eVolCheck: Incremental Upgrade Checker for C*

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

[1] University of Lugano, Switzerland, {name.surname}@usi.ch
[2] D3S, Faculty of Mathematics and Physics, Charles University, Czech Rep.

Abstract Software is not created at once. Rather, it grows incrementally version by version and evolves long after being first released. To be practical for software developers, the software verification tools should be able to cope with changes. In this paper, we present a tool, eVolCheck, that focuses on incremental verification of software as it evolves. During the software evolution the tool maintains abstractions of program functions, function summaries, derived using Craig interpolation. In each check, the function summaries are used to localize verification of an upgrade to analysis of the modified functions. Experimental evaluation on a range of various benchmarks shows substantial speedup of incremental upgrade checking of eVolCheck in contrast to checking each version from scratch.

1 Introduction

Software is rarely stable. Not only it gradually evolves during its development, but it is also subject to changes after it is released (e.g., bug fixes, component upgrades, platform changes, etc.). This evolution is an inherent part of software development and as such, it should be reflected also by the software verification tools. With this in mind, we developed a tool called eVolCheck, which focuses on incremental verification of software written in C.

The eVolCheck tool is a bounded model checker (BMC), which was specifically designed to handle incremental changes by focusing on the actual changes and to avoid resorting to the re-verification of the updated systems from scratch as most tools have to do in the presence of changes. In particular, it uses interpolation-based function summaries to localize and thus speedup the checks of new versions of a software. Concretely, eVolCheck maintains over-approximating summaries of all the program functions. After a change, it first attempts to verify that the old summaries are still valid for the changed program functions. Since this check considers only code of the function bodies, its old summary and potentially summaries of its callees, it is very local and thus it tends to be computationally inexpensive. If it succeeds, the upgrade is safe. Otherwise, the check is propagated to the callers of the modified functions. When the summary of the call tree root is shown to be violated, a real error is found and it is reported to the user along with an error trace. After each successful check, any invalidated summaries are regenerated so that they are ready for the check of the next version. In addition, eVolCheck features a counter-example guided refinement to deal with too coarse summaries during the checks.

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

The upgrade checking algorithm was originally described in [18] along with a discussion on its correctness. This paper 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.

The paper is structured as follows. In Section 2, we review the theoretical background of the algorithm with references for more detailed explanation. Section 3 describes the architecture of the `eVolCheck` tool together with the essential implementation details, while Section 4 focuses on the usage of the tool and its integration into `Eclipse`. In Section 5 we present experimental evaluation on various benchmarks. We list the related work in Section 6 and conclude in Section 7.

2 Background Theory

This section focuses on the theoretical background of the upgrade checking algorithm which is core of `eVolCheck`.

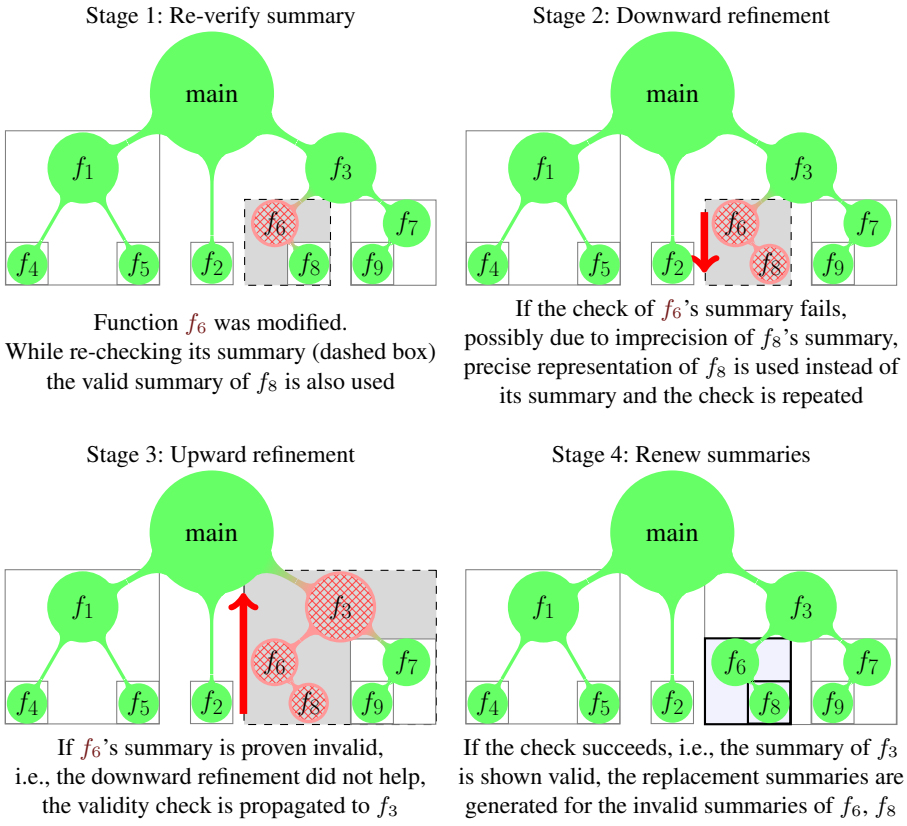

Figure 1: eVolCheck principle

Upgrade checking. During the software evolution, `eVolCheck` maintains over-approximations of input/output behaviors of all functions in the code, i.e., function summaries. Initially, the function summaries are generated during a bootstrapping run, which is equivalent to the standalone verification and implements the approach from [16]. Then, the function summaries are validated (some potentially replaced) during each successful upgrade check.

In each check, `eVolCheck` first identifies the set of modified functions on the syntactical level. For this purpose, we developed a tool called `goto-diff` which effectively detects the semantic changes (more details are in Section 3). Then `eVolCheck` attempts to show that the old function summaries are still valid over-approximations of the behavior of the modified functions (Stage 1 in Fig. 1). These are local and thus cheap checks. As an option of `eVolCheck`, to further speed up the check, all the valid summaries may be used in this check to abstract the corresponding function calls.

If the local checks succeed, the upgraded version is safe. If not, it can be either because the valid summaries of the called functions are not precise enough, in which case they are replaced by their precise representation, by performing *downward refinement* (Stage 2 in Fig. 1). Or it can be because the summary is indeed violated during the change, in which case the check is propagated to the parent functions, by performing *upward refinement* (Stage 3 in Fig. 1). This is iterated until either the check succeeds (Stage 4 in Fig. 1) on some level of the call tree, or the check fails for the `main` function in the root of the call tree. In the former case, new valid summaries are generated for the subtree, while in the latter case, a real error is identified and reported to the user.

As a result `eVolCheck` exploits the locality of the changes, which makes it a valuable tool for efficient verification of fine-grained changes that have limited impact throughout the code. Of course, should the change be extensive and span the entire code base, naturally, the check can become expensive. This is in line with the envisioned position of the tool in the development process to check changes on the level of individual commits rather than major revisions.

Interpolation. The upgrade checking algorithm is based on over-approximating function summaries, however, it is not strictly tied to any particular form of abstraction or means to derive the summaries. Of course, the particular summaries need to satisfy certain properties, e.g., the correctness Properties 1 and 2 (formally defined later in this Section), used to show correctness of the algorithm. Our implementation of the algorithm in `eVolCheck` uses function summaries derived by Craig interpolation [5]. In a nutshell, given two formulas A and B such that $A \wedge B$ is unsatisfiable, a Craig interpolant of A and B is a formula I, s.t., $A \implies I$, and $I \wedge B$ is unsatisfiable, and I contains only the shared free variables of A and B.

Intuitively, interpolants are an over-approximation of formula A still capturing the conflict with B, while using only the shared language of (A, B). Craig interpolants are usually constructed from a resolution proof of unsatisfiability of $A \wedge B$ and they have numerous applications in model checking (see, e.g., [11]). Note that interpolants of different strength (considering implication relation) can be obtained using different interpolation algorithms. In practice, additional properties of multiple interpolants generated from a single unsatisfiable formula are often required, resulting in *path interpolants* and *tree interpolants*. Note that it is often possible to ensure these additional properties by

careful construction of interpolants from the same proof of unsatisfiability [15].

Definition 1. *Let $A \wedge B \wedge C$ be an unsatisfiable formula and I_A, I_B, I_{AB} be Craig interpolants of $(A, B \wedge C)$, $(B, A \wedge C)$, and $(A \wedge B, C)$ respectively. The interpolants I_A, I_B, I_{AB} have the* tree interpolant *property iff $I_A \wedge I_B \implies I_{AB}$.*

In the implementation of `eVolCheck` algorithms, the tree interpolant property is essential as it must be satisfied to maintain valid function summaries and to ensure the correctness of the overall local upgrade checking.

Function summarization. Standard BMC creates a monolithic formula not well suited for interpolation, as symbols of different scopes get mixed in the formula both due to the encoding and optimizations. To solve this problem, we create a so called *partitioned bounded model checking formula* (PBMC formula) that isolates variables of functions in separate conjuncts of the formula and shares only the interface symbols of functions. That is input and output parameters[3] and a few helper symbols, as further explained in [16,17]. In particular, for each function call f, there is a helper propositional variable $error_f$, that evaluates to true when an error (assertion violation) is reachable in that function given the valuation of its input parameters.

When the PBMC formula is unsatisfiable, it is easy to partition it for interpolant generation for each function call, so that A corresponds to the function implementation (including its callees) and B to the calling context. The generated interpolants are then over-approximations of the functions input/output behavior and contain only the interface variables of the functions. In other words, the interpolants constitute over-approximating function summaries.

Correctness of upgrade checking. The correctness of the local upgrade checking algorithm is based on maintaining the following two properties:

$$error_{f_{main}} \wedge \sigma_{f_{main}} \rightarrow \bot \tag{1}$$

Given each function call f and its children calls $g_1, \ldots, g_n$:

$$\sigma_{g_1} \wedge \ldots \wedge \sigma_{g_n} \wedge \phi_f \rightarrow \sigma_f \tag{2}$$

Property 1 claims that the entire program is safe by the means of the summary of the `main` function, $\sigma_{f_{main}}$, and its inconsistency with the $error_{f_{main}}$ capturing reachability of an error in the call tree of `main`, i.e., the entire program.

Property 2 requires that the summaries of callees (σ_{g_i}) along with precise representation of the body of the caller (ϕ_f) are captured by the summary of the caller (σ_f). In other words, that the over-approximations of the callees are not too weak to be captured by the over-approximation of the caller.

With these two properties, correctness of the upgrade checking algorithm is easy to see. It suffices to recursively apply Property 2 to replace summaries occurring in Property 1. The results state that the precisely encoded program is error free.

In [18], we showed that the properties are established during the initial bootstrapping run, when all the summaries are generated, and that they are reestablished after

[3] Note that accessed global variables are handled as additional input/output arguments.

each successful run of the upgrade checking algorithm. The proof relies on the tree interpolant property. This becomes transparent when the inductive nature of both Property 2 and Def. 1 is observed side by side.

3 Tool Architecture

This section presents the architecture of the `eVolCheck` tool as depicted in Fig. 2. The tool uses the `goto-cc` compiler provided by the CProver framework[4]. 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. The resulting models are stored for future checks.

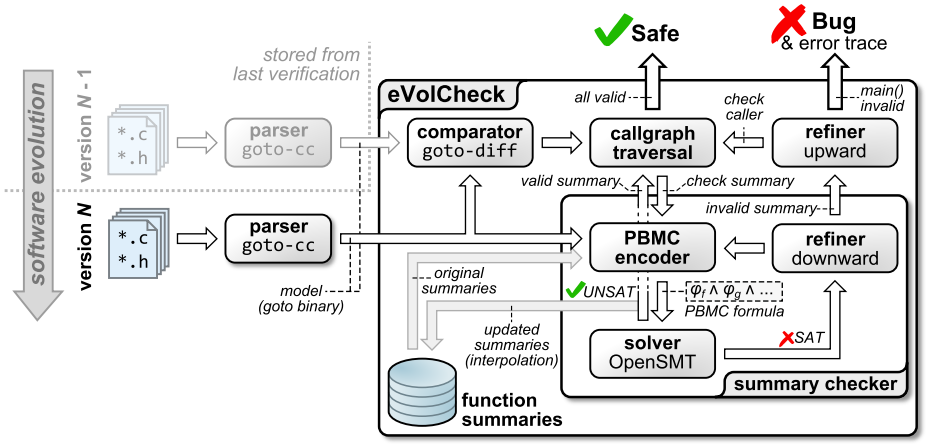

Figure 2: eVolCheck architecture overview

eVolCheck. The `eVolCheck` tool itself consists of a comparator that identifies the changed functions and function calls, a call graph traversal that attempts to check summaries of all the modified functions bottom up, an upward refiner that identifies parent functions to be rechecked when a summary check fails, and a summary checker that performs the actual check of a function against its summary. The summary checker in turn consists of a PBMC encoder that takes care of unwinding loops and recursion, generation of SSA form and bit-blasting, a solver wrapper that takes care of communication with the solver/interpolator (`OpenSMT` [2]), and a downward refiner that identifies ancestor functions to be refined when a summary check fails possibly due to imprecise representation of the ancestor function calls. 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 w.r.t. variables, irrelevant to the properties being checked. The second can compare the existent summaries for the same function and the same bound, and keep the more precise one.

[4] `www.cprover.org`

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 the models function by function. It uses the longest common subsequence algorithm 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 then 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 Section 2, to guarantee correctness of the upgrade check, `eVolCheck` requires a solver that is able to generate multiple interpolants with the tree interpolant property from a single satisfiability query. For this reason, we use the interpolating solver, `OpenSMT`, which creates multiple interpolants from the same unsatisfiability proof and provides API for convenient specification of the partitions corresponding to the functions in the call tree. Currently, we use `OpenSMT` in the SAT solving mode and bit-blast all formulas to the propositional level. As a result, `eVolCheck` provides bit-precise reasoning.

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 `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 toolset 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 verification/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.

4 Tool usage

The `eVolCheck` can be run from a command line as well as using the `Eclipse` plug-in. Its Linux binaries, benchmarks used for evaluation, a tutorial explaining how to use

eVolCheck and explanation of the most important parameters are available on-line for other researchers[5].

The following shows the example of usage of eVolCheck from a command line [6]:

1. Create a model of the base version of the program by running the goto-cc compiler. Choose one of the *_orig.c files in examples directory, for example by typing:

```
~/evolcheck$ ./goto-cc examples/valid/change_valid_orig.c
   -o examples/valid/change_valid_orig.out
```

The file examples/valid/change_valid_orig.c is the input source code and examples/valid/change_valid_orig.out is the resulting goto-binary. Note that the upgrade checking environment should be prepared for analysis by cleaning the repository with ~/evolcheck$ rm __summaries __omega before performing this step.

2. Run eVolCheck to perform the initial bootstrapping check of the program (parameter --init-upgrade-check):

```
~/evolcheck$ ./evolcheck --init-upgrade-check --unwind 10
   examples/valid/change_valid_orig.out
```

Note that the parameter --unwind <N> is optional and specifies the maximal number of unwindings of each loop.

3. Check the eVolCheck outputs. The following message at the end of the eVolCheck output indicates either that the program is safe:

```
ASSERTION(S) HOLD(S).
```

or that the program is buggy:

```
ASSERTION(S) DO(ES)N'T HOLD.
A real bug found.
```

In the latter case, a corresponding error trace manifesting the bug is part of the output as well. After a successful bootstrapping check, the summaries and their mapping to the calltree are created and stored for the subsequent upgrade checks in files __summaries and __omega respectively.

4. When the program is upgraded, goto-binary model of the new version is created again using the goto-cc compiler. Run it for the file corresponding *_upgr.c file chosen in Step 2:

```
~/evolcheck$ ./goto-cc examples/valid/change_valid_upgr.c
             -o examples/valid/change_valid_upgr.out
```

5. With the goto-binary model of the new version of the program, the actual upgrade check is performed (parameter --do-upgrade-check <file>) as follows:

```
~evolcheck$ ./evolcheck --do-upgrade-check examples/valid/change_valid_upgr.out
            --unwind 10 examples/valid/change_valid_orig.out
```

6. Check the eVolCheck output. There are several possible cases. Either the two programs have identical models, i.e., no or only simple syntactical changes occurred (examples for this case are located in the examples/ident folder), resulting in the following output:

[5] www.verify.inf.usi.ch/evolcheck

[6] the running example can be found at http://www.inf.usi.ch/phd/fedyukovich/evolcheck.lin32.tar.gz

```
The program models are identical.
```

Or the upgraded program was changed but it remains correct (examples are located in the `examples/valid` folder), resulting in the following message for each checked function summary:

```
... summary was verified.
```

Or the upgraded program is buggy (examples are located in `examples/not_valid`). The corresponding output contains the following message for the summary of the function `main`: ‘

```
Old summary is no more valid.
...
summary cannot be renewed. A real bug found.
```

7. Additional information about the usage of the tool can be found simply by typing

```
~/evolcheck$ ./evolcheck --help
```

Eclipse plug-in. Nowadays, IDEs form an essential part of software development tool chains. Therefore, we integrated `eVolCheck` into `Eclipse`, which is one of the most widely used IDE. Our plug-in hides some of the implementation details and provides much more comfort compared to the command line tool. As expected, the actual use of the plug-in follows the command line scenario.

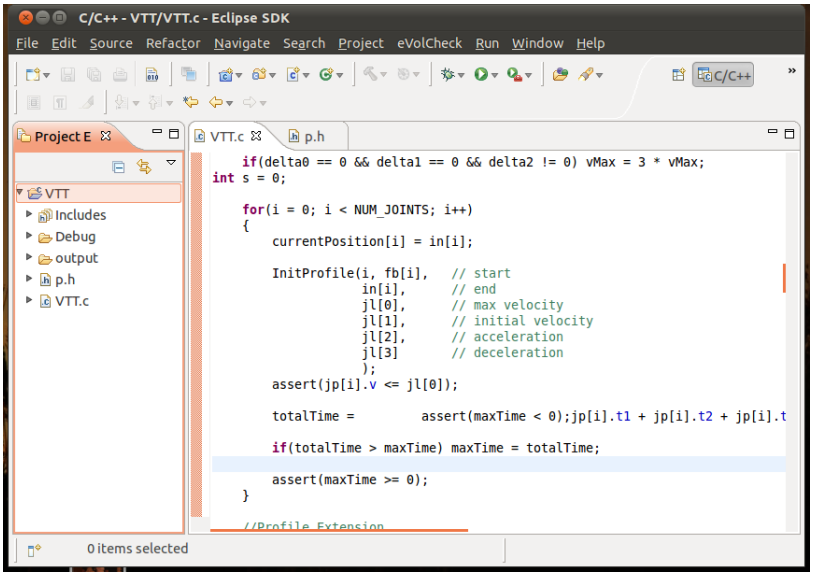

Figure 3: Eclipse developing prospective

1. The user develops a base version of the program. In order to specify properties, the assertions should be placed in the code (Fig. 3) or generated automatically by the tool. The examples of of the default properties are division by zero, 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 unwinding bound (Fig. 4). `Eclipse` then automatically creates the model (goto-binary) from the selected source files and keeps working with it.

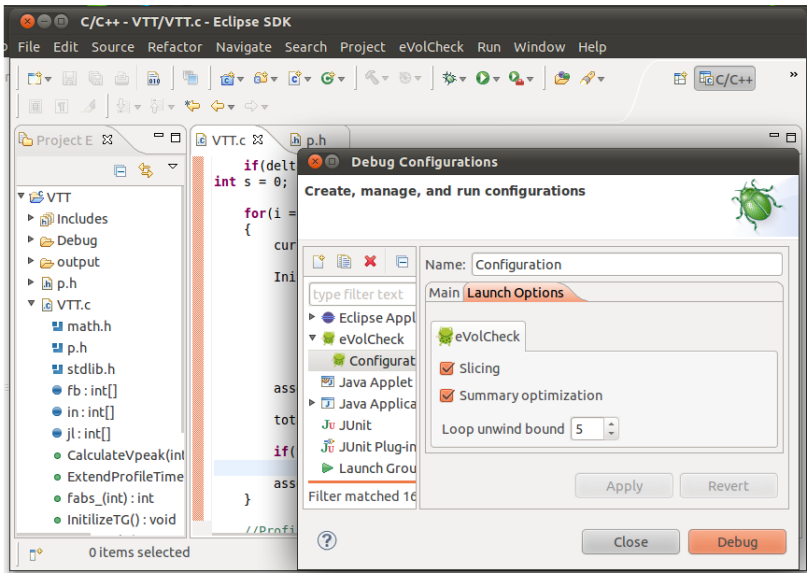

Figure 4: eVolCheck configuration window

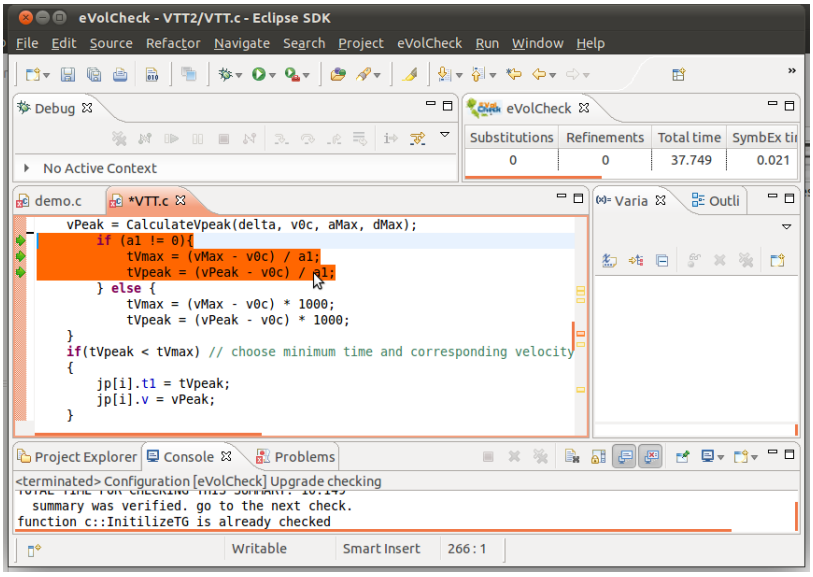

Figure 5: eVolCheck invokes goto-diff (changed lines are highlighted)

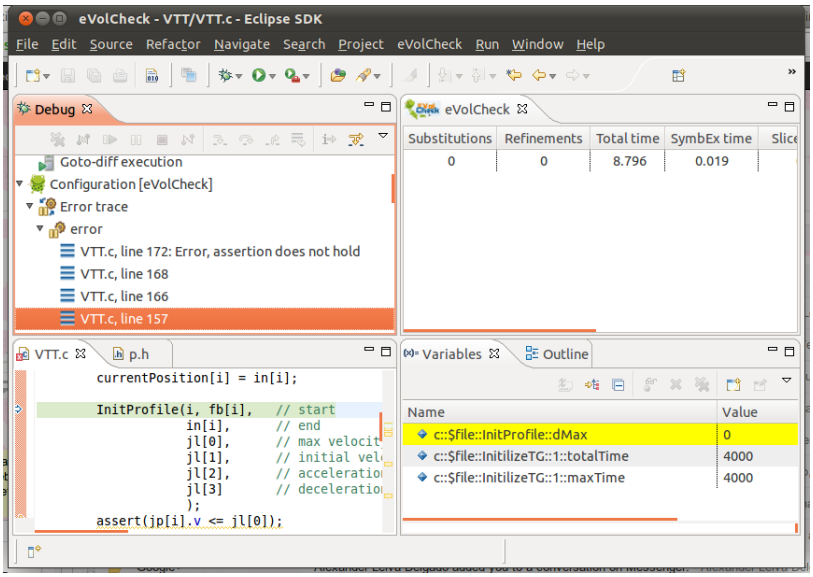

Figure 6: eVolCheck error trace

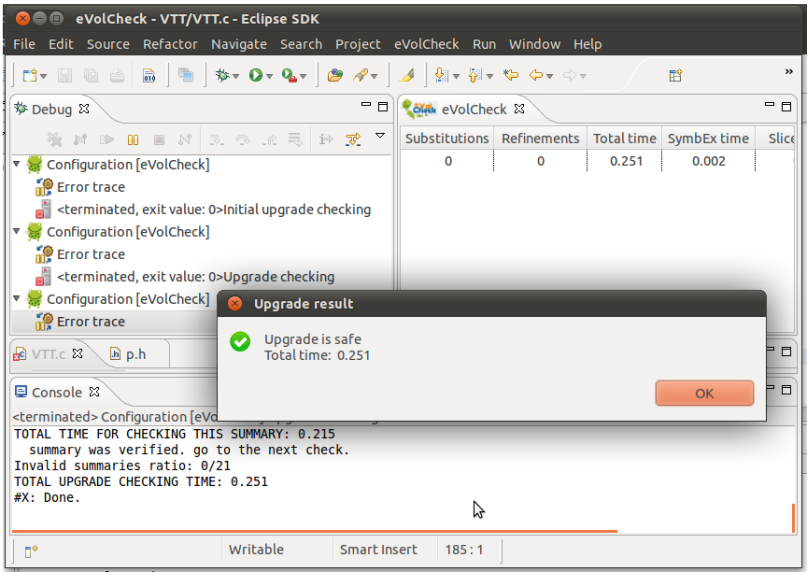

Figure 7: eVolCheck successful verification report

3. The plug-in searches for the last safe version of the current program (goto-binary created from the same selection of source files). 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 call trees for both the current and the previous versions. The modified lines of code are marked (Fig. 5) for the user review.

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. 6). 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. 7). 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 colored call-tree is available (Fig. 8).

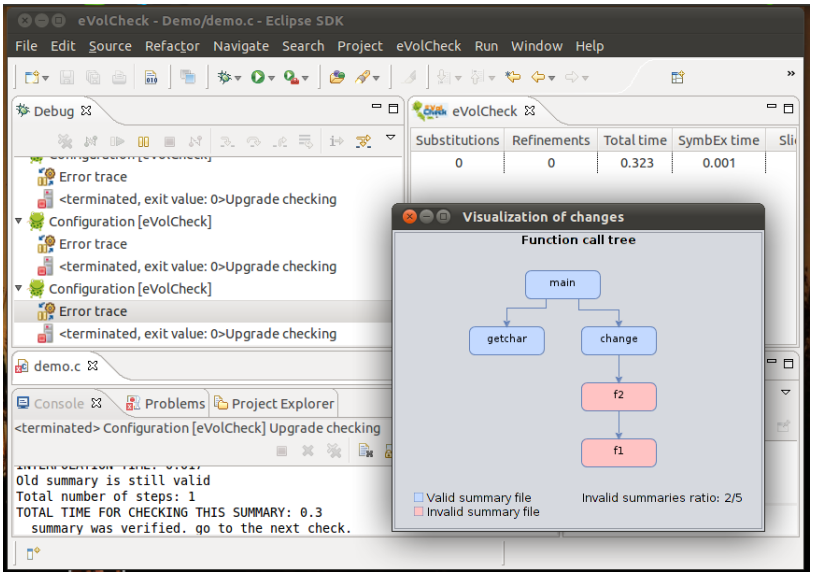

Figure 8: eVolCheck change impact

5 Evaluation

In addition to the standalone use of `eVolCheck` (as described in Section 4), the tool is used as a static analysis engine within the hybrid static/dynamic upgrade checking platform developed as part of the Pincette project[7]. The platform has been applied to analysis of software developed by Pincette's industrial partners, among which there are

[7] `www.pincette-project.eu`

Table 1: Experimental evaluation

Benchmark	Bootstrap		Upgrade check					
Name	Total [s]	Itp [s]	Total [s]	Diff [s]	Itp [s]	**Speedup**	Result	ISR
ABB_A	8.644	0.008	0.04	0.009	0.003	**220**x	SAFE	0/7
ABB_B	6.236	0.009	0.006	0.006	—	**935**x	SAFE	0/9
ABB_C	8.532	0.015	0.059	0.008	0.003	**157**x	SAFE	0/8
VTT_A	0.512	0.001	0.006	0.006	—	**85.5**x	SAFE	0/9
VTT_B	0.514	0.001	0.031	0.006	—	**0.7**x	BUG	1/9
euler_A	12.56	0.099	0.179	0.001	0.016	**70.4**x	SAFE	1/6
euler_B	12.547	0.095	2.622	0.001	0.031	**4.74**x	SAFE	3/5
life_A	13.911	1.366	0.181	0.001	<0.001	**77.0**x	SAFE	0/5
life_B	13.891	1.357	6.774	0.001	—	**0.31**x	BUG	5/5
arithm_A	0.147	0.007	0.355	0.001	—	**0.39**x	BUG	3/3
diskperf_A	0.167	0.001	0.024	0.008	<0.001	**5.79**x	SAFE	0/21
diskperf_B	0.137	0.001	0.062	0.009	—	**2.25**x	BUG	3/21
floppy_A	2.146	0.229	0.422	0.202	<0.001	**5.02**x	SAFE	0/226
floppy_B	2.183	0.237	2.277	0.206	—	**0.82**x	BUG	79/226
kbfiltr_A	0.288	0.011	0.081	0.023	0.001	**3.40**x	SAFE	1/63
kbfiltr_B	0.320	0.009	0.088	0.023	0.001	**1.85**x	SAFE	3/63

the VTT company with its control software for a maintenance robot for the ITER fusion reactor; the IAI company with the software for a stabilized optical device payload (MSEOS) of their unmanned airborne vehicles; and the ABB company with the software of their power grid protection units. As part of the project, eVolCheck is also integrated in the CCRT platform [8], a collaborative code review tool developed at IBM.

The eVolCheck tool was validated on a wide-range of various benchmarks among which are the validation cases, provided by the Pincette project collaborators. In particular, it was used to verify the C part of the implementation of the DTP2 robot controller, developed by the VTT company. It was also applied to the ABB validation cases on a code taken from the project implementing a core of a feeder protector and controller. The code originates from an embedded software used in the ABB hardware module. This is a large scale project containing many sub-projects which implement various functions of the feeder device. The total number of lines in the overall code is in millions. 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.

To demonstrate the applicability and advantages of eVolCheck, we provide evaluation details of several test cases. Five of them (ABB_n, VTT_n) were provided by the Pincette project partners for which the changes were extracted from the project repositories. Six other benchmarks were derived from Windows device driver library (diskperf_n, floppy_n, kbfiltr_n). The changes were introduced manually there. The rest of the benchmarks are the masters' student projects conducted at University of Lugano.

Table 1 represents results of the experiments. Each benchmark is shown in a separate

[8] CCRT is a proprietary tool of IBM

row, which summarizes statistics about the initial verification and verification of an upgrade. Time (in seconds) for running the syntactic difference check (**Diff**) and for generation of the interpolants (**Itp**) represents the computational overhead of the upgrade checking procedure, and included in the total running time (**Total**) of `eVolCheck`. Note that interpolation can not be performed at the buggy examples (marked as "—"), for which the corresponded PBMC formula is satisfiable. To show advantages of our upgrade checking approach, 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 posteriori estimation of the upgrade check complexity is shown in the row **ISR** (Invalid Summaries Ratio). This ratio represents the number of invalid summaries (due to the change) with respect to the number of nodes in the call tree of the verified program.

Discussion. Our evaluation demonstrates good performance of `eVolCheck`. In particular, the experiments show high efficiency of upgrade checking for safe upgrades since they result in a small number of refinements (both, upward and downward). This generally leads to a small number of invalidated summaries, as witnessed by the corresponding **ISR** (see, for example, the `ABB_`n cases, where summaries of all changed functions were proven valid). It is less efficient (for some tests) in case of buggy upgrades, since bugs frequently (as expected) effect larger portions of the program. In classical model checking, confirming the absence of bugs is usually more expensive (since it requires the full state-space search) then detecting the bugs (where the search can be terminated once the bug is detected) and we believe that the fact that `eVolCheck` works so well to confirm safety is very useful for routine analysis of upgrades.

The use of `goto-diff` has been very useful since it managed to detected many test cases with small syntactic changes which did not require running the main `eVolCheck` procedures. For example, in `VTT_A` and `ABB_B`, the comparator proved that the models are identical, 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.

6 Related Work

The general idea of interpolation-based function summarization was studied in various projects including earlier work of `eVolCheck` authors [12,13,1,16,8]. For instance, the authors of [8] generate sequence of inductive interpolants as summaries of recursive functions to be used in Hoare-style verification of a single (not upgraded) program. To our knowledge, they do not have implementation and do not extend the idea to upgrade checking. A tool called `FunFrog` [17] is an implementation of the idea from [16]. To the best of our knowledge, `eVolCheck` is the first tool which further extends interpolation-based function summaries to incremental checking of software upgrades.

Previously, other researchers attempted to reuse (parts of) models constructed during verification of the base version to speed up verification of software upgrades. Either,

the models constituted an entire reachable abstract state space [9,4] that was revalidated after a change. Alternatively, behavioral models of different components of the software [3] were constructed (by employing techniques for learning regular languages) and then substitutability between the original and the altered components was analyzed after each change. In comparison, `eVolCheck` stores information corresponding to the function calls. We argue that this is a natural abstraction boundary that is more stable than the abstract state space and at the same time more fine-grained than the entire software components.

There are also approaches that attempt to show equivalence of the original and the upgraded software [14,10,7]. The `SymDiff` tool [10] decides conditional partial equivalence, i.e., equivalence under certain input constraints. Moreover, `SymDiff` also allows automated extraction of the constraints and reports them to the user. The goal of *differential symbolic execution* [14] is to show equivalence of the two versions using symbolic execution. If the versions are not equivalent, a behavioral delta is constructed as a feedback for the user. In [7], a technique called *regression verification* for deciding partial equivalence of programs using model checking is introduced. As well as `eVolCheck`, regression verification starts with a syntactic difference check identifying the modified functions. Then the call graph is traversed starting from the leafs. During the traversal, old and new versions of each visited and possibly affected function is checked for equivalence. In this check, any called functions are abstracted using the same uninterpreted functions.

If we compare these approaches to `eVolCheck`, the fundamental difference is that `eVolCheck` does not care about equivalence. It only checks that no errors sneak in the code with the upgrade. This means that the equivalence-based approaches report all the nonequivalent behavior to the user, flooding the user with information in the process. In contrast, `eVolCheck` reports only the bugs added to the code, which we believe the users are really interested in. Another related benefit is that `eVolCheck` may skip processing parts of the code base (which could be hefty) that do not affect correctness of the upgrade.

In the context of compositional directed testing (a.k.a. white-box fuzzing), some authors study effects of upgrades on function summaries [6] with the goal to identify the affected summaries unusable for analysis of the new version. With the unusable summaries removed, the preserved ones are employed in the actual analysis as a second step. When compared, `eVolCheck` uses over-approximative interpolation-based function summaries and performs the actual verification during the analysis not separately.

7 Conclusion

This paper presented the incremental upgrade checker, `eVolCheck` along with its integration into the `Eclipse` development environment. This is the first tool which uses interpolation-based function summaries to localize and speed up the upgrade check. The tool was evaluated on a range of industrial and academic examples, and in the most cases showed notable speedup with relation to verification from scratch. In future, we would like to explore the possibility to use the information regarding the failed and successful intermediate summary checks in order to guide the user in finding the root cause

of the error, e.g., by emphasizing portions of the reported error trace corresponding to the failed intermediate summary checks. In addition, we consider integration with a versioning system (e.g., SVN), which would allow further integration into the software development process.

Acknowledgments. 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 comments on usability and correcting our English, Michael Tautschnig for his help with `CProver` and `goto-cc` adjustments, and Pincette validators for assistance with the industrial test cases.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In: VMCAI '12. LNCS, vol. 7148, pp. 39–55 (2012)
2. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Tools and Alg. for Con. and Anal. of Sys. (TACAS '10). LNCS, vol. 6015, pp. 150–153 (2010)
3. Chaki, S., Clarke, E., Sharygina, N., Sinha, N.: Dynamic Component Substitutability Analysis. In: FM '05. LNCS, vol. 3582, pp. 512–528. Springer (2005)
4. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: CAV '05. LNCS, vol. 3576, pp. 449–461 (2005)
5. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. of Symbolic Logic pp. 269–285 (1957)
6. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In: SAS '11. LNCS, vol. 6887 (2011)
7. Godlin, B., Strichman, O.: Regression verification. In: DAC '09. pp. 466–471 (2009)
8. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Principles of Prog. Languages (POPL '10). pp. 471–482. ACM (2010)
9. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme Model Checking. In: Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358 (2003)
10. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: a language-agnostic semantic diff tool for imperative programs. In: CAV '12. LNCS, vol. 7358, pp. 712–717 (2012)
11. McMillan, K.L.: Applications of Craig Interpolation in Model Checking. In: Tools and Alg. for Con. and Anal. of Sys. (TACAS '05). pp. 1–12. LNCS (2005)
12. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Computer Aided Verification (CAV '06). pp. 123–136. LNCS (2006)
13. McMillan, K.L.: Lazy Annotation for Program Testing and Verification. In: Computer Aided Verification (CAV' 10). pp. 104–118. LNCS (2010)
14. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: FSE '08. pp. 226–237 (2008)
15. Rollini, S.F., Sery, O., Sharygina, N.: Leveraging interpolant strength in model checking. In: Computer Aided Verification (CAV '12). LNCS, vol. 7358, pp. 193–209 (2012)
16. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based Function Summaries in Bounded Model Checking. In: HVC '11. LNCS, vol. 7261, pp. 257–272 (2011)
17. Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: Bounded Model Checking with Interpolation-based Function Summarization. In: ATVA '12. LNCS, vol. 7561, pp. 203–207 (2012)
18. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental Upgrade Checking by Means of Interpolation-based Function Summaries. In: FMCAD '12. LNCS (2012), to appear