On Testing Answer-Set Programs[1]

Tomi Janhunen,[2] **Ilkka Niemelä,**[2] **Johannes Oetsch,**[3] **Jörg Pührer,**[3] and **Hans Tompits**[3]

Abstract. Answer-set programming (ASP) is a well-acknowledged paradigm for declarative problem solving, yet comparably little effort has been spent on the investigation of methods to support the development of answer-set programs. In particular, *systematic testing of programs*, constituting an integral part of conventional software development, has not been discussed for ASP thus far. In this paper, we fill this gap and develop notions enabling the structural testing of answer-set programs, i.e., we address testing based on test cases that are chosen with respect to the internal structure of a given answer-set program. More specifically, we introduce different notions of *coverage* that measure to what extent a collection of test inputs covers certain important structural components of the program. In particular, we introduce metrics corresponding to path and branch coverage from conventional testing. We also discuss complexity aspects of the considered notions and give strategies how test inputs that yield increasing (up to total) coverage can be automatically generated.

1 INTRODUCTION

Answer-set programming (ASP) is not only one of the currently most widely-used computational approaches for realising nonmonotonic reasoning but also constitutes a viable paradigm for declarative problem solving. Indeed, the development of increasingly efficient solver technology allowed ASP to become an important host language for computing reasoning problems from diverse areas of AI like planning, diagnosis, information integration, and inheritance reasoning. The basic idea of ASP is to encode a problem instance as a logic program such that its models, referred to as *answer sets*, provide the solutions to the given instance (see the article by Gelfond and Leone [5] for an overview on ASP).

Although ASP is regarded as a programming paradigm, comparably little effort has been spent on the investigation of methods to support the development of answer-set programs. In particular, the *systematic testing of programs*, constituting a key element in a typical software development process, has been identified as a potentially crucial step in an answer-set programming methodology [3] but has not been formally studied for ASP so far. In this paper, we fill this gap and lay down the foundations for testing answer-set programs.

In a conventional setting, testing aims at increasing the reliability of a software component by executing the code with the intent of finding errors, i.e., mismatches between the actual and expected program output for some given input [12]. Contrary to verification, testing is in general unsuitable to establish total correctness of a program relative to some specification but can be seen as a computationally lighter approach where programming mistakes can be detected from the actual code. An obvious desideratum in testing is to obtain small collections of test cases with a high potential to detect errors. In this paper, we focus on test strategies that deduce test cases from the internal program structure. Hence, we follow principles of *structural testing*, also known as *white-box testing* [7, 12, 13].

In structural testing, *test coverage* plays an important role to measure the degree to which extent test cases cover the logic of a program. In procedural languages, the concept of *path coverage* aims at test cases that try out each execution path through a program component at least once. Although this sounds like a reasonable objective in the first place, complete path coverage is not feasible in general due to a combinatorial explosion. This is why weaker notions of coverage such as *branch coverage* have been introduced. Branch coverage requires only test cases such that each edge in the control-flow graph of the program is traversed at least once. More precisely, a branch is a decision that can have a true or a false outcome, like, e.g., checking the conditions of `if-then-else` and `do-while` statements. As ASP lacks an explicit notion of execution, conventional coverage notions do not apply and novel declarative concepts that reflect the structure of answer-set programs are required.

Structural testing for logic programming has been considered already for Prolog [8, 2]. In particular, Belli and Jack [2] have developed coverage notions based on the computational model of Prolog: A program is executed by posing a query against it, and answers are computed using SLD resolution. Consequently, goals are understood as input while the output corresponds to specific information that is extracted from a program via resolution and unification. In contrast, output in ASP corresponds to the answer sets of a program and input corresponds to a set of facts. Hence, the setting in these works is quite incompatible with that in ASP and cannot be used directly.

Our goal is to develop a general approach to systematic structural testing of answer-set programs. In particular, we focus on the analysis of different coverage notions and propose methods for test automation in ASP. The main contributions of this paper are as follows:

- We introduce a general framework for testing answer-set programs, specifying conditions under which a particular test case passes or fails, and define basic test coverage notions corresponding to path and branch coverage.
- We study results on the relations between the different test coverage notions and analyse their computational complexity.
- We lay down basic techniques for test automation using ASP itself, viz. for deciding test verdicts, determining coverage, and generating test cases with increasing coverage.

2 PRELIMINARIES

A *normal logic program*, or simply a *program*, is a finite set of rules of form

[1] This research has been partially supported by the Academy of Finland under project #122399 and the Austrian Science Fund (FWF) under grant P21698.

[2] Aalto University, Department of Information and Computer Science, P.O. Box 15400, FI-00076 Aalto, Finland.

[3] Technische Universität Wien, Institut für Informationssysteme 184/3, Favoritenstraße 9–11, A-1040 Vienna, Austria.

$$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n, \quad (1)$$

where $a, b_1, \ldots, b_m, c_1, \ldots, c_n$ are propositional atoms from some fixed denumerable alphabet At, and "not" denotes *default negation*. For a rule r of form (1), define the *head* of r as $H(r) = \{a\}$, the *positive body* of r as $B^+(r) = \{b_1, \ldots, b_m\}$, and the *negative body* of r as $B^-(r) = \{c_1, \ldots, c_n\}$. Furthermore, the *body* of r is $B(r) = B^+(r) \cup \text{not } B^-(r)$, where $\text{not } B^-(r) = \{\text{not } c \mid c \in B^-(r)\}$. A *fact* is a rule r satisfying $B(r) = \emptyset$.

An *interpretation* is a finite set of atoms from At. Given some rule r, an interpretation I *satisfies* $B(r)$, denoted $I \models B(r)$, iff both $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. The interpretation I satisfies a rule r of form (1), denoted $I \models r$, iff $I \cap H(r) \neq \emptyset$ whenever $I \models B(r)$. An interpretation I is a *model* of P, $I \models P$, if $I \models r$, for each rule $r \in P$.

Let HB_P denote the *Herbrand base* of a program P. Given an atom $a \in \text{HB}_P$, $\text{Def}_P(a) = \{r \in P \mid H(r) = \{a\}\}$ is the set of *defining rules* of a in P, i.e., the *definition* of a. Furthermore, $\text{SuppR}(P, I) = \{r \in P \mid I \models B(r)\}$ is the set of *supporting rules* of P under an interpretation I.

The *positive dependency graph* of a program P is the directed graph $\langle V, A\rangle$ where $V = \text{HB}_P$ and $\langle a, b\rangle \in A$ iff there is a rule $r \in P$ such that $a \in H(r)$ and $b \in B^+(r)$. A non-empty set L of atoms is a *loop* of a program P iff, for each pair $a, b \in L$ of atoms, there is a path π of length greater than or equal to 0 from a to b in the positive dependency graph of P such that each atom in π is in L.

Following Gelfond and Lifschitz [6], the *reduct* of a program P with respect to an interpretation I is the program $P^I = \{H(r) \leftarrow B^+(r) \mid r \in P \text{ and } B^-(r) \cap I = \emptyset\}$. An interpretation I is an *answer set* of P iff I is a minimal model of the reduct P^I. By $\text{AS}(P)$, we denote the collection of all answer sets of P. We say that an atom a is a *brave* (resp., *cautious*) *consequence* of P if a is in some (resp., each) answer set of P. Likewise, $\text{not } a$ is a brave (resp., cautious) consequence of P if a is not in some (resp., each) answer set of P.

3 TEST CASES FOR ASP

To apply systematic testing to answer-set programs, we make the notions of input and output for a program explicit. In this respect, we aim at a setting that reflects the common practice in ASP to write a program as a *uniform* problem encoding that, given a problem instance as input in form of a set of facts, has answer sets filtered to dedicated output atoms encoding the desired solutions. Hence, for distinguishing specific signatures for input and output symbols (as done in other work [4, 9]), we assume two finite sets $\mathbb{I}_P, \mathbb{O}_P \subseteq At$ of atoms for each program P, where $\mathbb{I}_P$ is the *input alphabet* of P and $\mathbb{O}_P$ is the *output alphabet* of P.

We first address the output of a program subject to input and output alphabets. Given a collection S of interpretations and some set $A \subseteq At$, $S|_A$ denotes the collection of interpretations in S projected on the atoms in A, formally $S|_A = \{I \cap A \mid I \in S\}$.

Definition 1. *For a program P with input and output alphabets $\mathbb{I}_P$ and $\mathbb{O}_P$, an* input *of P is any set $I \subseteq \mathbb{I}_P$ of atoms, and the* output *of P with respect to some input I of P is the set $P[I] = \text{AS}(P \cup I)|_{\mathbb{O}_P}$, where $P \cup I$ stands for P augmented by a fact $a \leftarrow$ for each $a \in I$.*

Example 1. *Consider a program P based on the following rules: $e \leftarrow d, \text{not } f$; $d \leftarrow a, b, \text{not } c$; $f \leftarrow c, \text{not } e$; and $c \leftarrow e, f$. Assume that $\mathbb{I}_P = \{a, b, c\}$ and $\mathbb{O}_P = \{e, f, g\}$. Let $I_1 = \{a, b\}$ and $I_2 = \{c\}$ be two inputs of P. Then, the resulting sets of outputs are $P[I_1] = \text{AS}(P \cup \{a, b\})|_{\mathbb{O}_P} = \{\{e\}\}$ and $P[I_2] = \{\{f\}\}$.*

Following conventional software testing [12], a test case for a program P consists of a precise description of the correct output of P given some input of P. Since answer-set programs are inherently non-deterministic, there can be several outputs or even no output for an actual input $I \subseteq \mathbb{I}_P$ supplied to a program P. Thus, we define a *specification* for a program P as a mapping σ from sets over $\mathbb{I}_P$ to collections of sets over $\mathbb{O}_P$. Then, the correct outputs for P for a given input I are determined by $\sigma(I)$. A specification σ can be seen as a *test oracle* determining the correct outputs for any input.

Definition 2. *Let P be a program and σ a specification for P. Then, a* test case *T for P and σ is a pair $\langle I, O\rangle$, where $I \subseteq \mathbb{I}_P$ is an input of P and $O = \sigma(I)$. The sets I and O are denoted by $\text{inp}(T)$ and $\text{out}(T)$, respectively.*

Definition 3. *A* test suite *$\mathcal{S}$ for some program P and some specification σ for P is a collection of test cases for P and σ. The collection of inputs of $\mathcal{S}$ is given by $\text{inp}(\mathcal{S}) = \{\text{inp}(T) \mid T \in \mathcal{S}\}$.*

The exhaustive test suite *for P and σ is the suite $\mathcal{E}_{P,\sigma} = \{\langle I, O\rangle \mid I \subseteq \mathbb{I}_P \text{ and } O = \sigma(I)\}$.*

As for each program P and each specification σ for P it holds that $\text{inp}(\mathcal{E}_{P,\sigma}) = 2^{\mathbb{I}_P}$, we call $2^{\mathbb{I}_P}$ the *exhaustive input collection* for P. Note that, for brevity, we usually leave the specification σ implicit and simply write $\mathcal{E}_P$ instead of $\mathcal{E}_{P,\sigma}$ in such a case.

Definition 4. *Let P be a program and $T = \langle I, O\rangle$ a test case for P. Then, P* passes *T if $P[I] = O$, otherwise P* fails *T.*

Likewise, a program P passes *a test suite $\mathcal{S}$ for P if P passes each test case $T \in \mathcal{S}$ and* fails *$\mathcal{S}$ otherwise.*

To distinguish a slightly weaker notion, we say that P is *compliant* with a test case $T = \langle I, O\rangle$ iff $P[I] \subseteq O$, i.e., P provides only correct but not necessarily all outputs for the input I.

Example 2. *The program P from Example 1 fails the test case $T = \langle\{a, b\}, \{\{e\}, \{f\}\}\rangle$ but is compliant with T.*

Definition 5. *Let P be a program and σ a specification for P. Then, P is* correct with respect to σ *if P passes the test suite $\mathcal{E}_{P,\sigma}$.*

Having fixed basic terminology and notation for testing, we proceed with our central concepts to realise structural testing in ASP.

4 COVERAGE METRICS

In this section, we present concepts analogous to path and branch coverage from conventional software testing for ASP. We first define the concept of a coverage function and show its inherent boundaries. Then, we define specific coverage functions for different entities, like rules, programs, definitions, and loops, respectively, and analyse the interconnections of these metrics. Finally, an example is given which illustrates the usefulness of our notions for identifying relevant test cases.

4.1 Coverage Functions

Traditionally, coverage is a relation between the input of test cases and particular syntactic entities (branching statements, execution paths, etc.) of a program. Coverage can thus be quantified by the sum of covered elements relative to the total number of (coverable) elements. The following central notion adopts conditions following Jack [8].

Definition 6. *A function γ from programs and collections of test inputs to the interval $[0, 1]$ is a* coverage function *if, for each program P and each collection $\mathcal{I} \subseteq 2^{\mathbb{I}_P}$ of inputs of P,*

(i) $\gamma(\mathcal{I}, P) = 1$ *if* $\mathcal{I} = 2^{\mathbb{I}_P}$*, and*

(ii) $\gamma(\mathcal{I}', P) \leq \gamma(\mathcal{I}, P)$*, for each collection of inputs* $\mathcal{I}' \subseteq \mathcal{I}$ *of* P*.*

We say that a collection $\mathcal{I} \subseteq 2^{\mathbb{I}_P}$ of inputs of a program P yields *total coverage* for P with respect to γ if $\gamma(\mathcal{I}, P) = 1$. Likewise, a test suite $\mathcal{S}$ yields total coverage for P with respect to γ if $\mathrm{inp}(\mathcal{S})$ yields total coverage for P with respect to γ. Note that the exhaustive input collection $2^{\mathbb{I}_P}$ trivially yields total coverage for any P with respect to γ. Thus, we call a coverage function γ *trivial* if, for any program P, only the input collection $2^{\mathbb{I}_P}$ yields total coverage for γ.

We call γ *clairvoyant* if, for each program P, each specification σ for P, and each test suite $\mathcal{S}$ for P and σ, if P passes $\mathcal{S}$ and $\mathcal{S}$ yields total coverage for P with respect to γ, P is correct with respect to σ.

It is well-known in traditional software testing that the correctness of a program cannot be established by structural testing because this kind of testing is inherently unable to detect that certain parts of a specification are not implemented. This limitation, made explicit in the following theorem, can be observed for testing in ASP as well, no matter how a notion of structural testing is designed.

Theorem 1. *No coverage function is both non-trivial and clairvoyant.*

Proof. Assume γ is a non-trivial and clairvoyant coverage function. As γ is non-trivial, there is a program P, a specification σ for P, and a test suite $\mathcal{S}$ for P and σ with $\mathrm{inp}(\mathcal{S}) \subset 2^{\mathbb{I}_P}$ and $\gamma(\mathrm{inp}(\mathcal{S}), P) = 1$. Since γ is clairvoyant, P is correct with respect to σ. From $\mathrm{inp}(\mathcal{S}) \subset 2^{\mathbb{I}_P}$, it follows that there is a test case T with $T \in \mathcal{E}_{P,\sigma}$ but $T \notin \mathcal{S}$. Define specification σ' for P as σ except that $\sigma'(\mathrm{inp}(T)) = \emptyset$ if $\sigma(\mathrm{inp}(T)) \neq \emptyset$, and $\sigma'(\mathrm{inp}(T)) = \{\emptyset\}$ otherwise. Clearly, $\mathcal{S}$ is a test suite for P and σ'. Furthermore, $\mathcal{S}$ yields total coverage for P with respect to γ and P passes $\mathcal{S}$ but P is not correct with respect to σ', as it fails the test case $\langle \mathrm{inp}(T), \sigma'(\mathrm{inp}(T))\rangle$. This is a contradiction to the assumption that γ is clairvoyant. □

Note that Theorem 1 is a very general result, not depending on the computational model of ASP, that carries over to structural testing in other programming paradigms when coverage functions of the kind as defined above are considered.

In what follows, we define specific coverage functions based on different notions of coverage. In particular, for a given class X of entities (like programs, rules, loops, etc.), we provide a function $\mathrm{covered_X}(\mathcal{I}, P)$—being, roughly speaking, determined as the number of entities in X which are covered by some input I from a collection $\mathcal{I}$ of inputs of P—from which coverage with respect to X is defined by means of what we call the *basic coverage function schema*:

$$\mathcal{C}_\mathrm{X}(\mathcal{I}, P) = \begin{cases} \dfrac{\mathrm{covered_X}(\mathcal{I}, P)}{\mathrm{covered_X}(2^{\mathbb{I}_P}, P)}, & \text{if } \mathrm{covered_X}(2^{\mathbb{I}_P}, P) > 0, \\ 1, & \text{otherwise}. \end{cases} \quad (2)$$

Based on this schema, we introduce the notion of *program coverage* in the next subsection that can be seen as a declarative analogue of path coverage for answer-set programs. In Section 4.3, we provide the notions of *rule*, *loop*, *definition*, and *component coverage* that amount to branch-coverage counterparts of program coverage. For each branch-like coverage notion X, positive and negative X coverage are defined such that total X coverage holds exactly if both total positive and total negative X coverage do. Note that the notions considered in this paper are tailored towards testing consistent answer-set programs.

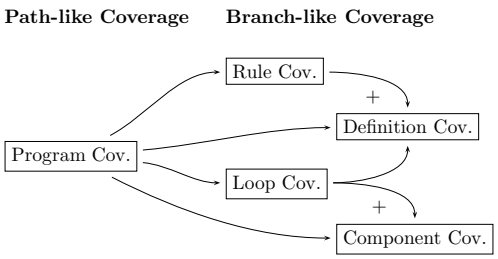

Figure 1. Relations between total coverage of different notions.

Figure 1 summarises the central relations between the coverage metrics as discussed below. An unlabelled edge from notion X to notion Y means that for a program P and a collection $\mathcal{I}$ of inputs of P, total X coverage for $\mathcal{I}$ and P implies total Y coverage for $\mathcal{I}$ and P. Moreover, an edge labelled with "+" means that total positive X coverage for $\mathcal{I}$ and P implies total positive Y coverage for $\mathcal{I}$ and P.

4.2 Path-like Test Coverage

We first define the notion of program coverage as an analogue of path coverage in conventional testing.

Definition 7. *Let P be a program, I an input of P, and $P' \subseteq P$. Then, I* covers *P' if $P' = \mathrm{SuppR}(P, X)$, for some $X \in \mathrm{AS}(P \cup I)$. Furthermore, a test case T for P* covers *P' if $\mathrm{inp}(T)$ covers P'.*

Given a collection $\mathcal{I}$ of inputs of P, by $\mathrm{covered_P}(\mathcal{I}, P)$ we understand the number of subsets of P that are covered by some input $I \in \mathcal{I}$. It is easy to check that instantiating the basic coverage function schema (2) with $\mathrm{covered_P}(\cdot, \cdot)$ (i.e., setting $\mathrm{X} = \mathrm{P}$) yields a coverage function. Accordingly, we call $\mathcal{C}_\mathrm{P}(\mathcal{I}, P)$ the *program coverage of $\mathcal{I}$ for P*.

Note that program coverage is a non-trivial coverage function. Thus, by Theorem 1, total program coverage for a program P and a test suite $\mathcal{S}$ for P does not necessarily imply that P is correct whenever P passes $\mathcal{S}$.

The significance of the following result is that total program coverage for a test suite $\mathcal{S}$ and a program P enforces that each answer set (modulo projection) of P joined with some input of P can be obtained by considering the inputs from the test cases in $\mathcal{S}$ only.

Theorem 2. *Let P be a program, $\mathcal{S}$ a test suite for P, and A the set of all atoms $a \in \mathrm{HB}_P$ such that $\mathrm{Def}_P(a) \neq \emptyset$.*

Then, $\mathcal{C}_\mathrm{P}(\mathrm{inp}(\mathcal{S}), P) = 1$ implies that, for each input I of P and for each $X \in \mathrm{AS}(P \cup I)$, $\mathcal{S}$ contains a test case T such that $X \cap A = Y \cap A$, for some $Y \in \mathrm{AS}(P \cup \mathrm{inp}(T))$.

Path coverage in conventional testing considers test cases to exercise each execution path of a component's control-flow graph. The number of paths is exponential in the number of branching statements in the worst case. Analogously, program coverage in ASP yields test cases that consider all possible bi-partitions of P into supporting and non-supporting sets of rules with respect to some answer set. The number of such partitions is exponential in the number of rules in general. While path coverage yields exhaustive test cases concerning the possible paths through a component, program coverage is exhaustive regarding the possible answer sets of a program (see Theorem 2).

4.3 Branch-like Test Coverage Notions

As an approximation of path coverage in conventional testing, branch coverage considers only test cases that cover each edge in the control-flow graph of a component at least once. The following notions approximate program coverage and thus aim at modelling branch coverage in ASP.

Rule coverage. We start with rule coverage that relates test cases to individual rules of a program such that single rules in a program are supporting at least once.

Definition 8. *Given a program P and an input I of P, a rule $r \in P$ is* positively covered *by I if $X \models B(r)$ for some answer set $X \in \mathrm{AS}(P \cup I)$, and r is* negatively covered *by I if $X \not\models B(r)$ for some answer set $X \in \mathrm{AS}(P \cup I)$. Furthermore, r is positively (resp., negatively) covered by a test case T for P if it is positively (resp., negatively) covered by* $\mathrm{inp}(T)$.

Given a collection $\mathcal{I}$ of inputs of P, by $\mathrm{covered}_{\mathrm{R}^+}(\mathcal{I}, P)$ (resp., $\mathrm{covered}_{\mathrm{R}^-}(\mathcal{I}, P)$) we understand the number of rules in P that are positively (resp., negatively) covered by some input $I \in \mathcal{I}$. Moreover, $\mathrm{covered}_{\mathrm{R}}(\mathcal{I}, P) = \mathrm{covered}_{\mathrm{R}^+}(\mathcal{I}, P) + \mathrm{covered}_{\mathrm{R}^-}(\mathcal{I}, P)$. We define, mutatis mutandis, the values $\mathcal{C}_{\mathrm{R}^+}(\mathcal{I}, P)$, $\mathcal{C}_{\mathrm{R}^-}(\mathcal{I}, P)$, and $\mathcal{C}_{\mathrm{R}}(\mathcal{I}, P)$ as instances of schema (2) as before (again giving rise to coverage functions) and refer to them as *positive rule coverage*, *negative rule coverage*, and *rule coverage* of $\mathcal{I}$ for P, respectively.

Example 3. *Recall program P from Example 1 with $\mathbb{I}_P = \{a, b, c\}$ and $\mathbb{O}_P = \{e, f, g\}$. Consider the inputs $I_1 = \{a, b\}$ and $I_2 = \{c\}$ of P. Since the unique answer set of $P \cup I_1$ is $\{a, b, d, e\}$, I_1 covers the first and the second rule in P positively and all other rules negatively. For I_2, the unique answer set of $P \cup I_2$ is $\{c, f\}$ which covers the third rule positively and all other rules negatively.*

Note that it is not always possible to positively or negatively cover a rule by an input. Re-examining the rules from Example 1 reveals that no test case is able to positively cover the last rule. On the other hand, facts are rules which cannot be negatively covered. Furthermore, for the program P and an input collection $\mathcal{I}$ consisting of I_1 and I_2 from Example 3, we get a coverage of $\mathcal{C}_{\mathrm{R}}(\mathcal{I}, P) = 1$. Thus, we obtain total rule coverage using two test cases only.

The next result shows how rule and program coverage are related:

Theorem 3. *For each program P and each collection $\mathcal{I}$ of inputs of P, total program coverage implies total rule coverage of $\mathcal{I}$ for P.*

Proof. Assume that $\mathcal{I}$ yields total program coverage but not total rule coverage for P, i.e., some input $I \in 2^{\mathbb{I}_P}$ positively (or negatively) covers a rule $r \in P$ but no input in $\mathcal{I}$ positively (or negatively) covers r. As I positively (negatively) covers r, we have $r \in P'$ ($r \notin P'$), for an $X \in \mathrm{AS}(P \cup I)$ with $P' = \mathrm{SuppR}(P, X)$. By definition, I covers P'. From total program coverage, some $I' \in \mathcal{I}$ must cover P'. Thus, for some $X' \in \mathrm{AS}(P \cup I')$, $P' = \mathrm{SuppR}(P, X')$. As $r \in P'$ ($r \notin P'$), I' positively (negatively) covers r which contradicts that no input in $\mathcal{I}$ positively (negatively) covers r. □

We continue with coverage notions that complement rule coverage to yield relevant test cases which would be missed otherwise.

Loop coverage. While rule coverage is related to classical models of a program, the following coverage notion is concerned with loops which capture positive recursion between rules. The relevance of loops for ASP is well acknowledged in the literature [11, 10].

Definition 9. *Let P be a program and I an input of P. A loop L of P is* positively covered *by I if there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that for every atom $a \in L$ there is a rule $r \in \mathrm{SuppR}(P, X)$ that defines a, and L is* negatively covered *by I if for some $X \in \mathrm{AS}(P \cup I)$ there is an $a \in L$ such that $\mathrm{Def}_P(a) \neq \emptyset$ and no rule $r \in \mathrm{SuppR}(P, X)$ defines a. As well, L is positively (resp., negatively) covered for a test case T if* $\mathrm{inp}(T)$ *positively (resp., negatively) covers L.*

Similar to the above, we define, mutatis mutandis, the (*positive* or *negative*) *loop coverage* values $\mathcal{C}_{\mathrm{L}^+}(\mathcal{I}, P)$, $\mathcal{C}_{\mathrm{L}^-}(\mathcal{I}, P)$, and $\mathcal{C}_{\mathrm{L}}(\mathcal{I}, P)$ like their corresponding notions for rule coverage.

Example 4. *Consider the program $P = \{a \leftarrow b, c;\ b \leftarrow a, d;\ e \leftarrow b, c, \mathrm{not}\ d\}$ with $\mathbb{I}_P = \{a, b, c, d\}$. Besides the singleton loops $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, and $\{e\}$, P has the loop $L = \{a, b\}$. It is easily verified that rule coverage can be achieved with inputs $\{b, c\}$ and $\{a, d\}$. Notably, neither of these test cases accounts for the mutual dependency of the first and the second rule due to the loop L. Loop coverage, however, would require an input, e.g., $\{a, c, d\}$, that explicitly covers L.*

Note that rule coverage and loop coverage are not directly comparable; neither notion implies the other. However, we can relate loop coverage and program coverage by the following result:

Theorem 4. *For each program P and each collection $\mathcal{I}$ of inputs of P, total program coverage implies total loop coverage for $\mathcal{I}$ and P.*

The proof is analogous to the proof of Theorem 3. Loop coverage comes with a decisive drawback: a program contains, in general, exponentially many loops compared to the number of rules. We thus introduce next two coverage notions that further approximate loop coverage. On the one hand, definition coverage will address singleton loops, and, on the other hand, component coverage will address maximal loops. Both the number of singletons and maximal loops are bounded by the number of rules in a program.

Definition coverage. Rule coverage concentrates on the satisfaction of rule bodies in a program P. Each body $B(r)$ can be viewed as a conjunction, but there are also disjunctions implicitly present in a logic program: Given a program P, the rules involved in the definition of $\mathrm{Def}_P(a) = \{r_1, \ldots, r_n\}$ of an atom $a \in \mathrm{HB}_P$ effectively stand for $B(r_1) \vee \ldots \vee B(r_n)$. If we try to cover both truth values of such disjunctions, we arrive at the following notion:

Definition 10. *Let P be a program and I an input of P. An atom a is* positively covered *by I if there is some $r \in \mathrm{SuppR}(P, X)$ that defines a for an answer set $X \in \mathrm{AS}(P \cup I)$, and a is* negatively covered *by I if $\mathrm{Def}_P(a) \neq \emptyset$ and no $r \in \mathrm{SuppR}(P, X)$ defines a for an $X \in \mathrm{AS}(P \cup I)$. Moreover, a is positively (resp., negatively) covered by a test case T for P if it is positively (resp., negatively) covered by* $\mathrm{inp}(T)$.

Again, we define the (*positive* or *negative*) *definition coverage* values $\mathcal{C}_{\mathrm{D}^+}(\mathcal{I}, P)$, $\mathcal{C}_{\mathrm{D}^-}(\mathcal{I}, P)$, and $\mathcal{C}_{\mathrm{D}}(\mathcal{I}, P)$ similar to rule and loop coverage. Since atoms can be regarded as conditions in a rule body that trigger the activation of a rule, definition coverage, roughly speaking, relates to *condition coverage* for conventional software testing which requires that all Boolean subexpressions are exercised.

Example 5. *Recall program P from Example 1 and the inputs I_1 and I_2 from Example 3. It can be verified that the input collection $\{I_1, I_2\}$ yields total definition coverage for P.*

Definition coverage is a special case of loop coverage in the sense that a is positively (resp., negatively) covered by some input I iff the singleton loop $\{a\}$ is positively (resp., negatively) covered by I.

Theorem 5. *For each program P and each collection $\mathcal{I}$ of inputs of P, total loop coverage implies total definition coverage for $\mathcal{I}$ and P.*

Furthermore, total positive rule coverage implies total positive definition coverage for $\mathcal{I}$ and P.

Component coverage. The subset-maximal loops of a program P correspond to the sets of nodes of the strongly connected components (SCC) of the positive dependency graph of P. Vis-à-vis to definition coverage, we define (strongly connected) *component coverage* as an approximation of loop coverage.

Definition 11. *Let P be a program and I an input of P. An SCC C of P is* positively covered *by I if there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that for every node a in C, there is some $r \in \mathrm{SuppR}(P, X)$ that defines a, and C is* negatively covered *by I if there is some $X \in \mathrm{AS}(P \cup I)$ such that for every node a in C, $\mathrm{Def}_P(a) \neq \emptyset$ and there is no $r \in \mathrm{SuppR}(P, X)$ that defines a.*

As for rules, loops, and definitions, we define the (*positive* or *negative*) *component coverage* values $\mathcal{C}_{\mathrm{C}^+}(\mathcal{I}, P)$, $\mathcal{C}_{\mathrm{C}^-}(\mathcal{I}, P)$, and $\mathcal{C}_{\mathrm{C}}(\mathcal{I}, P)$ using the general schema (2).

Theorem 6. *For each program P and each set $\mathcal{I}$ of inputs of P, total program coverage implies total component coverage for $\mathcal{I}$ and P.*

Furthermore, total positive loop coverage implies total positive component coverage for $\mathcal{I}$ and P.

Note that the latter property holds by definition, as every SCC corresponds to a maximal loop. Negative component coverage requires that no node in the component is supported whereas only one unsupported atom is sufficient for negative loop coverage. Hence, total negative loop coverage does not guarantee total negative component coverage. Component coverage is, however, designed as an approximation of loop coverage. This is formalised by the next result:

Theorem 7. *Given a program P, an input I of P, and an SCC C of P with set L of nodes, positive component coverage for C by I implies positive loop coverage for each loop $L' \subseteq L$ of P by I. Likewise, negative component coverage for C implies negative loop coverage for each $L' \subseteq L$.*

4.4 An Illustrative Example

We round off this section by elucidating the advantages of structure-based testing compared to random testing by means of an example that is concerned with the formalisation of some logical condition—a rather common pattern in logic programming.

Assume that we want to encode the condition

$$\varphi_n = (p_1 \wedge \cdots \wedge p_{n-1}) \rightarrow p_n$$

by a logic program P_n such that $\mathbb{I}_{P_n} = \{p_1, \cdots, p_n\}$, $\mathbb{O}_{P_n} = \{t\}$, and $P_n[I] = \{\{t\}\}$ iff $I \models \varphi_n$.

Consider the following *incorrect* realisation of the program P_n: $\{c \leftarrow p_1, \ldots, p_{n-1};\ f \leftarrow c, p_n;\ t \leftarrow \text{not } f\}$. The first rule states that the conjunction in φ_n is true iff all its operands are true. The second rule is a failed attempt to express that the implication in φ_n is false iff the antecedent is true and the conclusion is false: a negation preceding p_n was accidentally omitted by the programmer. The third rule states that $I \models \varphi_n$ iff $I \not\models \varphi_n$ does not hold.

Note that the exhaustive test suite $\mathcal{E}_{P_n}$ for P_n consists of 2^n test cases but P fails only two test cases. If we randomly pick a test case T from $\mathcal{E}_{P_n}$, the limit probability that P fails T is 0 as n approaches infinity; thus, random testing is unlikely to reveal this error for large n.

We next consider rule coverage to guide the generation of test cases. To positively cover the second rule, we need a test case with input $I = \{p_1, \ldots, p_n\}$, which covers the first rule positively and the third rule negatively as well. This test input reveals the error already since $P_n[I]$ yields$\{\emptyset\}$ instead of $\{\{t\}\}$. A second test case suffices to obtain total rule coverage, hence we need only two test cases to obtain total rule coverage and to reveal the error.

5 COMPLEXITY ASPECTS

We now turn to complexity issues related to coverage problems and analyse the inherent complexity of relevant decision problems. We assume that the reader is familiar with basic notions from complexity theory. Recall that $\mathrm{D^P}$ is the class of decision problems that can be characterised by a conjunction of an NP and an independent coNP problem. First, we note that in the general case determining the test verdict for a test case is a challenging computational task:

Theorem 8. *Given a program P and a test case T, determining whether P passes T is $\mathrm{D^P}$-complete.*

Proof (*sketch*). Given a program P and a test case T, determining whether P is compliant with T can be shown to be coNP-complete. The problem of deciding whether $\mathrm{out}(T) \subseteq P[\mathrm{inp}(T)]$ holds can be shown to be NP-complete. It follows that deciding whether P passes T is $\mathrm{D^P}$-complete. □

Theorem 9. *Given a program P, $P' \subseteq P$, and an input I of P, deciding whether I covers P' is computable in polynomial time.*

Proof (*sketch*). The following procedure decides whether I covers P' in polynomial time: Define X as the smallest set such that (i) $I \subseteq X$ and (ii) for each $r \in P'$, $H(r) \cup B^+(r) \subseteq X$. Then, I covers P' iff $P' = \mathrm{SuppR}(P, X)$ and $X \in \mathrm{AS}(P \cup I)$. □

Theorem 10. *Given a program P and an input I of P, deciding whether (i) a rule r, (ii) an atom a, (iii) a loop L, or (iv) an SCC C in P is positively (resp., negatively) covered by I is* NP*-complete, respectively.*

Proof (*sketch*). For membership, guess an interpretation X over $\mathrm{HB}_P \cup I$ and check in polynomial time whether $X \in \mathrm{AS}(P \cup I)$ and (i) $X \models B(r)$, for positive rule coverage, (ii) $\mathrm{Def}_P(a) \cap \mathrm{SuppR}(P, X) \neq \emptyset$, for positive definition coverage, (iii) for each $m \in L$, $\mathrm{Def}_P(m) \cap \mathrm{SuppR}(P, X) \neq \emptyset$, for positive loop coverage, and (iv) for each $n \in C$, $\mathrm{Def}_P(n) \cap \mathrm{SuppR}(P, X) \neq \emptyset$, for positive component coverage. Membership for notions of negative coverage can be shown analogously. Also, hardness follows by suitable reductions from the problem of deciding brave consequence. □

Theorem 11. *Given a program P and a collection of its inputs $\mathcal{I}$, deciding whether (i) $\mathcal{I}$ yields total program coverage for P is* coNP*-complete, (ii) $\mathcal{I}$ yields total rule, definition, or component coverage for P is in Δ_2^P, and (iii) $\mathcal{I}$ yields total loop coverage for P is in Π_2^P.*

Proof (*sketch*). We give a schema for showing membership that applies to cases (i)–(iii). To decide the complementary problem of total coverage, for some X in P, where X is a set of rules, a rule, a definition, a loop, etc., check if both (a) some test input in $2^{\mathbb{I}_P \cap \mathrm{HB}_P}$ covers

X and (b) no test input in $\mathcal{I}$ covers X. For Item (i), (a) and (b) can be decided in polynomial time. This implies coNP-membership for deciding total program coverage. Hardness can be shown by a simple reduction from program inconsistency to total program coverage. For Item (ii), $|P|$ is an upper bound for the number of rules and SCCs in a program, as well as of the number of atoms in a definition. Thus, X can be enumerated in polynomial time. Problem (a) can be decided in NP and (b) can be decided in coNP which implies Δ_2^P membership for total rule, definition, or component coverage. For Item (iii), X can be guessed. Problem (a) can be decided in NP and (b) can be decided in coNP which implies Π_2^P membership for total loop coverage. □

Note that, for a program P, $2^{|P|}$ is an asymptotic upper bound for the size of a minimal test suite $\mathcal{S}$ for P that yields total program or loop coverage. For rule, definition, and component coverage, a respective upper bound is $|P|$. Hence, the possibility of a compact test suite for the latter case—essential for testing in practice—comes at a computational cost which is presumably unavoidable.

6 TEST AUTOMATION

For automating key tasks of testing answer-set programs, we could use any suitable programming paradigm. However, in this section, we discuss how the key tasks can be implemented using ASP techniques so that ASP solvers can be used directly for test automation.

Determining test verdicts. In testing, one of the key tasks is to decide test verdicts. We start by outlining how to use ASP techniques for this. Given a program P and a test case $T = \langle I, O\rangle$, we assume that the correct outputs O are given by an ASP program. A natural way is to encode each set $O' = \{a_1, \ldots, a_n\} \in O$ as a rule

$$r(O'): \quad ok \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m,$$

where $\{b_1, \ldots, b_m\} = \mathbb{O}_P \setminus \{a_1, \ldots, a_n\}$. More compact encodings are often possible where the idea is that for the encoding program $\Pi(O)$ we set the input alphabet $\mathbb{I}_{\Pi(O)} = \mathbb{O}_P$ and the output alphabet $\mathbb{O}_{\Pi(O)} = \{ok\}$ such that $\Pi(O)[O'] = \{\{ok\}\}$ iff $O' \in O$. Now, P is compliant with a test case $T = \langle I, O\rangle$ iff ok is a cautious consequence of the program $P \cup I \cup \Pi(O)$. A program P passes a test case $T = \langle I, O\rangle$ iff P is compliant with T and $O \subseteq P[I]$. A straightforward approach to checking the latter condition is by determining whether for every $O' \in O$, ok is a brave consequence of $P \cup I \cup \{r(O')\}$. Using a more involved translation, deciding whether $O \subseteq P[I]$ holds can be reduced to a single brave consequence check as suggested by Theorem 8.

Checking coverage. Given a test input, evaluating coverage for the branch-like notions is NP-hard as shown in Theorem 10. However, we can use ASP techniques to check coverage. E.g., given a program P and an input I of P, we can determine whether a rule $r \in P$ is covered positively (resp., negatively) by I by checking whether sat (resp., not sat) is a brave consequence of $P \cup I \cup \{sat \leftarrow B(r)\}$. Other coverage notions can be handled in a similar way.

Generating covering test cases. In testing, there are various strategies to obtain coverage. In randomised testing, the idea is to generate random test inputs and observe how different coverage metrics evolve when more tests are run. This approach can be used for testing ASP programs, too. When using randomly generated test inputs, it is typically possible to increase coverage only up to a limited degree. For a more intelligent goal directed testing, techniques for generating test inputs guaranteed to increase coverage (up to total coverage) are needed. ASP techniques can also be used for this task, for example, by adding to a program P the set of rules $C(\mathbb{I}_P) = \{a \leftarrow \text{not } a' \mid a \in \mathbb{I}_P\} \cup \{a' \leftarrow \text{not } a \mid a \in \mathbb{I}_P\}$. Then, test inputs covering a given element (a rule, a loop, etc.) can be obtained using brave reasoning techniques as described above provided that witnessing input for a brave consequence is returned by the reasoning engine. For example, a test input covering a rule r positively can be obtained (if it exists) by checking whether sat is a brave consequence of $P \cup C(\mathbb{I}_P) \cup \{sat \leftarrow B(r)\}$, provided that the reasoning engine answering this query is able to accommodate a witnessing answer set from which a test input can be extracted (if such an answer set exists). A simple way of implementing such a brave reasoning engine is to use an ASP solver to look for an answer set of the program $P \cup C(\mathbb{I}_P) \cup \{sat \leftarrow B(r)\} \cup \{\leftarrow \text{not } sat\}$.

7 CONCLUSION

In this paper, we laid the foundation for a formal and systematic study of testing for ASP. Besides fostering future research, a direct benefit of this work is that it forms the basis for tools that support the generation of high coverage test suites. We expect that such tools will significantly facilitate the development of answer-set programs, especially as part of integrated-development environments (IDEs).

For the sake of a clear presentation, we restricted ourselves to propositional normal programs. Further classes of programs, like disjunctive programs or programs with variables, are left for future work. Also, coverage notions designed to yield test cases for detecting program inconsistencies are not discussed here due to space reasons. Another interesting question is how different notions of modularity can be employed to reduce the computational costs of testing. Complementing this work on structural testing, we also plan to study principles from *functional* or *black-box testing* [12, 1].

REFERENCES

[1] B. Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, 1999.

[2] F. Belli and O. Jack, 'Declarative paradigm of test coverage', *Softw. Test, Verif. Reliab*, **8**(1), 15–47, (1998).

[3] M. Brain, O. Cliffe, and M. De Vos, 'A pragmatic programmer's guide to answer set programming', in *Proc. SEA*, pp. 49–63, (2009).

[4] M. Gelfond and A. Gabaldon, 'Building a knowledge base: An example', *Annals of Mathematics and Artificial Intelligence*, **25**(3-4), 165–199, (1999).

[5] M. Gelfond and N. Leone, 'Logic programming and knowledge representation - the A-Prolog perspective', *Artificial Intelligence*, **138**(1-2), 3–38, (2002).

[6] M. Gelfond and V. Lifschitz, 'The stable model semantics for logic programming', in *Proc. ICLP/SLP*, pp. 1070–1080, (1988).

[7] J. C. Huang, 'An approach to program testing', *ACM Computing Surveys*, **7**(3), 113–128, (September 1975).

[8] O. Jack, *Software Testing for Conventional and Logic Programming*, Walter de Gruyter & Co. Hawthorne, NJ, USA, 1996.

[9] T. Janhunen, E. Oikarinen, H. Tompits, and S. Woltran, 'Modularity aspects of disjunctive stable models', *Journal of Artificial Intelligence Research*, **35**, 813–857, (August 2009).

[10] J. Lee, 'A model-theoretic counterpart of loop formulas', in *Proc. IJCAI 2005*, pp. 503–508. Professional Book Center, (2005).

[11] F. Lin and Y. Zhao, 'ASSAT: Computing answer sets of a logic program with SAT solvers', *Artificial Intelligence*, **157**(1–2), 115–137, (2004).

[12] G. J. Myers, *Art of Software Testing*, John Wiley & Sons, Inc., New York, NY, USA, 1979.

[13] S. C. Ntafos, 'A comparison of some structural testing strategies', *IEEE Transactions on Software Engineering*, **14**(6), 868–874, (June 1988).