FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization*

Ondrej Sery[1,2], Grigory Fedyukovich[1], 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. This paper presents FunFrog, a tool that implements a function summarization approach for software bounded model checking. It uses interpolation-based function summaries as over-approximation of function calls. In every successful verification run, FunFrog generates function summaries of the analyzed program functions and reuses them to reduce the complexity of the successive verification. To prevent reporting spurious errors, the tool incorporates a counterexample-guided refinement loop. Experimental evaluation demonstrates competitiveness of FunFrog with respect to state-of-the-art software model checkers.

1 Introduction

Bounded model checkers (BMC) [1] search for errors in a program within the given bound on the maximal number of loop iterations and recursion depth. Typically, the check is repeated for different properties to be verified and thus large amount of the work is repeated. This raises a problem of constructing an incremental model checker. In this paper, we present a tool, FunFrog, that serves this goal. From a successful verification run, FunFrog extracts function summaries using Craig interpolation [3]. The summaries are then used to represent the functions in subsequent verification runs, when the same code is analyzed again (e.g., with respect to different properties). Significant time savings can be achieved by reusing summaries between the verification runs.

To be able to use interpolation for function summarization, FunFrog converts the unwound program into a partitioned bounded model checking (PBMC) formula. For each function to be summarized, this formula is partitioned into two parts. The first part symbolically encodes the function itself and all its callee functions. The second part encodes the remaining functions, i.e., the calling context of the function. Given the two parts, a Craig interpolant that constitutes the function summary is then computed. Our function summaries are over-approximations of the actual behavior of the functions. As a result, spurious errors may occur due to a too coarse over-approximation. To discard spurious errors, FunFrog implements a counterexample-guided refinement loop.

The paper provides an architectural description of the tool implementing the function summarization approach to bounded model checking and discusses the tool usage and experimentation on various benchmarks[1].

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

[1] Further details on interpolation-based function summarization can be found in [4].

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 203–207, 2012.
© Springer-Verlag Berlin Heidelberg 2012

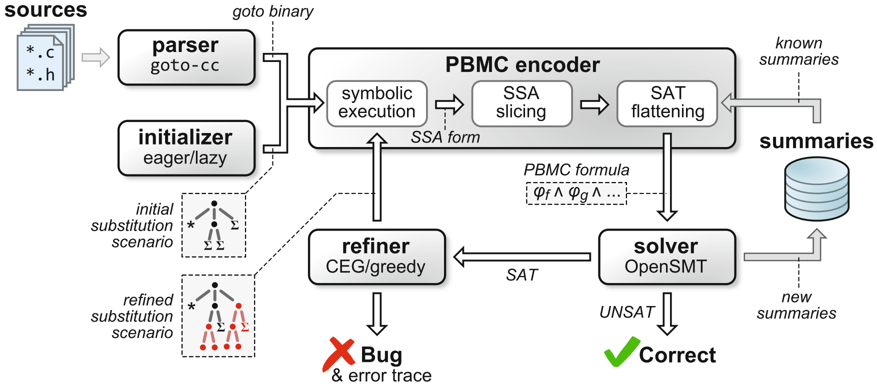

Fig. 1. FunFrog architecture overview

2 Tool Architecture

The architecture of FunFrog is depicted in Fig. 1. The tool takes a C program and uses the *parser* for pre-processing. The parser produces an intermediate code representation, which is then used for encoding into a PBMC formula by *PBMC encoder*. Encoding is achieved using *symbolic execution*, which unwinds the program and prepares its static single assignment (SSA) form, *SSA slicing* that removes the SSA steps irrelevant to the property, and *SAT flattening* that produces the final formula by encoding it into propositional logic. FunFrog loads function summaries from a persistent storage and attempts to use them during encoding as over-approximations of the corresponding program functions. The tool passes the resulting formula to a *solver*. If the formula is unsatisfiable, the program is safe and FunFrog uses interpolation to generate new function summaries and stores them for use in later runs. In case of a satisfiable formula, FunFrog asks *refiner* whether a refinement is necessary. If so, FunFrog continues by precisely encoding the functions identified by the refiner. If a refinement is not necessary (i.e., no summarized function call influences the property along the counterexample), the counterexample is real, and the program is proven unsafe. In the following, we describe each step of FunFrog in more detail.

Parsing. As the first step, the source codes are parsed and transformed into a *goto-program*, where the complicated conditional statements and loops are simplified using only guards and goto statements. For this purpose, FunFrog uses `goto-cc`[2] , i.e., a parser specifically designed to produce intermediate representation suitable for formal verification. Other tools from the CProver[2] framework can be used to alter this representation. For example, `goto-instrument` injects additional assertions (e.g., array bound tests) to be checked during analysis.

Symbolic execution. In order to unwind the program, the intermediate representation is symbolically executed tracking the number of iterations of loops. The result of this step is the SSA form of the unwound program, i.e., a form where every variable is assigned at most once. This is achieved by adding version numbers to the variables. In FunFrog,

[2] `http://www.cprover.org/`

this step is also influenced by the choice of an *initial substitution scenario*. Intuitively, it defines how different functions should be encoded (e.g., using precise encoding or using a summary).

Slicing. After the symbolic execution step, slicing is performed on the resulting SSA form. It uses dependency analysis in order to figure out which variables and instructions are relevant for the property being analyzed. The dependency analysis also takes summaries into account. Whenever an output variable of a function is not constrained by a function summary, its dependencies need not be propagated and a more aggressive slicing is achieved.

SAT flattening. When the SSA form is pruned, the PBMC formula is created by flattening into propositional logic. The choice of using SAT allows for bit-precise reasoning. However, in principle, the SAT flattening step could be substituted by encoding into a suitable SMT theory that supports interpolation.

Solving. The PBMC formula is passed to a SAT solver to decide its satisfiability. FunFrog uses OpenSMT [2] in the SAT solver mode for both satisfiability checks and as an interpolating engine. Certain performance penalties follow from the additional bookkeeping in order to produce a proof of unsatisfiability used for interpolation.

Summaries extraction. When the PBMC formula is unsatisfiable, FunFrog extracts function summaries using interpolation using the proof of unsatisfiability. The extracted summaries are serialized in a persistent storage so that they are available for other FunFrog runs. In this step, FunFrog also compares the new summaries with any existing summaries for the same function and the same bound, and keeps the more precise (tighter over-approximation) one.

Refiner. The refiner is used to identify and to mark summaries directly involved in the error trace. We call this strategy CEG (counterexample-guided). Alternatively, the refiner can avoid identification of summaries in the error trace and can mark all summaries for refinement (greedy strategy). In other words, greedy strategy falls back to the standard BMC, when the summaries are not strong enough to prove the property.

3 Tool Usage

When running FunFrog, the user can choose the preferred initial substitution scenario, a refinement strategy and whether summaries optimization and slicing should be performed. The user can also specify the unwinding bound; the overall bound as well as bounds for particular loops. The input code is expected to contain user provided assertions to be checked for violations. The user can choose which assertion(s) should be checked by FunFrog. Linux binaries of FunFrog as well as the benchmarks used for evaluation are available online for other researchers[3]. The webpage also contains a tutorial explaining how to use FunFrog and explanation of the most important parameters.

Experiments. In order to evaluate FunFrog, we compared it with other state-of-the-art C-model checkers CBMC (v4.0), SATABS (v3.0 with Cadence SMV v10-11-02p46),

[3] `www.verify.inf.usi.ch/funfrog`

Table 1. Verification times [s] of FunFrog, CBMC, SATABS, and CPAchecker, where '∞' is a timeout (1h), '×' - bug in safe code, '†' - other failure (We notified the tool authors about the issues), number of lines of code, preprocessed code instructions in `goto-cc`, function calls, and assertions.

benchmark					FunFrog details						total			
	#LoC	#Instructions	#func. calls	#assertions	#ref. iter.	symb. ex.	slicing	flattening	solving	interpol.	FunFrog	CBMC	SATABS	CPAchecker
`floppy`	10288	2164	227	8	0	4.80	0.07	6.05	2.94	0.57	14.54	19.59	918.25	383.97
`kbfiltr`	12247	1052	64	8	0	1.72	0.01	2.38	0.23	0.15	4.60	5.33	91.37	†
`diskperf`	6324	2037	182	5	0	2.41	0.01	2.31	0.42	0.36	5.60	21.42	146.82	259.26
`no_sprintf`	178	68	6	2	0	0.01	0.00	0.03	0.03	0.01	0.08	0.01	125.69	2.96
`gd_simp`	207	82	4	5	0	0.03	0.00	0.07	0.05	0.01	0.17	0.03	∞	×
`do_loop`	126	176	12	7	3	7.74	2.66	2.58	2.29	0.11	15.78	19.52	∞	×
`goldbach`	268	344	22	6	0	0.41	0.00	1.53	2.03	0.78	5.78	15.44	∞	†

and CPAchecker (v1.1). CBMC and FunFrog are BMC tools, provided with the same bound. We evaluated all tools (with default options) on both real-life industrial benchmarks (including Windows device drivers) and on smaller crafted examples designed to stress-test the implementation of our tool and verified them for user defined assertions (separate run for each assertion). The assertions held, so FunFrog had the opportunity to extract and reuse function summaries.

Table 1 reports the running times of all the tools[4]. In case of FunFrog, the summaries were generated after the first run (for the first assertion in each group) and reused in the consecutive runs (for the rest of (#asserts - 1) assertions). To demonstrate the performance of FunFrog, the running times of different phases of its algorithm were summed across all runs for the same benchmark. Note that the time spent in counterexample analysis (i.e., the only computation, needed for refinement) is negligible, and thus not reported in a separate column, but still included to the total.

As expected, FunFrog was outperformed by CBMC on the smaller examples without many function calls, but FunFrog's running times were still very competitive. On majority of the larger benchmarks, FunFrog outperformed all the other tools. These benchmarks feature large number of function calls so FunFrog benefited from function summarization.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

[4] The complete set of experiments can be found at `www.verify.inf.usi.ch/funfrog`.

2. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
3. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. of Symbolic Logic, 269–285 (1957)
4. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based Function Summaries in Bounded Model Checking. In: HVC 2011. LNCS (2011) (to appear)