An SMT-Based Concolic Testing Tool for Logic Programs[⋆]

Sophie Fortz[1], Fred Mesnard[2], Etienne Payet[2], Gilles Perrouin[1], Wim Vanhoof[1], and German Vidal[3]

[1] Université de Namur, Belgique
[2] LIM - Université de la Réunion, France
[3] MiST, VRAIN, Universitat Politècnica de València, Spain

Concolic testing is a well-established validation technique for imperative and object-oriented programs [3,6], but only recently investigated for functional and logic programming languages. Concolic testing for logic programming was initially studied by Mesnard, Payet and Vidal [4], while Giantsos *et al.* [2] and Tikovsky *et al.* [7] considered concolic testing of functional programs.

Concolic testing performs both concrete and symbolic execution in parallel: given a test case (atomic goal), e.g., $p(a)$, we evaluate both $p(a)$ (the *concrete* goal) and $p(X)$ (the *symbolic* goal), where X is a fresh variable, using a concolic execution extension of SLD resolution. Only (nondeterministic) executions of $p(X)$ matching the concrete goal are explored. The generation of alternative test cases relies on a path coverage criterion and is computed through solving *selective unification* problems [4,5]. The previous algorithm [4] does not scale well and does not support negative constraints. By defining selective unification problems as constraints on Herbrand terms and relying on an SMT solver, we address both scalability and completeness issues.

Our concolic scheme extends the previous framework in [4]. Let us illustrate the main ideas with an example. Consider the following logic program:

$$(\ell_1)\ p(a). \qquad (\ell_2)\ p(X) \leftarrow q(X). \qquad (\ell_3)\ q(b).$$

where ℓ_1, ℓ_2, ℓ_3 are (unique) clause labels. With an initial call $p(a)$, the algorithm considers all the possible matching clauses (i.e., all combinations from the heads of the clauses l_1 and l_2): $\{\}, \{p(a)\}, \{p(X)\}, \{p(a), p(X)\}$.

The considered initial call already covers the last case (i.e., it matches with both $p(a)$ and $p(X)$). As for the remaining cases, $\{\}$ (matching no clause) is clearly unfeasible; the case $\{p(a)\}$ (matching with $p(a)$ but not with $p(X)$) is unfeasible as well. The last case $\{p(X)\}$ (matching with $p(X)$ but not with $p(a)$) is feasible with, e.g., $p(b)$. Thus $p(b)$ is our next initial goal. In a second iteration, our algorithm will produce only one test case, $p(c)$, which first matches clause ℓ_1 and, then, the call $q(c)$ fails. Here, we assume that the domain comprises at least one more constant, e.g., c. Such a test case was not generated by the technique

⋆ Third author is a research associate at FNRS that also supports this work (O05518F-RG03). The last author is partially supported by the EU (FEDER) and the Spanish MCI/AEI under grants TIN2016-76843-C4-1-R/PID2019-104735RB-C41 and by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust).

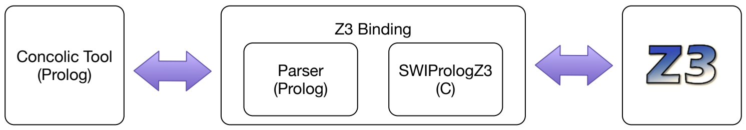

Fig. 1. Implementation workflow.

in [4]. In our tool, this case is produced by solving the following constraint: $p(X) \neq p(a) \wedge (\exists Y\ p(X) = p(Y)) \ \wedge\ q(X) \neq q(b)$, which is solved using an SMT solver. Finally, we consider $p(c)$ as the initial goal and no more alternative test cases are produced, so our algorithm yields three test cases: $p(a)$, $p(b)$ and $p(c)$, achieving a full coverage.

Our prototype is implemented in SWI-Prolog [8] and the Z3 SMT solver [1], as depicted in Figure 1. Since the number of test cases is typically infinite, we consider a maximum term depth to ensure termination. We compared our tool with the previous one, contest [4], on six programs. Regarding execution times, our new tool exhibits a certain overhead on small programs with a low depth due to the calls to the SMT solver. As program size and/or depth increase, the new tool performs up to 10 times faster than contest. We note that the number of test cases generated by the tools are not comparable since our new framework avoids a source of incompleteness (as mentioned in the previous example), but also restricts the number of test cases by forbidding the binding of so-called *output* arguments (which is allowed in contest). More details can be found in the companion paper: http://arxiv.org/abs/2002.07115. The implementation is also publicly available at https://github.com/sfortz/Pl_Concolic_Testing.

References

1. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340. Springer-Verlag, Berlin, Heidelberg (2008)
2. Giantsios, A., Papaspyrou, N., Sagonas, K.: Concolic testing for functional languages. Science of Computer Programming **147**, 109–134 (2017)
3. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proc. of PLDI'05. pp. 213–223. ACM (2005)
4. Mesnard, F., Payet, É., Vidal, G.: Concolic testing in logic programming. TPLP **15**(4-5), 711–725 (2015). https://doi.org/10.1017/S1471068415000332
5. Mesnard, F., Payet, É., Vidal, G.: Selective unification in constraint logic programming. In: Vanhoof, W., Pientka, B. (eds.) PPDP. pp. 115–126. ACM (2017)
6. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/ FSE. pp. 263–272. ACM (2005)
7. Tikovsky, J.R.: Concolic testing of functional logic programs. In: Declarative Programming and Knowledge Management, pp. 169–186. Springer (2017)
8. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. TPLP **12**(1-2), 67–96 (2012). https://doi.org/10.1017/S1471068411000494