HSF(C): A Software Verifier Based on Horn Clauses
(Competition Contribution)

Sergey Grebenshchikov[1], Ashutosh Gupta[2], Nuno P. Lopes[3], Corneliu Popeea[1], and Andrey Rybalchenko[1]

[1] Technische Universität München
[2] IST Austria
[3] INESC-ID / IST - TU Lisbon

Abstract. HSF(C) is a tool that automates verification of safety and liveness properties for C programs. This paper describes the verification approach taken by HSF(C) and provides instructions on how to install and use the tool.

1 Verification Approach

HSF(C) is a tool for verification of C programs based on predicate abstraction and refinement following the counterexample-guided abstraction refinement (CEGAR) paradigm [4]. There are a number of successful tools [1,7,5,10,2] based on abstraction refinement. We give here a brief description of our verification algorithm; interested readers can find more details about the underlying theory behind our implementation in [10,6].

The algorithm used in HSF(C) is a generalization of the CEGAR scheme that deals with Horn-like clauses instead of transition systems/programs with procedures. We use Horn clauses to represent both the program to be verified and the proof rule used for verification, i.e., safety checking for programs with procedures. The proof rule lists premises for program safety and requires auxiliary assertions that represent inductive invariants. Given Horn clauses as input, our algorithm proceeds in three steps.

1. With a fixed set of predicates, initially empty, we find a solution for the auxiliary assertions. At this step we perform logical inference and rely on abstraction to ensure termination in the presence of recursion and to ensure efficiency in the presence of large sets of clauses.
2. We check whether the computed solution satisfies program safety. If so, the verification succeeds and the algorithm returns "safe".
3. We check whether the logical inference performed in the first step in the setting without any abstraction yields a solution that still violates program safety. If the violation is still present then we return "unsafe" and the inference tree as an error path that reaches the error location. Otherwise, we use the obtained solution to refine the abstraction function and go back to the first step.

C. Flanagan and B. König (Eds.): TACAS 2012, LNCS 7214, pp. 549–551, 2012.

2 Software Architecture

HSF(C) relies on the CIL library [9] as a frontend for our tool. An additional frontend step transforms the CIL abstract syntax tree representation in Horn clauses. The Horn clauses are generated automatically from a proof rule for safety checking with support for procedure summarization [11]. These Horn clauses are then solved with the CEGAR algorithm described in the previous section. Our solver is implemented in Prolog and compiled using the SICStus Prolog compiler [12]. Our implementation relies on the constraint solver for linear arithmetic CLP(Q) [8].

3 Discussion

In this section, we present our experience running HSF(C) on the benchmarks from the software competition.

ControlFlowInteger. We found the approach based on abstraction and refinement well suited for the benchmarks in the category. The proofs found by HSF(C) typically involve a small number of predicates. For handling the few pointers and heap-allocated structures that are used by these benchmarks, we found the use of the pointer analysis provided by CIL to be sufficiently precise.

SystemC. The benchmarks from this category encode concurrency from the program in finite-domain auxiliary variables. Some heuristics proposed in [3] combine explicit-state model checking to model the states of the SystemC scheduler with predicate abstraction. Such heuristics would have been more appropriate to handle these benchmarks than our current approach based only on predicate abstraction.

Other Categories. HSF(C) did not participate in the other four competition categories, i.e., Concurrency, DeviceDrivers, DeviceDrivers64, and HeapManipulation. Here we list the current limitations of our tool that we need to address to be able to handle these benchmarks:

- Concurrency: the frontend of HSF(C) needs a model for the functions from the Pthreads library.
- DeviceDrivers, DeviceDrivers64: some of the features of the C language that HSF(C) does not precisely model are fixed-size integers, union types, volatile variables, and bitwise operations.
- HeapManipulation: the pointer analysis that we use is not precise to handle the data structures from these benchmarks.

To summarize, we ran our tool on 158 benchmarks from two categories. HSF(C) obtained the following results:

- ControlFlowInteger (140/144 points): for all benchmarks where HSF(C) reported a result, the result was correct. HSF(C) ran out of time for two benchmarks.
- SystemC (8/87 points): for all benchmarks where HSF(C) reported a result, the result was correct. HSF(C) ran out of time or memory on 57 benchmarks.

4 Tool Setup

HSF(C) can be downloaded from the following webpage:

`http://www7.in.tum.de/tools/hsf/`

The HSF(C) distribution consists of three statically compiled binaries that correspond to the C frontend, a converter to Horn clauses, and the Horn clause solver. The distribution also contains a script that runs the three executables with appropriate parameters. The tool should be run as follows: `./qarmc.sh <file.c>`. The working directory (`PWD`) must be the directory where the HSF(C)'s files are located. The only required library is the standard glibc 32-bit.

References

1. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL (2002)
2. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
3. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: A software model checking approach. In: FMCAD, pp. 51–59 (2010)
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
5. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
6. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL, pp. 331–344 (2011)
7. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)
8. Holzbaur, C.: OFAI clp(q,r) Manual, Edition 1.3.3. Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09 (1995)
9. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
10. Podelski, A., Rybalchenko, A.: ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
11. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL, pp. 49–61 (1995)
12. The Intelligent Systems Laboratory. SICStus Prolog User's Manual. Swedish Institute of Computer Science, Release 4.2.0 (2011)