PeRIPLO: A Framework for Producing Effective Interpolants in SAT-based Software Verification*

Simone Fulvio Rollini, Leonardo Alt, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina

Faculty of Informatics, University of Lugano
Via Giuseppe Buffi 13, CH-6904 Lugano, Switzerland

Abstract. Propositional interpolation is widely used as a means of overapproximation to achieve efficient SAT-based symbolic model checking. Different verification applications exploit interpolants for different purposes; it is unlikely that a single interpolation procedure could provide interpolants fit for all cases. This paper describes the PeRIPLO framework, an interpolating SAT-solver that implements a set of techniques to generate and manipulate interpolants for different model checking tasks. We demonstrate the flexibility of the framework in two software bounded model checking applications: verification of a given source code incrementally with respect to various properties, and verification of software upgrades with respect to a fixed set of properties. Both applications use interpolation for generating function summaries. Our systematic experimental investigation shows that size and logical strength of interpolants significantly affect verification, that these characteristics depend on the role played by interpolants, and that therefore techniques for tuning size and strength can be used to customize interpolants in different applications.

1 Introduction

A common approach for verifying a program is to express its behavior in a symbolic form and to check such representation against a given specification. One of the most appreciated techniques based on symbolic reasoning is SAT-based symbolic model checking [1], where both the program and the specification are encoded as an instance of the propositional satisfiability problem (SAT), and a SAT-solver is used to determine whether the specification is satisfied or violated. The SAT-based approach allows bit-level reasoning, important both in software and hardware applications, e.g., when dealing with pointer arithmetic and overflow. Successful tools exist for SAT-based verification include CBMC, SATABS, and CPAchecker.

In the last years, Craig interpolation [2] has been widely adopted as a means for abstraction in symbolic model checking [3]. Interpolants are usually computed from resolution refutations; several interpolation algorithms exist in the

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

literature [3,4,5] and different interpolants can be generated from the same refutation. While interpolation-based verification is critically affected by the quality of the generated interpolants, it is still not clear what makes an interpolant *good* in a particular framework. Two characteristics that have shown promise are logical strength and size: [5,6] suggest that weaker or stronger interpolants might be more appropriate for different applications, while [7] provides evidence that compact interpolants are beneficial in hardware model checking.

This paper addresses the problem of generating effective interpolants in the context of SAT-based Bounded Model Checking (BMC) [8] for software, and studies the impact of size and strength in verification. Specifically, we present the PeRIPLO[1] framework and discuss its ability to drive interpolation by providing routines that act on complementary levels: (i) manipulation (including compression) of the resolution refutations generated by a SAT-solver, from which interpolants are computed, and (ii) systematic variation of the strength of the interpolants, as allowed by the Labeled Interpolation Systems [5].

As case studies we consider two applications of BMC: verification of a C program incrementally with respect to a number of different properties (as in the FunFrog tool [9]), and incremental verification of different versions of a C program with respect to a fixed set of properties (as in the eVolCheck tool [10]). Both applications rely on interpolation to generate abstractions of the behavior of function calls (*function summaries*); the goal of summarization is to store and reuse information about already analyzed portions of a program, to make subsequent verification checks more efficient. If summaries (i.e. interpolants) are fit, a remarkable performance improvement is usually achieved; if spurious errors have been introduced due to over-approximation, (some of) the summaries need to be refined, which might be resource-consuming. The challenge we address is to use PeRIPLO to drive the generation of interpolants so as to obtain effective summaries.

The novelty of our work lies in the following contributions:

- An interpolation framework, PeRIPLO, able to generate individual interpolants and collections of interpolants satisfying particular properties. PeRIPLO offers a set of tunable techniques to manipulate refutations and to obtain interpolants of different strength from them; it can be integrated in any SAT-based verification framework which makes use of interpolants.
- Solid experimental evidence that compact interpolants improve performance in the context of software BMC. To the best of our knowledge, the only previous work to concretely assess the impact of the size of interpolants is [7], which addresses the use of interpolants in hardware model checking.
- A first systematic evaluation of the impact of interpolant strength in a specific verification domain. We target function summarization in software BMC and show that interpolants of different strength are beneficial to different applications; in particular, stronger and weaker interpolants are respectively suitable for the FunFrog and eVolCheck approaches. These results match the intuition behind the use of interpolants as function summaries.

[1] PeRIPLO can be found at `http://verify.inf.unisi.ch/periplo.html`

2 Interpolation

Craig interpolation [2] has found successful application in the context of model checking and it is at the base of techniques like predicate abstraction [6], counterexample guided abstraction refinement [11], interpolation-based function summarization [9], upgrade checking [10], and lazy abstraction with interpolants [12]. Formally, given an unsatisfiable conjunction $A \wedge B$, an interpolant I is a formula implied by A ($A \rightarrow I$), unsatisfiable with B (i.e., $B \wedge I \rightarrow \bot$) and defined on the common symbols of A and B. In other words an interpolant can be seen as an over-approximation of A that is still unsatisfiable with B.

Several algorithms are available to construct different interpolants for an unsatisfiable conjunction $A \wedge B$; yet, it is still an open problem to identify what characteristics make some interpolants better than others in a particular verification framework. In this paper we target two features which are intuitively relevant to model checking and for which preliminary evidence has been provided in the recent literature: *logical strength* [6,5] and *structural size* [7] (intended as the number of logical connectives in a formula).

Strength. A formula ϕ is said to be *stronger* than ψ if $\phi \rightarrow \psi$ (resp. ψ is *weaker* than ϕ). Interpolants are inherently over-approximations, thus a stronger or weaker interpolant is expected to drive verification in terms of a finer or coarser approximation. [5] offers an adequate framework to conduct an investigation of interpolant strength: it in fact presents the *Labeled Interpolation Systems* (LISs) for systematically building propositional interpolants of different strength from a single resolution refutation, generalizing the algorithms previously introduced by Pudlák [4] and McMillan [13]. A LIS is a procedure that, given a refutation of $A \wedge B$ and a labeling function, outputs an interpolant for $A \wedge B$. The authors define a partial order over the labeling functions and relate the corresponding interpolants by strength; [5] proves that the collection of systems represents a *complete lattice*, where McMillan's system M is the greatest element (i.e., it generates the strongest interpolant), the system M' dual to McMillan's is the least (i.e., it generates the weakest interpolant) and Pudlák's P is in between.

Size. Besides semantic features like strength, syntactic features like interpolant size are also likely to affect the verification performance: generating, storing and using smaller and less redundant formulae require fewer computational resources. Supporting evidence is given by [7], where compact interpolants prove beneficial in the context of hardware unbounded model checking. The usefulness of small interpolants is also intuitively clear for the function summarization based approaches considered in this paper, where interpolants correspond to summaries that are used multiple times in subsequent verification attempts.

Reduction of the interpolant size can be achieved both in an indirect and in a direct manner. Interpolants are computed from refutations, and their size is linear in the number of nodes of the DAGs representing the refutations. A simple indirect way to obtain a smaller interpolant is to first compress the refutation and then to apply an interpolation procedure; several compression algorithms exist in the literature, ranging from structural hashing to partial regularization [14,15,16,17,18]. A second way, complementary to proof compression, is to view interpolants as Boolean circuits and address them directly by means of logic synthesis techniques, including BDD sweeping, functional reduction and multi-level structural and functional hashing [19,7].

3 PeRIPLO

PeRIPLO (Proof tRansformer and Interpolator for Propositional LOgic) is an open-source SAT solver, built on MiniSAT 2.2.0 [20], that provides proof logging, proof manipulation routines and propositional interpolation. It can be used as a standalone tool or as a library; its routines are accessible via configuration file or API. Figure 1 illustrates the tool architecture.

PeRIPLO receives as input a propositional formula ϕ from the *verification environment*, and passes it to the *SAT solver*, that checks satisfiability while performing proof logging. If the formula is unsatisfiable, a resolution refutation Π is built in form of a directed acyclic graph.

Π can be further processed by the *proof transformer*, for example it can be compressed or manipulated as a preliminary step to interpolation.

Once Π is available, the environment can ask the *interpolator* for the generation of an individual or a collection of interpolants $\{I_i\}$ by means of an interpolation system Itp, providing a subdivision of ϕ into $A \wedge B$; if the collection is related to some interpolation property P, then an additional checking phase can be enabled to ensure that P is satisfied.

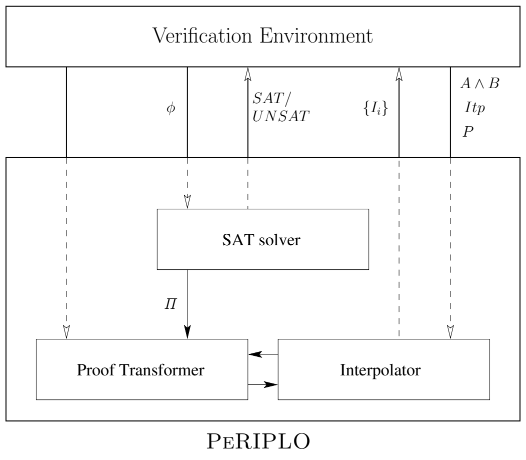


Fig. 1. PeRIPLO architecture.

Interpolant strength. PeRIPLO realizes the Labeled Interpolation Systems of [5] and allows to systematically vary the strength of the interpolants. It is able to produce both individual interpolants and collections of interpolants, w.r.t. various interpolation properties (e.g., tree interpolation, see §5) and in accordance with the constraints posed by the properties on the LISs [21,22].

Proof compression. PeRIPLO allows to compress refutations by means of the following techniques, which target different kinds of redundancies in proofs: (i) the `RecyclePivotsWithIntersection` (`RPI`) algorithm of [16,15], (ii) the `LowerUnits` (`LU`) algorithm of [16], (iii) a structural hashing based approach (`SH`) similar to that of [14], (iv) the local rewriting rules of [17,18,23]. Some manipulation routines are available depending on the LIS chosen: for example, in case of McMillan's LIS M it is possible to perform a fast transformation of the refutation to achieve a partial CNFization of the interpolant [6,18]. The local rewriting rules can also be applied to further strengthen or weaken the interpolant with respect to a given LIS [6]. PeRIPLO does not implement techniques to directly minimize the interpolants after their generation (as, e.g., in [7]); nevertheless, structural hashing is performed while building formulae, for a more efficient representation in memory.

4 Function Summaries in Bounded Model Checking

SAT-based BMC is one of the most successful approaches to software verification. It checks a program w.r.t. a property by 1) unwinding loops and recursive function calls up to a given bound, 2) encoding program and negated property into a propositional *BMC formula*, and 3) using a SAT-solver to check the BMC formula. If the formula is unsatisfiable, the program is safe w.r.t. the bound; otherwise, a satisfying assignment identifies a behavior that violates the property.

We describe in the following two BMC applications which employ interpolation-based *function summaries* as over-approximations of function calls. These applications, respectively implemented in the FunFrog [24] and eVolCheck [25] tools, prove particularly suitable to PeRIPLO, due to the impact size and strength of interpolants can have on verification.

FunFrog. In [9], Sery et al. present a framework to perform incremental verification of a set of properties. Summaries are used to store information about the already analyzed portions of the program, which helps to check subsequent properties more efficiently.

A summary I_f for a function f is an interpolant constructed from an unsatisfiable BMC formula $\phi \equiv A_f \wedge B_\pi$, where A_f encodes f and its nested calls, B_π the rest of the program and the negated property π (which holds for the program). While checking the program w.r.t. another property π', the BMC formula changes to $A_f \wedge B_{\pi'}$; I_f is used in place of f: if $I_f \wedge B_{\pi'}$ turns out to be unsatisfiable, then the summary is still valid and π' is proved to hold in the program. If instead $I_f \wedge B_{\pi'}$ is satisfiable, satisfiability could be caused by the overapproximation due to I_f: I_f is replaced by the precise encoding of f and the check is repeated. If $A_f \wedge B_{\pi'}$ is satisfiable, the error is real; if $A_f \wedge B_{\pi'}$ is unsatisfiable, then the error is spurious and I_f is *refined* to a new I'_f.

The ability to reuse summaries depends on their quality. According to our intuition, *accurate summaries* (i.e. *strong interpolants*) are effective in FunFrog: a summary in fact over-approximates the behavior of a function call w.r.t. an assertion; the more precise the summary is, the more closely it reflects the behavior of the corresponding function and the more likely it is to be employed in the verification of subsequent assertions.

eVolCheck. The upgrade checking algorithm of [10] uses function summarization for BMC in a different way. Verification is done simultaneously w.r.t. a fixed set of properties, but for a program that undergoes modifications. Summaries $\{I_i\}$ are computed for the function calls $\{A_{f_i}\}$ of the original version of the program, and applied to perform local incremental checks of the new version. If the old summaries are general enough to over-approximate the new behavior of the modified functions $\{A_{f'_j}\}$ (i.e. $A_{f'_j} \to I_j$) then the new version is safe. Otherwise, the summaries of the caller functions of the $\{A_{f'_j}\}$ are checked in the same way. If the check succeeds, new summaries $\{I'_j\}$ are generated that *refine* the old $\{I_j\}$. This process continues up to the call tree root. If in the end the summary of the main function is proven invalid, then the new version is buggy.

In contrast with FunFrog, *coarse summaries* (i.e. *weak interpolants*) are more suitable for eVolCheck; the underlying intuition is that weaker interpolants represent abstractions which are more "tolerant" and are more likely to remain valid when the functions are updated.

Compact summaries are expected to yield a more efficient verification both in the FunFrog and eVolCheck frameworks: on one hand, storing and reusing smaller formulae is less expensive, on the other hand, summary reduction via proof compression allows to remove redundancies while keeping the relevant information; in the FunFrog approach, additionally, new summaries are built when possible from refutations involving previously computed summaries.

5 Experimental Evaluation

We evaluated FunFrog and eVolCheck on a collection of 50 crafted C benchmarks characterized by a non trivial call tree structure reflecting the structure of real C programs used in previous experimentation [24,25]. The benchmarks contain assertions distributed on different levels of the tree, which makes them particularly suitable for summary-based verification. FunFrog and eVolCheck employ PeRIPLO for symbolic reasoning and interpolation; they provide as input BMC formulae and receive as output interpolants, specifying a LIS depending on the desired interpolant strength. Proof compression techniques can also be applied in order to produce smaller summaries. The experiments were carried out on a 64-bit Ubuntu server featuring a Quad-Core 4GHz Xeon CPU, with a memory threshold of 13GB[2].

[2] Tools and data are available at `http://verify.inf.unisi.ch/files/LPAR.tar.gz`

FunFrog. In a first phase, FunFrog was run to check the assertions of each benchmark incrementally w.r.t. the call tree, with the goal of maximizing the reuse of summaries. Consider a program with the following chain of nested calls:

$$main()\{\, f()\{\, g()\{\, h()\{\} \, Assert_g\} \, Assert_f\} \, Assert_{main}\}$$

where $Assert_x$ denotes an assertion in the body of a function x. In a successful scenario, (i) $Assert_g$ is checked and a summary I_h for h is created; (ii) $Assert_f$ is efficiently verified by exploiting I_h (I_g is then built over I_h) and (iii) so is $Assert_{main}$ by means of I_g. Each benchmark was tested in different configurations: with/without performing proof compression before interpolation and choosing one among M, P, M' to compute all the interpolants. Compression consisted of a sequential run of `LU,SH,RPI` (see §3); this particular combination is effective in reducing proofs, as shown in [18].

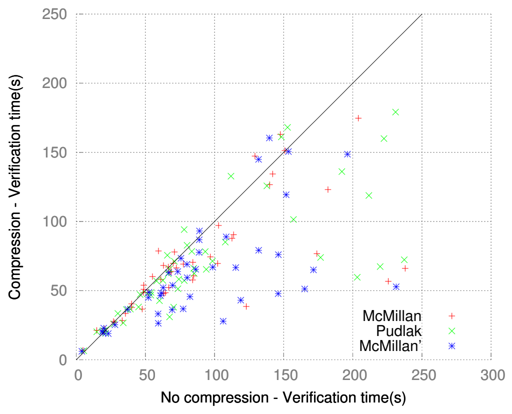
eVolCheck. In a second phase, new versions of the benchmarks were created, modifying syntax/semantics of the original programs. First eVolCheck was run to check all assertions at once, yielding a collection of function summaries; then the new program versions were verified w.r.t. the same assertions by using the summaries. As discussed in [10], while performing upgrade checking the interpolants need to satisfy a property known as *tree interpolation.* In [22] it is proved that tree interpolation is satisfied by M, P but not by M'; for this reason we only took into account M and P for experimentation. Compression was performed as in FunFrog.

Experimental Results. Small interpolants indeed have a strong impact on the performance in both frameworks. Figure 2 compares the verification times for the benchmarks in FunFrog (a) and eVolCheck (b), with and without performing proof compression before interpolation. Table 1 provides additional statistics for the individual interpolation systems: *#Refinements* denotes the total amount of summary refinements in FunFrog, while *#Invalid summaries* the total number of summaries that in eVolCheck were made invalid because of program updates; *Avg*$|I|$ and *Time(s)* indicate the average size of interpolants and the average verification time over all the benchmarks; $Time_C\,/\,Time_V$ *ratio* is the ratio between the time spent for proof compression and the verification time.

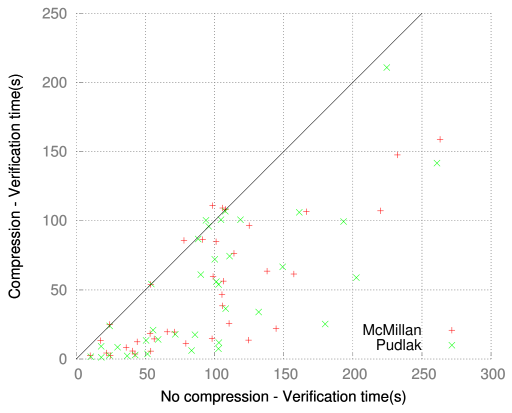
Figure 2 and Table 1 show the remarkable performance improvement achieved by exploiting proof compression; FunFrog, e.g., obtains a reduction in the average interpolants size Avg$|I|$ up to 95% and a speedup up to 54%. Note also in Figure 2 that the effect of compression increases with the complexity of the benchmark; the overhead due to applying compression techniques becomes in fact less and less significant as the benchmark verification time grows.

According to the intuitions discussed in §4, *strong interpolants* prove beneficial in FunFrog, while *weak interpolants* are more suitable for eVolCheck; this is represented in Table 1 by a smaller amount of summary refinements in FunFrog and of invalidated summaries in eVolCheck.

The results show that the size of interpolants seems to have definitely an overall greater impact than interpolant strength. Verification time, in fact, is



(a) FunFrog



(b) eVolCheck

Fig. 2. Proof compression effect on verification time.

principally determined by the size of the summaries, so that, even in presence of a larger amount of refinements or invalidated summaries, smaller summaries tend to lead to a better performance.

It is important to remark that both size and strength are dependent on the features of the refutations from which the interpolants are produced, as well as on the specific interpolation algorithms, and that these aspects cannot be considered separately. For example, in our experimentation we found considerable differences in the size of the interpolants generated by the three LISs, and in the effect of proof compression: interpolants generated with M' in FunFrog were on average twice as big as those generated with M, but they benefited the most from compression.

Moreover, among all existing refutations for a certain unsatisfiable formula (including those obtained via compression), there might be some which are of better "quality" w.r.t. interpolation by means of LISs. A good refutation could be characterized by a large logical "distance" between the interpolant I yielded

Table 1. Verification statistics for FunFrog and eVolCheck.

(a) FunFrog

No Compression	M	P	M'
#Refinements	290	298	308
Avg $\|I\|$	38886.62	39372.07	72994.08
Time(s)	4568.08	4929.93	6805.81

Compression	M	P	M'
#Refinements	293	293	294
Avg $\|I\|$	4336.21	3402.58	3255.69
Time(s)	3327.56	3450.17	3201.72
$\text{Time}_C/\text{Time}_V$ ratio	0.32	0.33	0.32

(b) eVolCheck

No Compression	M	P
#Invalid summaries	65	63
Avg $\|I\|$	334554.64	377903.11
Time(s)	4322.57	4402.00

Compression	M	P
#Invalid summaries	63	62
Avg $\|I\|$	12579.89	12929.82
Time(s)	2073.79	2057.34
$\text{Time}_C/\text{Time}_V$ ratio	0.19	0.19

by M and I' yielded by M', where the distance between I and I' — remember that $I \rightarrow I'$ — is defined as the number of models of I' that are not models of I. A large distance in this sense would allow for a higher degree of variation in the coarseness of summaries, with direct impact on verification.

6 Conclusions

Craig interpolation is a standard means for abstraction in symbolic model checking, but it is still not clear what makes interpolants good in a particular verification framework. We addressed the problem of generating effective interpolants by evaluating the impact of size and logical strength in the context of software SAT-based BMC. To this end, we introduced PeRIPLO, a novel framework that drives interpolation by providing routines for manipulation of the resolution refutations from which the interpolants are computed and for systematic variation of the interpolants strength. As case studies we considered two BMC applications which use interpolation to generate function summaries: (i) verification of a C program incrementally with respect to a number of different properties, and (ii) incremental verification of different versions of a C program with respect to the fixed set of properties. We provided solid experimental evidence that compact interpolants improve the verification performance in the two applications. We also carried out a first systematic evaluation of the impact of strength in a specific verification domain, showing that different applications benefit from interpolants of different strength: specifically, stronger and weaker interpolants are respectively desirable in (i) and (ii).

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: TACAS. (1999) 193–207
2. Craig, W.: Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. Journal of Symbolic Logic **22**(3) (1957) 269–285
3. McMillan, K.: Interpolation and SAT-Based Model Checking. In: CAV. (2003)
4. Pudlák, P.: Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. Journal of Symbolic Logic **62**(3) (1997) 981–998
5. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant Strength. In: VMCAI. (2010) 129–145
6. Jhala, R., McMillan, K.: Interpolant-Based Transition Relation Approximation. In: CAV. (2005) 39–51
7. Cabodi, G., Lolacono, C., Vendraminetto, D.: Optimization Techniques for Craig Interpolant Compaction in Unbounded Model Checking. In: DATE 13. (2013)
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Volume 58 of Advances in Computers. Elsevier (2003) 117–148
9. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based Function Summaries in Bounded Model Checking. In: HVC. (2011) 160–175
10. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental Upgrade Checking by Means of Interpolation-based Function Summaries. In: FMCAD. (2012)
11. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from Proofs. In: POPL. (2004) 232–244
12. McMillan, K.: Lazy Abstraction with Interpolants. In: CAV. (2006) 123–136
13. McMillan, K.: An Interpolating Theorem Prover. In: TACAS. (2004) 16–30
14. Cotton, S.: Two Techniques for Minimizing Resolution Proofs. In: SAT. (2010)
15. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-Time Reductions of Resolution Proofs. In: HVC. (2008) 114–128
16. Fontaine, P., Merz, S., Paleo, B.W.: Compression of Propositional Resolution Proofs via Partial Regularization. In: CADE. (2011) 237–251
17. Rollini, S.F., Bruttomesso, R., Sharygina, N.: An Efficient and Flexible Approach to Resolution Proof Reduction. In: HVC. (2010) 182–196
18. Rollini, S., Bruttomesso, R., Sharygina, N., Tsitovich, A.: Resolution Proof Transformation for Compression and Interpolation. In: http://arxiv.org/abs/1307.2028
19. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.: Robust Boolean reasoning for Equivalence Checking and Functional Property Verification. IEEE Transactions on CAD **21**(12) (2002) 1377–1394
20. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: SAT. (2004) 502–518
21. Rollini, S., Sery, O., Sharygina, N.: Leveraging Interpolant Strength in Model Checking. In: CAV. (2012) 193–209
22. Gurfinkel, A., Rollini, S., N.Sharygina: Interpolation Properties and SAT-Based Model Checking. In: ATVA. (2013)
23. Bruttomesso, R., Rollini, S., Sharygina, N., Tsitovich, A.: Flexible Interpolation with Local Proof Transformations. In: ICCAD. (2010) 770–777
24. Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: Bounded Model Checking with Interpolation-based Function Summarization. In: ATVA. (2012) 203–207
25. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental Upgrade Checker for C. In: TACAS. (2013) 292–307