Random vs. Structure-Based Testing of Answer-Set Programs: An Experimental Comparison[⋆]

Tomi Janhunen[1], Ilkka Niemelä[1], Johannes Oetsch[2], Jörg Pührer[2], and Hans Tompits[2]

[1] Aalto University, Department of Information and Computer Science,
P.O. Box 15400, FI-00076 Aalto, Finland
{Tomi.Janhunen,Ilkka.Niemela}@aalto.fi

[2] Technische Universität Wien, Institut für Informationssysteme 184/3,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. Answer-set programming (ASP) is an established paradigm for declarative problem solving, yet comparably little work on testing of answer-set programs has been done so far. In a recent paper, foundations for structure-based testing of answer-set programs building on a number of coverage notions have been proposed. In this paper, we develop a framework for testing answer-set programs based on this work and study how good the structure-based approach to test input generation is compared to random test input generation. The results indicate that random testing is quite ineffective for some benchmarks, while structure-based techniques catch faults with a high rate more consistently also in these cases.
Keywords: answer-set programming, structure-based testing, random testing

1 Introduction

Answer-set programming (ASP) is an established approach to declarative problem solving with increasingly efficient solver technology enabling applications in a large range of areas. However, comparably little effort has been spent on software development methods and, in particular, on developing testing techniques for answer-set programs.

In previous work [1], we introduced a *structure-based testing approach* for answer-set programs. In this approach, testing is based on test cases that are chosen with respect to the internal structure of a given answer-set program. More specifically, we introduced different notions of *coverage* that measure to what extent a collection of test inputs covers certain important structural components of a program. Another typical method used in conventional software development is *random testing*. Although random testing is able to discover some errors quite quickly, it often needs to be complemented with other techniques to increase test coverage. The effectiveness of such more systematic testing methods is usually evaluated by comparison to random testing [2].

In this paper, we continue our work on testing and develop methods for *structure-based test input generation* based on our previous work [1] and for *random test input generation*. The basic research question we address is how the structure-based approach

[⋆] This work was partially supported by the Austrian Science Fund (FWF) under project P21698 and the Academy of Finland under project 122399.

to test input generation compares to random test input generation. To this end, we evaluate structure-based and random testing using benchmark problems from the second ASP competition [3]. The evaluation is based on *mutation analysis* [4], i.e., program instances are rendered incorrect by injecting faults according to a mutation model. Then, the two approaches are compared on the basis how well they can catch such faults.

2 Background

We deal with a class of logic programs that corresponds to the ground fragment of the `Gringo` input language [5, 6], which extends normal programs by *aggregate atoms*. The semantics of this class is defined in terms of *answer sets* (also known as *stable models*) [7]. We denote the collection of all answer sets of a program P by $\mathrm{AS}(P)$. Given a program P and an interpretation I, the set of *supporting rules of P with respect to I*, $\mathrm{SuppR}(P, I)$, consists of all rules $r \in P$ whose body is true in I. Moreover, for an ordinary atom a, $\mathrm{Def}_P(a)$ is the set of *defining rules* of a in P, i.e., the set of all rules in P where a occurs positively in the rule head (possibly within an aggregate atom).

We next review some definitions related to testing logic programs [1]. With each program P, we associate two finite sets of ordinary atoms, $\mathbb{I}_P$ and $\mathbb{O}_P$, which are the respective input and output signatures of P. A *test input*, or simply *input*, of P is a finite set of atoms from $\mathbb{I}_P$. Given some input I of P, the *output* of P with respect to I, $P[I]$, is defined as $P[I] = \{S \cap \mathbb{O}_P \mid S \in \mathrm{AS}(P \cup I)\}$. Moreover, a *specification s* for P is a mapping from the set of test inputs of P into sets of subsets of $\mathbb{O}_P$. P is considered to be *correct* with respect to its specification s if $P[I] = s(I)$ for each test input I of P. Moreover, we define (i) a *test case* for P as a pair $\langle I, O\rangle$, where I is an input of P and $O = s(I)$, (ii) a *test suite* for P as a set of test cases for P, and (iii) the *exhaustive test suite* for P as the collection of all test cases for P. We say that P *passes* a test case $T = \langle I, O\rangle$ if $P[I] = O$, and P *fails* for T otherwise; and P passes a test suite $\mathcal{S}$ if P passes all test cases in $\mathcal{S}$, and P fails for $\mathcal{S}$ otherwise.

Given a program P and an input I of P, a proper rule $r \in P$ (i.e., a rule which is not a constraint) is *positively covered* (resp., *negatively covered*) by I if $r \in \mathrm{SuppR}(P, X)$ (resp., $r \notin \mathrm{SuppR}(P, X)$) for some $X \in \mathrm{AS}(P \cup I)$. A non-empty definition $\mathrm{Def}_P(a)$ of an ordinary atom a appearing in P is *positively covered* (resp., *negatively covered*) by I if there is some $X \in \mathrm{AS}(P \cup I)$ such that $a \in X$ (resp., $a \notin X$). On the other hand, a constraint $c \in P$ is *covered* by I if the body of c is true in some answer set $X \in \mathrm{AS}((P \setminus \{c\}) \cup I)$, i.e., c is temporarily removed from P for this notion. For any of these coverage notions, we say that a test suite $\mathcal{S}$ yields *total coverage* on a program P if each rule (definition, constraint) in P which is covered (positively and negatively) by an input of the exhaustive test suite for P is covered by an input from $\mathcal{S}$.

3 Experimental Setup

Our experimental setup involves six steps: (1) selecting some benchmark programs; (2) grounding them to obtain reference programs; (3) injecting faults into the reference programs to generate mutants; (4) generating random test suites for mutants; (5) generating structure-based test suites for mutants; and (6) comparing the resulting test suites. We describe these steps in more detail in what follows.

(*1*) *Selection of Benchmark Instances.* We consider programs from the benchmark collection of the second ASP competition [3] for our experiments. In particular, we selected the problems *Maze Generation*, *Solitaire*, *Graph Partitioning*, and *Reachability* to represent typical program structures in ASP. The Maze Generation problem is about generating a maze on a two-dimensional grid. The problem encoding is prototypical for the guess-and-check paradigm in ASP and involves inductive definitions to express the reachability conditions. A distinguishing feature of this problem is that solutions are partially fixed in the input and completed in the output. Solitaire is a simple planning game that is played on a grid with 33 cells. The structure of the encoding is slightly simpler than for Maze Generation since no positive recursion is involved. Graph Partitioning is a graph-theoretical problem about partitioning the nodes of a weighted graph subject to certain conditions. It is representative for hard problems on linked data structures that involve integer calculations. Finally, the objective of the Reachability problem is to exhibit a path between two dedicated nodes in a directed graph. Reachability is a representative for problems solvable in polynomial time. The central aspect is to efficiently compute the transitive closure of the edge relation of the input graph.

(*2*) *Grounding.* Since our testing approach is defined for propositional programs, we need to ground the programs from Step (1). Uniform ASP encodings usually involve a natural parameter that allows one to scale the size of ground instances. For Maze Generation, we used a bound on the size of the grid of 7×7 to obtain a finite grounding. Since Solitaire is a planning problem, we parameterized it by setting the upper bound on the lengths of plans to 4. For Graph Partitioning, since it takes graphs as input, one natural scaling parameter is a bound on the number of nodes which we fixed to 7. Furthermore, we fixed a bound of 3 on the maximal number of partitions and a bound of 20 on maximal weights assigned to edges. For Reachability, we fixed an upper bound of 6 on the number of nodes as the scaling parameter. We refer to the ground program obtained in this step as *reference program* since it later serves as a *test oracle*, i.e., as a method to determine the correct output given some input.

(*3*) *Fault Injection.* Based on grounded encodings from the ASP competition, we follow an approach that is inspired by *mutation analysis* [4]. In this step, we apply certain *mutation operations* on a program to obtain versions, so called *mutants*, where small faults have been injected. Intuitively, mutation operations mimic typical mistakes made by a programmer like omission of certain elements or misspelling of names. In particular, we generate different classes of mutants, one class for each mutation operation. To this end, we consider (i) deleting a single body literal, (ii) deleting a single rule of a program, (iii) swapping the polarity of literals, (iv) increasing or decreasing the bound of an aggregate by one, and (v) replacing an atom by another atom appearing in the program. For each grounded benchmark encoding, we generated different classes of mutants, in particular, one class of mutants for each mutation operation. The bound modification was not applied to the Maze Generation and Reachability instances which do not involve aggregate atoms nor bounds. Each mutant was generated by applying a mutation operation precisely once. Each class consists of 100 mutants which are publicly available.[3] We used dedicated equivalence tests [8] to eliminate equivalent mutants that cannot be distinguished from the reference program by any test input.

(*4*) *Random Test Suite.* In software testing, a rigorous theoretical analysis of the fault detection capabilities of systematic testing methods is practically infeasible. What usually is done is that systematic methods are compared to the *random-testing standard* [2]. Random testing means that test inputs are randomly selected from the input space—usually a uniform distribution is used. For each benchmark problem in our experiments, we formalized the preconditions as a simple program mainly consisting of integrity constraints. Any model of such a program constitutes an admissible input, i.e., one which is consistent with the preconditions of the encoding. Hence, the task of generating admissible random inputs can be reduced to computing uniformly-distributed answer sets of logic programs.

Our approach to random testing is to incrementally and in a highly randomized way build an answer set of the program that formalizes the preconditions of a benchmark problem. The algorithm takes a program P with input signature $\mathbb{I}_P$ and generates a random test input I for P such that $P[I] \neq \emptyset$ incrementally starting from the empty set. For each atom from $\mathbb{I}_P$, one after the other, we decide by tossing a coin whether or not it should be contained in the final input. Furthermore, we always check if such a decision allows to proceed in a way that the final input is consistent with the program. This check is realized using ASP and a method which does not guarantee strictly uniform distribution of inputs as a reasonable compromise. For each mutant of each benchmark class, we generated 1000 random inputs.

(*5*) *Structure-Based Test Suite.* We next review our approach to generate test inputs that yield total coverage. First, we sketch how we can use ASP itself to generate covering inputs. Let P be a program with input signature $\mathbb{I}_P$. We define the *input generator* for P, denoted $\mathrm{IG}(P)$, as the program $\{a' \leftarrow \mathrm{not}\ a''; a'' \leftarrow \mathrm{not}\ a'; a \leftarrow a' \mid a \in \mathbb{I}_P\}$ where all primed and double primed atoms do not occur anywhere else. A program P is labeled for rule coverage as the program $\mathrm{LR}(P) = \{\mathrm{H}(r) \leftarrow r'; r' \leftarrow \mathrm{B}(r) \mid r \in P\}$ where r' is a globally new label for each $r \in P$. To cover individual rules either positively or negatively, we use programs $P_r^+ = \mathrm{LR}(P) \cup \{\leftarrow \mathrm{not}\ r'\}$ and $P_r^- = \mathrm{LR}(P) \cup \{\leftarrow r'\}$, respectively. Then, the inputs for P that cover r positively and negatively are in a one-to-one correspondence with the respective sets $\{X \cap \mathbb{I}_P \mid X \in \mathrm{AS}(\mathrm{IG}(P) \cup P_r^+)\}$ and $\{X \cap \mathbb{I}_P \mid X \in \mathrm{AS}(\mathrm{IG}(P) \cup P_r^-)\}$ of inputs. Similar reductions can be devised for the other coverage notions. The preconditions of each benchmark program P form a set C of constraints which can be incorporated to the preceding reductions in this way restricting the reductions to admissible test inputs of P. Given a mutated program P, we generate test suites that yield total rule coverage for P using the reduction P_r^+. We first generate a test suite $\mathcal{S}$ that obtains total positive rule coverage for P. To this end, we pick a rule $r \in P$ not yet positively covered and check whether there is an input and a corresponding answer set X that positively covers r. If such an input exists, r is marked as positively covered, if not, r is marked as positively failed, meaning that no inputs exist that cover r positively. Then, some simple bookkeeping takes place: any rule that is positively, resp., negatively, covered by the answer set X is marked accordingly. The procedure iterates until all rules are marked as positively covered or failed. A similar

[3] `http://www.kr.tuwien.ac.at/research/projects/mmdasp/mutant.tgz.`

Table 1. Catching rates for the Maze Generation and Reachability benchmark.

Maze Generation

	size of test suite	AR	LD	RD	PS	Total
Random Testing	1000	0.03	0.18	0.00	0.16	0.09
Definition Coverage	36	0.67	0.63	0.74	0.78	0.71
Rule Coverage	85	0.78	0.66	0.78	0.75	0.74
Constraint Coverage	176	0.81	0.74	0.81	0.90	0.82
Definition & Constraint Coverage	212	0.86	0.87	0.85	0.93	0.88
Rule & Constraint Coverage	261	0.89	0.88	0.86	0.94	0.89

Reachability

	size of test suite	AR	LD	RD	PS	Total
Random Testing	1000	0.61	0.56	0.59	0.69	0.61
Definition Coverage	37	0.09	0.17	0.00	0.01	0.07
Rule Coverage	592	0.47	0.67	0.07	0.63	0.46
Constraint Coverage	36	0.15	0.02	0.10	0.07	0.09
Definition & Constraint Coverage	73	0.21	0.19	0.10	0.08	0.15
Rule & Constraint Coverage	628	0.56	0.69	0.17	0.64	0.52

procedure is then applied to extend $\mathcal{S}$ to obtain total negative rule coverage. The method for the other notions of coverage is analogous. Note that for a program P, there is a test suite that obtains total rule, definition, or constraint coverage whose size is bounded by the number of rules in P. Though our method does not guarantee minimality of a test suite, it always produces one within this bound.

(*6*) *Comparison.* Finally, the different classes of test inputs are evaluated with respect to their potential to *catch* mutants, i.e., to reveal the injected faults. To determine whether a mutant passes a test case, we use the reference program as a test oracle. Given a class M of mutants and a test suite $\mathcal{S}$, the catching rate of $\mathcal{S}$ on M is the ratio of mutants in M that fail for $\mathcal{S}$ and the total number of mutants in M.

4 Results and Discussion

Our test results in terms of catching rates for the Maze Generation and Reachability benchmarks are summarized in Table 1. The mutation operations studied were atom replacement (AR), literal deletion (LD), rule deletion (RD), and literal polarity swapping (PS). Besides definition, rule, and constraint coverage, we considered the combinations of definition and rule coverage with constraint coverage. The column "size of test suite" in the tables refers to the size of the generated test suites for each mutant class. For the coverage-based suites, the numbers are averages over all test suites in a class.

For Maze Generation, random testing only obtains very low catching rates, while systematic testing yields total rates of up to 0.9 using a significantly smaller test suite. One explanation for the poor performance of random testing is that the probability that a random test input is consistent with a mutant is very low. Inputs that yield no or few outputs for a mutant seem to have a low potential for fault detection. The situation is different for the Reachability benchmark: random testing performs slightly better than systematic testing. This is mainly explained by the higher number of test inputs used for random testing. The conclusion is that the considered structural information seems to provide no additional advantage compared to randomly picking inputs for this benchmark. As regards structural testing, the best detection rate is obtained by combining rule and constraint coverage. However, this comes at a cost of larger test suites compared to the other coverage notions. We conducted similar experiments using the Graph Partitioning and Solitaire benchmarks. Interestingly, it turned out that testing seems to be trivial in these cases. Since both random and structural testing achieved a constant catching rate of 1.00 for all mutant classes, we omit the respective tables. In fact, almost any test input catches all mutants for these programs. This nicely illustrates how the non-determinism involved in answer-set semantics affects testing. While in conventional testing only a single execution results from a particular test input, the situation in ASP is different. Given a test input, the output of a uniform encoding can consist of a vast number of answer sets, each of which potentially revealing a fault. For example, typical random inputs for Solitaire yield tens of thousands of answer sets which gives quite a lot information for testing. In other words, the probability that some effects of a fault show up in at least one answer set seems to be quite high.

For some encodings, systematic testing gives a clear advantage over random testing while for other programs, it could not be established that one approach is better than the other. These findings are in principle consistent with experiences in conventional software testing. The results indicate that random testing is quite effective in catching errors provided that sufficiently many admissible test inputs are considered. There is no clear-cut rule when to use random or systematic testing. The advantage of random testing is that inputs can be easily generated, directly based on a specification, and that the fault detection rates can be surprisingly good. The advantage of structure-based testing is that resulting test suites tend to be smaller which is especially important if testing is expensive. For some problems, structure-based testing is able to detect faults at a high rate but random testing is very ineffective even with much larger test suites.

References

1. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: Proc. ECAI 2010. IOS Press (2010) 951–956
2. Hamlet, R.: Random testing. In: Encyclopedia of Software Engineering, Wiley (1994) 970–978
3. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The second answer set programming competition. In: Proc. LPNMR 2009. Volume 5753 of LNCS. Springer (2009) 637–654
4. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection help for the practicing programmer. IEEE Computer **11**(4) (1978) 34–41
5. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Proc. LPNMR 2007. Volume 4483 of LNCS. Springer (2007) 266–271
6. Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder gringo. In: Proc. LPNMR 2009. Volume 5753 of LNCS, Springer (2009) 502–508
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. 5th Logic Programming Symposium, MIT Press (1988) 1070–1080
8. Janhunen, T., Oikarinen, E.: LPEQ and DLPEQ – translators for automated equivalence testing of logic programs. In: Proc. LPNMR 2004. Volume 2923 of LNCS. Springer (2004) 336–340