Interpolation-based Software Verification with WOLVERINE[⋆]

Daniel Kroening[1] and Georg Weissenbacher[2]

[1] Computer Science Department, Oxford University
[2] Department of Electrical Engineering, Princeton University

Abstract. WOLVERINE is a software verification tool using Craig interpolation to compute invariants of ANSI-C and C++ programs. The tool is an implementation of the *lazy abstraction* approach, generating a reachability tree by unwinding the transition relation of the input program and annotating its nodes with interpolants representing safe states. WOLVERINE features a built-in interpolating decision procedure for equality logic with uninterpreted functions which provides limited support for bit-vector operations. In addition, it provides an API enabling the integration of other interpolating decision procedures, making it a valuable source of benchmarks and allowing it to take advantage of the continuous performance improvements of SMT solvers. We evaluate the performance of WOLVERINE by comparing it to the predicate abstraction-based verifier SATABS on a number of verification conditions of Linux device drivers.

1 Introduction

The last decade has seen significant progress in the area of automated software verification, manifesting itself in a number of impressive verification tools. A recent and comprehensive survey of software verification techniques is provided in [1] and a comparison of verification tools can be found in [2]. One approach that received particular attention is predicate abstraction [3], a technique that constructs a conservative abstraction of the original program using a finite set of first-order-logic predicates to track relevant facts about the program variables.

The performance of such predicate abstraction-based software model checkers is contingent on suitable predicates. Contemporary verification tools (e.g., Microsoft's SLAM [4]) derive these predicates from spurious counterexamples in an iterative manner [5, 6]. Recent incarnations of this technique (such as BLAST [7]) rely on Craig interpolation to derive predicates, taking advantage of the inherent properties of interpolants which enable concise abstractions.

Some flavours of this interpolation-based abstraction mechanism avoid the use of predicate abstraction to construct an abstract transition relation altogether [8–10]. This omission has several advantages:

⋆ Supported by the EU FP7 STREP MOGENTES and a gift from the Intel Labs Academic Research Office.

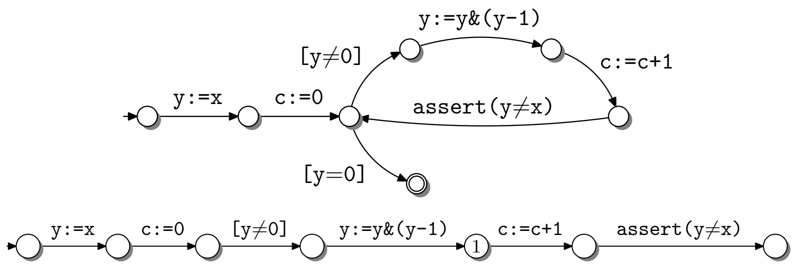


Fig. 1. A program and one of its execution traces

– It eliminates computationally expensive calls to a theorem prover that predicate abstraction-based verifiers require to construct an abstraction, and
– it decreases the size and the complexity of the implementation of the verification tool significantly (by about two thirds in our experience).

The algorithm presented in [8] (and our implementation Wolverine) essentially follows the *lazy abstraction* paradigm [11], but retains the abstract transition function at the coarsest level (determined by the control-flow-graph of the program under scrutiny). The reachability tree obtained by unwinding this transition function is annotated with interpolants representing an over-approximation of the reachable states. A fixed point of these annotations constitutes an invariant establishing the correctness of the program with respect to a given safety property. Section 2 provides more details.

Contributions. Wolverine is a software verification tool checking reachability properties stated in terms of assertions and implements the algorithm described in [8]. It is, to the best of our knowledge, the first freely available[3] software model checker for C/C++ programs based on this algorithm.

Wolverine features a built-in interpolating decision procedure for equality logic with uninterpreted functions which provides limited support for bit-vector operations, while many comparable verification tools use linear arithmetic to approximate semantics of the program. In addition, it provides a programming interface (described in Section 3) enabling the integration of other interpolating decision procedures, allowing it to take advantage of the continuous performance improvements of interpolating SMT solvers (see, for instance, [12–14]).

We present an evaluation of our implementation in Section 4 and provide a tool to generate additional benchmarks on the website of Wolverine.

2 Implementation

The implementation of Wolverine is based on the CProver framework (written in C++), which also forms the foundation of the verification tools CBMC [15],

[3] Source available under a BSD-style license on http://www.cprover.org/wolverine.

SatAbs [16], and Impact [8]. We describe the implementation of Wolverine using the example program in Figure 1. The program implements Wegner's algorithm, determining how many bits of `x` are set to one. The assertion `assert(y≠x)` is a naïve safeguard against non-termination. Wolverine uses symbolic simulation to construct a reachability tree. To this end, it unwinds the control flow graph of the program until an assertion is reached. The shortest execution trace of our example program reaching the assertion is shown at the bottom of Figure 1. In order to check whether the trace violates the assertion, Wolverine transforms it into static single assignment form:

$$\overbrace{(\mathtt{y}_1 = \mathtt{x}_0) \wedge (\mathtt{y}_1 \neq 0) \wedge (\mathtt{y}_2 = \mathtt{y}_1 \& (\mathtt{y}_1 - 1))}^{\text{prefix}} \overset{①}{\wedge} \overbrace{(\mathtt{y}_2 = \mathtt{x}_0)}^{\neg\ \text{assertion}} \qquad (1)$$

Using slicing, we eliminate the assignments to `c`, since they are not relevant to the correctness of the program. The effect of negating the asserted condition is that every satisfying assignment of Formula (1) represents a witness for an assertion violation. Note, however, that the formula is unsatisfiable and therefore this execution cannot violate the assertion. Accordingly, the sub-formula tagged "prefix" in Formula (1) represents a set of reachable states (at location ① in the trace) that is *safe* with respect to the assertion.

Wolverine splits the symbolic representation of the execution trace into n partitions $A_1, \ldots, A_n$, one for each basic block traversed by the trace. It passes these n formulas on to an *interpolating* decision procedure (by default the built-in algorithm described in [17, 18]), which returns $n-1$ interpolants $I_1, \ldots, I_{n-1}$ that satisfy the following conditions [8]:

1. For all $1 \leq j \leq n$, $(I_{j-1} \wedge A_j)$ implies I_j (with $I_0 = \mathsf{true}$ and $I_n = \mathsf{false}$), and
2. for all $1 \leq j < n$, I_j refers only to SSA variables that occur in $A_1, \ldots, A_j$ as well as in $A_{j+1}, \ldots, A_n$.

The first condition guarantees that each interpolant I_j represents a set of safe states at the respective program location from which no state violating the assertion is reachable via the given trace (i.e., the interpolants and the program statements in the trace form Hoare triples). The second condition above guarantees that the interpolants refer only to SSA variables that are *live* at the corresponding location in the trace. For instance, a valid sequence of interpolants for the formulas $(\mathtt{y}_1 = \mathtt{x}_0)$, $(\mathtt{y}_1 \neq 0)$, $(\mathtt{y}_2 = \mathtt{y}_1 \& (\mathtt{y}_1 - 1))$, and $(\mathtt{y}_2 = \mathtt{x}_0)$ would be $\mathtt{y}_1 = \mathtt{x}_0$, $(\mathtt{y}_1 = \mathtt{x}_0) \wedge (\mathtt{y}_1 \neq 0)$, and $(\mathtt{x}_0 \neq 0) \wedge (\mathtt{y}_2 \leq \mathtt{x}_0 - 1)$.

After mapping the SSA variables back into the original program context, Wolverine annotates the corresponding path in the reachability tree accordingly, e.g., the node ① is labelled $(\mathtt{x} \neq 0) \wedge (\mathtt{y} \leq \mathtt{x} - 1)$.

Wolverine continues expanding the reachability tree until each leaf is either fully expanded or *covered* by a previously discovered node. A node is covered if its (or one of its predecessors') annotation implies the annotation of a previously discovered node associated with the same program location.[4] For instance, if

[4] The fact that interpolation is non-monotonic imposes some restrictions on the covering relation, which are described in more detail in [8, 19].

WOLVERINE annotates a node ② (which succeeds ① in the reachability tree and also corresponds to the program location following the assignment `y:=y&(y-1)`) with $(\mathtt{x} \neq 0) \wedge (\mathtt{y} \leq \mathtt{x} - 1)$, then ② is covered by ①.

The built-in decision procedure supports bit-vector operations using a limited set of inference rules (such as $(t_1 = t_2 \,\&\, t_3) \vdash (t_1 \leq t_2)$ and $(1 > t_1) \vdash (t_1 = 0)$ for terms t_i of type unsigned integer). Details are provided in [19, 17]. Moreover, it uses eager bit-blasting and a SAT solver to identify an unsatisfiable core before invoking the "word-level" interpolating decision procedure [19, 18]. If the decision procedure fails to provide an interpolant, WOLVERINE falls back on using the weakest precondition. Finally, using the option `--interpolator smt-out`, WOLVERINE is able to print the SSA instances in the SMT-LIB format, enabling the generation of benchmarks.

3 Interface for Interpolating Decision Procedures

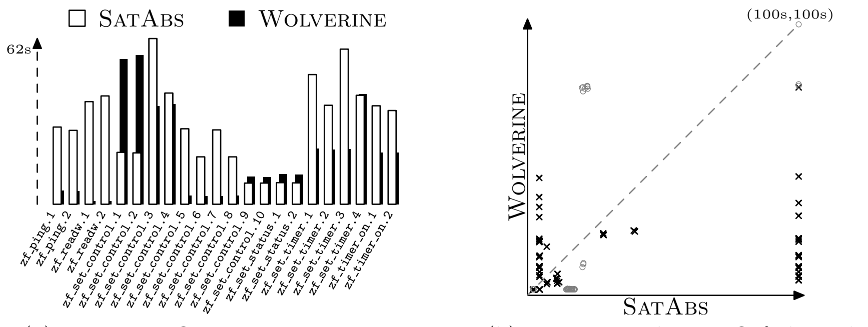
WOLVERINE provides a C++ interface for calling external interpolating decision procedures. In order to integrate an external solver into WOLVERINE, the programmer has to implement a class inheriting from `external_interpolatort`:

```
class external_interpolatort: public wolver_interpolatort {
     ...
     virtual bool initialise();
     virtual bool process_options(const optionst&);
  protected:
     virtual bool translate(const expr_listt&)=0;
     virtual decision_proceduret::resultt solve()=0;
     virtual bool read_interpolants(expr_listt&)=0;       };
```

The public methods `initialise` and `process_options` provide an opportunity to initialise the external tool and to deal with command line parameters. WOLVERINE provides the class `external_processt`, which supports the execution of and communication with command line tools.

The methods `translate` and `read_interpolants` are required to convert formulas between the representation used by the external interpolator and expressions in the CPROVER format. CPROVER expressions (represented by the class `exprt`) are annotated syntax trees with typing information. The class `typet` is used to store types. WOLVERINE expects the interpolants returned to be typed. Accordingly, an interpolator which discards the typing information needs to restore it before returning a result to WOLVERINE. The CPROVER framework provides support for this task in form of the methods `c[pp]_typecheck`.

The method `solve` returns `D_UNSATISFIABLE`, `D_SATISFIABLE`, or `D_ERROR`. In the latter case, WOLVERINE provides the option to fall back on computing interpolants using the weakest precondition. If the instance is satisfiable, the trace represents a valid counterexample and is reported. Otherwise, the method `read_interpolants` is expected to return in its parameter a sequence of typed expressions which satisfy the conditions stated in Section 2.



(a) `machzwd`: IO port assertions (b) `machzwd` and `nbd`: IO & bounds

Fig. 2. Performance results Wolverine vs SatAbs on DDVerify drivers.

4 Checking Linux Device Drivers

Figure 2 provides a comparison of Wolverine with the predicate-abstraction based verifier SatAbs on a number of sequential device driver benchmarks (generated with DDVerify [20], which provides a harness and an OS model annotated with assertions) on a 3GHz Intel Core i7 CPU with 4GB RAM.[5] Figure 2(a) shows the run-time of Wolverine and SatAbs on 22 assertions related to the usage of IO ports for the `machzwd` device driver [20]. Wolverine performs better than SatAbs in all but 6 cases. The performance gain is particularly impressive for the assertions `zf_readw.`[1,2], for which both tools report a counterexample. We attribute this to the lower overhead of the search algorithm of Wolverine. In the case of the claims `zf_set_control.`[1,2], we observe a large number of coverage checks in Wolverine for different branches of the reachability tree, and the *eager* abstraction approach of SatAbs prevails.

The scatter-plot in Figure 2(b) shows the run-time for 73 array bound checks for the `machzwd` driver (displayed using $\times$) and 78 array bound and IO properties for the driver `nbd` (indicated by $\circ$). SatAbs exceeded the time-out of 100 seconds for 23 properties of `machzwd`, and SatAbs as well as Wolverine timed out in 22 cases for `nbd`. Our results suggest that, while SatAbs is significantly faster when few predicates are sufficient to prove an assertion correct, Wolverine's lazy approach is more robust as the number of predicates increases.

5 Conclusion

Wolverine is a freely available implementation of the interpolation-based lazy abstraction algorithm presented in [8]. Its modular design enables the integration of modern interpolating SMT solvers, making it future-proof and (when combined with DDVerify [20]) a valuable source for benchmarks. Our experimental evaluation shows that our implementation is competitive when compared

[5] Performance results for the device drivers presented in [7] are reported in [19].

to existing predicate-abstraction based verification tools. As future work, we intend to integrate and study the performance impact of different interpolation techniques.

References

1. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41** (2009) 21:1–21:54
2. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) **27** (2008) 1165–1178
3. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. Volume 1254 of LNCS. Springer (1997) 72–83
4. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Integrated Formal Verification (IFM). Volume 2999 of LNCS. Springer (2004)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. Volume 1855 of LNCS., Springer (2000) 154–169
6. Ball, T., Rajamani, S.: Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical Report 2002-09, Microsoft Research (2002)
7. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. ACM (2004) 232–244
8. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV. Volume 4144 of LNCS. Springer (2006) 123–136
9. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL, ACM (2010) 471–482
10. Caniart, N.: Merit: An interpolating model-checker. In: CAV. Volume 6174 of LNCS., Springer (2010) 162–166
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. ACM (2002) 58–70
12. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: CAV. Volume 5123 of LNCS., Springer (2008) 304–308
13. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: TACAS. Volume 6015 of LNCS., Springer (2010) 150–153
14. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Transactions on Computational Logic (2010) to appear.
15. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Springer (2004) 168–176
16. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS. Volume 3440 of LNCS. Springer (2005) 570–574
17. Kroening, D., Weissenbacher, G.: An interpolating decision procedure for transitive relations with uninterpreted functions. In: HVC. LNCS, Springer (2011)
18. Kroening, D., Weissenbacher, G.: Lifting propositional interpolants to the word-level. In: FMCAD, IEEE (2007) 85–89
19. Weissenbacher, G.: Program Analysis with Interpolants. PhD thesis, Oxford University (2010)
20. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: ASE, IEEE (2007) 501–504