WOLVERINE: Battling Bugs with Interpolants*
(Competition Contribution)

Georg Weissenbacher[1,**], Daniel Kroening[2], and Sharad Malik[1]

[1] Department of Electrical Engineering, Princeton University
[2] Department of Computer Science, Oxford University
(georg.weissenbacher@magd.oxon.org)

Abstract. WOLVERINE is a software verifier that checks safety properties of sequential ANSI-C and C++ programs, deploying Craig interpolation to derive program invariants. We describe the underlying approach and the architecture, and provide instructions for installation and usage.

1 Approach

WOLVERINE [1] is a software verification tool for ANSI-C and C++ programs that aims at finding either a Hoare-style correctness proof or a counterexample for a given reachability property. The tool is an implementation of the interpolation-based lazy abstraction algorithm [2] outlined in Figure 1. A description of the steps ① to ⑤ of Figure 1 is provided below.

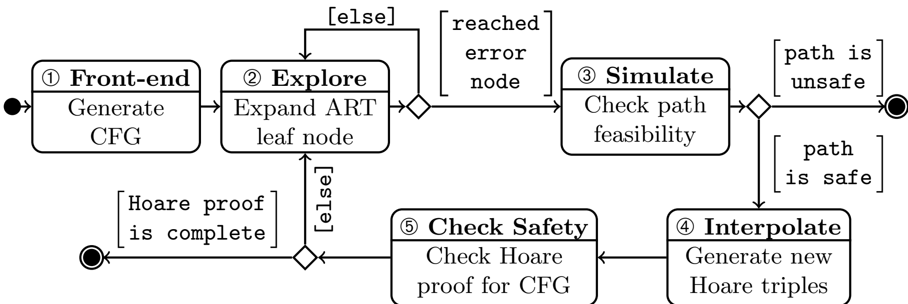
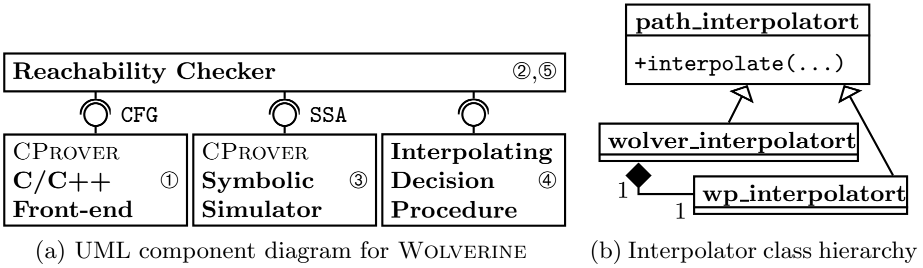


Fig. 1. UML activity diagram describing the work-flow of WOLVERINE

① WOLVERINE generates a control flow graph (CFG) representation of the program and encodes reachability properties using assertions/error nodes.
② Following the lazy abstraction paradigm established by [3], WOLVERINE constructs an abstract reachability tree (ART). To this end, it explores the paths of the CFG (in a depth first search manner) until it encounters an assertion.[1]

* Supported by a gift from the Intel Labs Academic Research Office.
** Corresponding author.
[1] The search algorithm of WOLVERINE 0.5c incorporates a constant propagation domain in order to enable early pruning of infeasible execution traces.

C. Flanagan and B. König (Eds.): TACAS 2012, LNCS 7214, pp. 556–558, 2012.
© Springer-Verlag Berlin Heidelberg 2012



(a) UML component diagram for Wolverine (b) Interpolator class hierarchy

Fig. 2. Software architecture of Wolverine

③ Given a path that reaches an assertion, Wolverine deploys symbolic simulation to determine whether the path corresponds to a feasible program execution violating the assertion; such unsafe executions are reported.

④ If the path is safe, Wolverine uses Craig interpolation to generate Hoare triples that prove that the assertion holds (c.f. [4]) and updates the edges and nodes of the ART accordingly: the spurious counterexample serves as a catalyst for refining the current approximation of safely reachable states [5].

⑤ If the Hoare triples of the ART are sufficient to prove the safety of all paths of the CFG, Wolverine concludes that the program is correct. Otherwise, the tool continues to expand paths that are not yet covered (step ②).

2 Software Architecture

Figure 2a shows the components and architecture of Wolverine. Our implementation uses the front-end (①) and the symbolic simulator (③) of the CProver framework (`http://www.cprover.org`). Wolverine uses an interpolating decision procedure (④) to extract Hoare triples from infeasible paths. To this end, the tool deploys its built-in decision procedure for equality logic with uninterpreted functions and limited support for bit-vectors [6,7,8] and falls back on the weakest precondition should this interpolator fail (see Figure 2b).

3 Tool Setup and Usage

Installation. Binaries for Linux, Windows, and MacOS X can be downloaded from the project website (`http://www.cprover.org/wolverine`) and should be deployed in a directory listed in the `PATH` environment variable. Wolverine requires a pre-processor (`cl.exe`, which is part of Visual Studio Express, on Windows and GNU's `gcc` on Unix-based platforms) and the header files typically packaged with it to be installed.

Usage. Wolverine must be executed from within the Visual Studio command prompt on Windows or a terminal on Linux and Mac OS X, and accepts options and source file names of the program to be verified as operands. By default, Wolverine scans the program for assertions and checks whether they hold. If executed with the option `--error-label ERROR`, Wolverine checks whether the label `ERROR` is reachable.

By default, Wolverine assumes that the host platform and the target platform for the program under test coincide. Therefore, in order to verify a Windows device driver on a Linux host, the options `--no-library --i386-win32` are recommended (but were not applied in the competition). The target processor architecture of the program under test has to be specified using the options `--32` or `--64` where it differs from the host. In the competition, these options were applied accordingly to all benchmarks.

4 Strengths and Limitations

While Wolverine shares many of the characteristics of predicate abstraction-based verifiers (most prominently, Slam [9]), it avoids the computationally expensive image computation required to construct the abstraction (c.f. [2]), enabling the rapid detection of counterexamples (discussed in [1]).

Wolverine's performance is contingent on the Hoare triples that the interpolating decision procedure derives from spurious counterexamples. The inherent properties of interpolants typically enable concise abstractions. A "diverging" sequence of predicates, however, can result in a failed verification attempt. The built-in interpolator of Wolverine version 0.5c provides no support for linear arithmetic, quantified invariants, and heap models. In the competition, this led to a sub-optimal performance of Wolverine for benchmarks containing arithmetic expressions, unbounded arrays, or dynamic data structures. Moreover, Wolverine does currently not support the verification of concurrent programs.

References

1. Kroening, D., Weissenbacher, G.: Interpolation-Based Software Verification with Wolverine. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 573–578. Springer, Heidelberg (2011)
2. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
3. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70. ACM (2002)
4. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM (2004)
5. 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)
6. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant Strength. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer, Heidelberg (2010)
7. Kroening, D., Weissenbacher, G.: Lifting propositional interpolants to the word-level. In: FMCAD, pp. 85–89. IEEE (2007)
8. Kroening, D., Weissenbacher, G.: An Interpolating Decision Procedure for Transitive Relations with Uninterpreted Functions. In: Namjoshi, K., Zeller, A., Ziv, A. (eds.) HVC 2009. LNCS, vol. 6405, pp. 150–168. Springer, Heidelberg (2011)
9. Ball, T., Rajamani, S.K.: The slam project: Debugging system software via static analysis. In: POPL, pp. 1–3. ACM (2002)