a semantically clean representation of the source code, some syntactically different programs may result in an equivalent representation, i.e., some refactoring changes may be shown safe already at this stage without running the upgrade checking algorithm. Second, `goto-diff` compares the call trees of the programs. For each pair of matching functions, `goto-diff` analyzes their bodies.[4] Unreachable functions of the programs are not processed. Finally, `goto-diff` outputs the new call tree, marked by old summaries and the *changed* set of modified functions. Afterwards, eVolChecks performs the actual upgrade check by following the steps of Alg. 2. After its completion, the result of the change validation is returned to the user. If the upgrade is unsafe, an error is displayed, and the user is expected to fix the error. After the fix is done, it should be checked based on the latest correct version. Otherwise, the program is correct, the new call tree and the summaries are stored for the use by the next upgrade checking run.

Experiments. We evaluated eVolCheck on a set of industrial benchmarks. Four of them (`VTT_`n) were provided by our industrial partner, the VTT company. The rest is derived from a library of Windows device drivers (`floppy_`n, `kbfiltr_`n, `diskperf_`n). Safety of all benchmarks was verified against assertions, either existing in the code or inserted by us into code without assertions. Table II contains results of the experiments. Each row corresponds to a different benchmark, groups of columns represent statistics about the bootstrapping verification and verification of two upgrades, respectively. NoI estimates the size of the original source code as a number of instructions in the `goto-binary` (NoI is an accurate representation of code without definitions, and often represents much higher number of lines of code). The overhead introduced by upgrade checking, i.e. the syntactic difference check (Diff) and the interpolants generation (Itp), is also included in the total running time (Total). To show advantages of our upgrade checking approach, for each change we calculated the speedup (Speedup) of the upgrade check versus verification of the changed code from scratch, performed only for the sake of comparison reasons and not displayed in the table.

In order to demonstrate different performance of our technique, we chose two different classes of changes for each benchmark. The first class (1st change) represents changes with small impact. As expected, those can be verified with a few local checks. The second one (2nd change) presents upgrades that affect large portion of the code, potentially causing traversal of the complete call tree of the program.

Our experiments demonstrate that for the class of problem with small impact, the upgrade checking approach outperforms the standalone verification (order(s) of magnitude speedup). For the second class of changes, the performance of the upgrade check varies. For some cases, analysis could be done locally and the speedup is still substantial. For cases where the algorithms needed to analyze large portion of the call tree, the performance, as expected, degrades. Note that the bad performance occurs when the change introduces a bug (indicated by '—' in the Itp column; the PBMC formula is satisfiable and iterpolants are not generated). In this case, the upgrade check traverses to the root of the call tree, in order to reconstruct a complete error trace. Of course, this can be an easy task when the change is close to the root of the call tree (e.g., in the `floppy_D` benchmark). The results support our initial intuition that upgrade checking works well for incremental changes, which is the most common class in the evolution of systems.

V. 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 [12]. 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 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 in order 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 [12].

The authors of [13] study effects of code changes on function summaries used in compositional directed testing (a.k.a. white-box fuzzing). They use notion of must summaries as an under-approximation of the behavior. The goal of [13] 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. Our algorithm differs by using over-approximative interpolation-based function summaries and by performing the actual verification during the analysis.

Another group of related work aims at equivalence checking of programs [14], [15], [16]. Differential Symbolic Execution [14] attempts to show equivalence of two versions of code using symbolic execution or to compute a behavioral delta

[3]http://www.cprover.org/goto-cc/

[4]Two functions match iff their signatures are the same (function name, types and order of arguments, and return type).

Table II: Experimental evaluation

benchmark		bootstrap		1st change				2nd change			
name	NoI	Total [s]	Itp [s]	Total [s]	Diff [s]	Itp [s]	Speedup	Total [s]	Diff [s]	Itp [s]	Speedup
VTT_A	329	4.889	0.133	0.318	0.006	<0.001	**15.6**x	15.102	0.006	—	**0.3**x
VTT_B	332	23.178	0.003	7.793	0.007	0.007	**3.0**x	7.805	0.007	0.014	**3.0**x
VTT_C	129	0.144	0.001	0.017	0.002	<0.001	**8.4**x	0.187	0.002	—	**0.8**x
VTT_D	247	24.735	0.001	0.008	0.008	<0.001	**3098.0**x	46.910	0.006	—	**0.8**x
floppy_A	292	1.025	0.015	0.039	0.009	0.002	**26.1**x	0.201	0.009	0.013	**5.0**x
floppy_B	294	0.763	0.003	0.038	0.009	<0.001	**19.8**x	0.046	0.009	0.001	**16.4**x
floppy_C	2082	1.280	0.004	0.383	0.182	<0.001	**3.4**x	0.394	0.183	0.001	**3.2**x
floppy_D	2099	60.469	0.257	0.374	0.182	<0.001	**161.7**x	3.614	0.189	—	**16.8**x
kbfiltr_A	529	1.307	0.014	0.030	0.011	<0.001	**43.1**x	0.111	0.012	0.006	**10.6**x
kbfiltr_B	529	1.040	0.001	0.052	0.011	0.001	**19.6**x	1.835	0.011	—	**0.6**x
kbfiltr_C	1010	2.522	0.014	0.063	0.021	0.002	**40.2**x	0.124	0.021	0.002	**20.3**x
kbfiltr_D	1011	3.060	0.009	0.061	0.022	<0.001	**50.5**x	0.231	0.022	0.003	**7.0**x
diskperf_A	486	1.028	0.001	0.033	0.008	<0.001	**31.3**x	1.751	0.008	—	**0.6**x
diskperf_B	492	2.580	0.049	0.091	0.009	0.006	**28.3**x	2.468	0.009	0.029	**1.1**x
diskperf_C	1664	1.126	0.001	0.072	0.034	<0.001	**15.6**x	0.097	0.034	0.001	**11.5**x
diskperf_D	1685	38.609	1.179	0.295	0.035	0.016	**130.4**x	0.508	0.035	0.020	**75.7**x

when not equivalent. The comparison is function-by-function, the unchanged portions of code are abstracted by the same uninterpreted functions. A similar approach is implemented in the SymDiff tool [15], which decides conditional partial equivalence, i.e., equivalence under certain input constraints. Moreover, SymDiff also allows extraction of the constraints and reports them to the user. Regression Verification [16] employs model checking to prove partial equivalence of programs. As in our algorithm, regression verification starts with syntactic difference check, that identifies the set of modified functions. Then it also traverses the call graph bottom-up, and separately checks equivalence between the old and new versions of the of functions, while other functions are abstracted again using the same uninterpreted functions. In these approaches, if the versions do differ, the user is alerted and possibly informed what the different output is and for which input it occurs. For evolving systems, the versions almost always differ and thus the user is distracted by many such reports. In contrast, our algorithm focuses on checking safety of the versions with respect to assertion violation and the user is only alerted when a new violation is introduced by the change. Also, our approach may skip processing parts of the program, if they do not influence safety of the code.

Last group of related work includes approaches using interpolation-based function summaries (such as [4], [5], [6]). 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 [7].

VI. Conclusion

We presented a new upgrade checking algorithm using interpolation-based function summaries. Instead of model checking 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.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Tools and Alg. for the Const. and Anal. of Systems (TACAS '99)*, vol. 1579 of *LNCS*, pp. 193–207, 1999.

[2] S. Graf and H. Saïdi, "Construction of Abstract State Graphs with PVS," in *Computer Aided Verification, CAV '97*, LNCS, pp. 72–83, 1997.

[3] K. L. McMillan, "Applications of Craig Interpolation in Model Checking," in *Tools and Alg. for the Const. and Anal. of Systems (TACAS '05)*, vol. 3440 of *LNCS*, pp. 1–12, 2005.

[4] K. L. McMillan, "Lazy abstraction with interpolants," in *Computer Aided Verification (CAV '06)*, vol. 4144 of *LNCS*, pp. 123–136, 2006.

[5] K. L. McMillan, "Lazy annotation for program testing and verification," in *Computer Aided Verification (CAV' 10)*, vol. 6174 of *LNCS*, pp. 104–118, 2010.

[6] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "Whale: An Interpolation-Based Algorithm for Inter-procedural Verification," in *Verification, Model Checking, and Abstract Interpretation (VMCAI '12)*, vol. 7148 of *LNCS*, pp. 39–55, 2012.

[7] O. Sery, G. Fedyukovich, and N. Sharygina, "Interpolation-based function summaries in bounded model checking," in *Haifa Verification Conference (HVC '01)*, 2011. (to appear).

[8] D. Babić and A. J. Hu, "Structural Abstraction of Software Verification Conditions," in *Computer Aided Verification (CAV '07)*, vol. 4590 of *LNCS*, pp. 371–383, 2007.

[9] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," *J. of Symbolic Logic*, vol. 22, no. 3, pp. 269–285, 1957.

[10] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations," *J. of Symbolic Logic*, vol. 62, no. 3, pp. 981–998, 1997.

[11] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The OpenSMT Solver," in *Tools and Alg. for the Const. and Anal. of Systems (TACAS '10)*, vol. 6015 of *LNCS*, pp. 150–153, 2010.

[12] S. Chaki, E. Clarke, N. Sharygina, and N. Sinha, "Dynamic Component Substitutability Analysis," in *Int. Symp. of Formal Methods Europe (FM '05)*, vol. 3582 of *LNCS*, pp. 512–528, Springer, 2005.

[13] P. Godefroid, S. K. Lahiri, and C. Rubio-González, "Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation," in *Static Anal. Symp. (SAS '11)*, vol. 6887 of *LNCS*, 2011.

[14] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu, "Differential symbolic execution," in *Foundations of SW Engineering (FSE '08)*, pp. 226–237, 2008.

[15] M. Kawaguchi, S. K. Lahiri, and H. Rebelo, "Conditional equivalence," Tech. Rep. MSR-TR-2010-119, Microsoft Research, 2010.

[16] B. Godlin and O. Strichman, "Regression verification," in *Design Automation Conference (DAC '09)*, pp. 466–471, 2009.