Document downloaded from:

http://hdl.handle.net/10251/178919

This paper must be cited as:

Fortz, S.; Mesnard, F.; Payet, E.; Perrouin, G.; Vanhoof, W.; Vidal, G. (2020). An SMT-Based Concolic Testing Tool for Logic Programs. Springer Nature. 215-219. https://doi.org/10.1007/978-3-030-59025-3_13

The final publication is available at

https://doi.org/10.1007/978-3-030-59025-3_13

Copyright Springer Nature

Additional Information

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

Abstract. Concolic testing combines symbolic and concrete execution to generate test cases that achieve a good program coverage. Its benefits have been demonstrated for more than 15 years in the case of imperative programs. In this work, we present a concolic-based test generation tool for logic programs which exploits SMT-solving for constraint resolution.

1 Concolic Testing of Logic Programs

Concolic testing is a well-established validation technique for imperative and object-oriented programs [3,8], but only recently investigated for functional and logic programming languages. Concolic testing for logic programming was initially studied by Vidal [11] and Mesnard *et al.* [4], while Giantsos *et al.* [2] and Tikovsky *et al.* [10] 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. The symbolic goal mimics the steps of the concrete goal but is aimed at gathering constraints that can be later used to produce alternative test cases. In particular, alternative test cases are computed by solving so-called *selective unification* problems [4,6]. The previous algorithm introduced by Mesnard *et al.* [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.

Let us motivate our approach by illustrating one of the problems of the previous framework [4]. Consider the following logic program defining predicates $p/1$ and $q/1$:

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

⋆ 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).

where ℓ_1, ℓ_2, ℓ_3 are (unique) clause labels. Given an initial call, say $p(a)$, the algorithm considers all possible matching clauses for this call (i.e., all combinations from clauses l_1 and l_2) and produces the sets $\{\}$, $\{p(a)\}$, $\{p(X)\}$, and $\{p(a), p(X)\}$ with the heads of the clauses in each combination.

The considered initial call already covers the last case (i.e., it matches both $p(a)$ and $p(X)$). As for the remaining cases:

- *Matching no clause*. This case is clearly unfeasible, since the head of the second clause, $p(X)$, matches any call.
- *Matching only clause* ℓ_1. This case is unfeasible as well since every atom that unifies with $p(a)$ will also unify with $p(X)$.
- *Matching only clause* ℓ_2. This case is clearly feasible with, e.g., $p(b)$, since $p(b)$ unifies with $p(X)$ but it does not unify with $p(a)$. Thus $p(b)$ is our next initial goal.

In the second iteration, $p(b)$ calls $q(b)$ (using clause ℓ_2) and, then, successfully matches clause ℓ_3. Since we only have one clause defining $q/1$, the only alternative consists in producing an initial call to $p/1$ that i) unifies with clause ℓ_2 but not with clause ℓ_1 and, then, ii) calls $q/1$ but fails. Unfortunately, since the approach of Mesnard *et al.* [4] cannot represent negative constraints, the algorithm tries to find an instance $p(X)\sigma$ of $p(X)$ such that $q(X)\sigma$ does not unify with $q(b)$. A possible solution is then $p(a)$. Observe that this goal will not achieve the desired result (matching clause ℓ_2 and then fail) since it will match clause ℓ_1 and terminate successfully. Indeed, since $p(a)$ was already considered, the concolic testing algorithm of Mesnard *et al.* [4] terminates computing the test cases $\{p(a), p(b)\}$, which is unnecessarily incomplete.

For example, if we assume that the domain comprises at least one more constant, say c, then the right set of test cases should be $\{p(a), p(b), p(c)\}$, so that the last test case actually matches clause ℓ_2 and then fails. In this work, we overcome the above problem by introducing constraints, which can represent both positive and negative information. In particular, the search for an instance of $p(X)$ that first unifies with clause ℓ_2 only, and then fails, is represented as follows:

$$p(X) \neq p(a) \wedge \forall Y \ (p(X) \neq p(Y) \vee q(Y) \neq q(b))$$

Solving this constraint (using an SMT solver) would produce the desired test case, $p(c)$, thus achieving a full path coverage.

For this purpose, we have designed a concolic testing tool for logic programs that is based on the following principles:

- As in the approach of Mesnard *et al.* [4], we instrument a *deterministic* semantics for logic programs (inspired by the linear semantics of Ströder *et al.* [9]) in order to perform both concrete and symbolic execution in parallel.
- In contrast to previous approaches, our instrumented semantics also considers negative constraints, so that the problems mentioned above can be avoided (i.e., our implemented semantics is complete in more cases).
- Finally, the generated constraints are solved using a state-of-the-art constraint solver, namely the Z3 SMT solver [1]. This allows us to make concolic testing more efficient in practice.

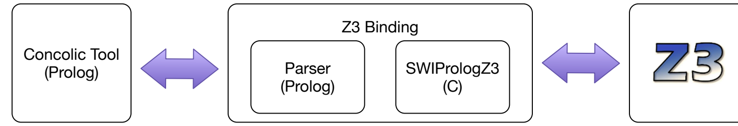

Fig. 1. Implementation workflow.

Table 1. Summary of experimental results

Subject program	size	Initial goal	Ground Args	Max Depth	time concolic	time contest	#TCs concolic	#TCs contest
Nat	2	nat(0)	1	1	0.050	0.0273	3	4
Nat	2	nat(0)	1	5	0.0897	0.1554	7	12
Nat	2	nat(0)	1	50	1.6752	19.5678	52	102
Generator	7	generate(empty,_A,_B)	1	1	1.4517	0.7096	9	9
Generator	7	generate(empty,T,_B)	2	1	1.3255	4.4820	9	9
Generator	7	generate(empty,T,H)	3	1	1.3211	crash	9	N/A
Activities	38	what_to_do_today(sunday, sunny,wash_your_car)	3	2	6.3257	timeout	122	N/A
Cannibals	78	start(config(3,3,0,0))	1	2	0.0535	timeout	2	N/A
Family	48	parent(dicky,X)	1	1	20.0305	64.1838	9	19
Monsters and mazes	113	base_score(will,grace)	2	2	0.2001	0.4701	6	7

2 A Concolic Testing Tool for Prolog

Our prototype is implemented in SWI-Prolog [12] and the Z3 SMT solver [1], as depicted in Figure 1. Regarding the termination of concolic testing, we impose a maximum term depth for the generated test cases. Since the domain is finite and we do not generate duplicated test cases, termination is trivially ensured.

Let us show some selected results from a preliminary experimental evaluation of our concolic testing tool. We selected six programs from previous benchmarks [4] and from GitHub[4]. We ran concolic testing between 3 and 100 executions on a MacBook Pro hexacore 2,6 Ghz with 16 GB RAM in order to get reliable results. Reported times, in seconds, are the average of these executions. Our results are reported in Table 1. Here, concolic refers to our tool, while contest refers to the tool introduced by Mesnard *et al.* [4]; the size of a subject program is the number of its source lines of code; column Ground Args displays the number of ground arguments in the initial symbolic goal; and #TCs refers to the number of generated test cases. A timeout for contest is set to 1000 seconds (the crash is an overflow).

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, our tool performs up to 10 times faster than contest. We note that the number of test cases generated by the tools are not comparable

[4] https://github.com/Anniepoo/prolog-examples

since our new framework avoids a source of incompleteness (as mentioned in the previous section), 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.

3 Conclusion

In this paper, we report our experience in the development of an SMT-based concolic testing tool that is based on the approach of *Mesnard et al.* [4] but adds support for negative constraints, thus overcoming some of the limitations of previous approaches [5,6]. Our preliminary experimental evaluation has shown promising results regarding the scalability of the method.

Recently, concolic testing has been extended to CLP programs [7], so that both positive and negative constraints can be represented in a natural way. As future work, we plan to extend our concolic testing tool to the case of CLP programs.

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.: On the completeness of selective unification in concolic testing of logic programs. In LOPSTR'16. Lecture Notes in Computer Science, vol. 10184, pp. 205–221. Springer (2017)
6. Mesnard, F., Payet, É., Vidal, G.: Selective unification in constraint logic programming. In: Vanhoof, W., Pientka, B. (eds.) PPDP. pp. 115–126. ACM (2017)
7. Mesnard, F., Payet, É., Vidal, G.: Concolic Testing in CLP. CoRR **abs/2008.00421** (2020), https://arxiv.org/abs/2008.00421
8. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/ FSE. pp. 263–272. ACM (2005)
9. Ströder, T., Emmes, F., Schneider-Kamp, P., Giesl, J., Fuhs, C.: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog. In: LOPSTR'11. pp. 237–252. Springer LNCS 7225 (2011)
10. Tikovsky, J.R.: Concolic testing of functional logic programs. In: Declarative Programming and Knowledge Management, pp. 169–186. Springer (2017)
11. Vidal, G.: Concolic execution and test case generation in Prolog. In LOPSTR 2014. Lecture Notes in Computer Science, vol. 8981, pp. 167–181. Springer (2015)
12. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. TPLP **12**(1-2), 67–96 (2012). https://doi.org/10.1017/S1471068411000494