Institute of Computer Science
Knowledge Processing and Information Systems

Computing coverage metrics for answer set programms

Bachelorthesis

Jakob Westphal

May 25, 2023

First supervisor: Prof. Dr. Torsten Schaub
Second supervisor: Tobias Stolzmann

Abstract

Answer Set Programming (ASP) is a popular declarative programming paradigm. However, relatively little work has been done on improving the development process for answer set programs. Workflow tools such as a testing suite are an integral part of the software development process for other programming paradigms but are almost nonexistent for ASP. The concept of code coverage is widely used during testing to evaluate or generate new test cases.

In this thesis, we use the existing definitions of five code coverage metrics for propositional normal logic programs and develop a practical way to compute them using the power of ASP itself. We also investigate ideas how these metrics can be extended to also be applicable to more complex program classes. Specifically, we define syntactic transformations of answer set programs which allow us to compute all five coverage metrics for multiple test cases all at once. We also implement these transformations in a prototype coverage testing tool. Finally, we show that the chosen approach to checking coverage for propositional programs can be easily adapted to also work with further program classes.

Zusammenfassung

Answer Set Programming (ASP) ist ein beliebtes deklaratives Programmierparadigma. Allerdings war die Verbesserung des Entwicklungsprozesses von Answer Set Programmen bisher nur selten Thema der Forschung. Workflow-Tools wie eine Testsuite sind ein wichtiger Teil des Softwareentwicklungsprozesses bei anderen Programmierparadigmen, existieren aber kaum für ASP. Das Konzept der Testabdeckung ist weit verbreitet beim Testen um vorhandene Testfälle zu evaluieren oder neue zu generieren.

In dieser Arbeit benutzen wir die bereits existierenden Definitionen von fünf Testabdeckungsmetriken für propositionale Logikprogramme und entwickeln einen praktischen Weg diese mit Hilfe von ASP zu berechnen. Wir untersuchen zudem Ansätze, diese Metriken so zu erweitern, dass sie auch auf komplexere Programmklassen angewendet werden können. Konkret definieren wir syntaktische Transformationen von Answer Set Programmen, die es uns erlauben, alle fünf Coverage Metriken für beliebig viele Testfälle gleichzeitig zu berechnen. Außerdem implementieren wir diese Transformationen in dem Prototyp eines Coverage-Test-Tools. Schließlich zeigen wir, dass der gewählte Ansatz um Code Coverage für propositionale Programme zu testen so angepasst werden kann, dass er auch bei komplexeren Programmklassen eingesetzt werden kann.

Contents

1 Introduction

Answer Set Programming (ASP) [Lif19] is one of the most successful programming paradigms for declarative problem solving in particular in the field of knowledge representation and reasoning. As such, it has found applications in many different fields both in academic and industry work [EGL16]. However, even though it has been around for over 20 years [Nie99; MT99], there is still a significant lack of good workflow tools. Tools like a testing suite, a debugging tool or an annotation language can help make the development process more streamlined and allow different approaches that have proven very effective in conventional programming languages such as test driven development (TDD, [Fra+03]). Research into these topics exists and has yielded results, however many topics remain unexplored.

This thesis will focus on one of these mostly unexplored topics which is code coverage for ASP. Code coverage is a concept that is widely used in conventional programming languages. Its efficacy has been evaluated many times and it has proven to be one of the most practical ways to test the adequacy of existing test suites or to generate new test suites with high fault detection [GJG14]. Unfortunately, the coverage concepts used in conventional testing cannot be easily applied to answer set programs. Notions such as *path coverage* or *branch coverage* common in procedural programming languages rely on evaluating paths through the control flow graph of a program. Such a graph cannot be constructed for answer set programs since, due to their declarative nature, no explicit notion of execution exists.

Testing and coverage has been discussed for declarative programming. [BJ98] introduces coverage notions for logic programs based on the Prolog model of posing queries to a program and resolving them using SLD resolution. The concepts of unification and anti-instances are used to define the coverage metrics. This means that even though Prolog is a declarative language and in many ways similar to ASP, these notions are not compatible with ASP as it relies on rule instantiation instead of unification. Work has also been done on testing constraint programs, another paradigm used in logic programming, however it was concluded, that developing notions of test coverage would not be possible in this case [LGL10].

Finally, while general approaches to testing in ASP have been discussed multiple times, e.g., in [GOT17; ABR21; Oet22], the only attempt at realising code coverage for answer set programs has been done in [Jan+10]. There, five coverage metrics are defined for propositional normal programs based on the existing concepts of path and branch coverage in conventional testing.

The goal of this thesis is to find a way to implement these metrics by relying as much as possible on the power of ASP itself. To achieve this, we will introduce syntactic transformations for answer set programs. These transformations add new atoms to the program which make it possible to compute the coverage metrics by solving the program with the integrated ASP system clingo. This approach is also implemented in a prototype coverage testing tool. Additionally, since propositional normal programs are rare in a practical setting, an informal attempt at extending the coverage metrics so they are applicable to more complex program classes is made.

This work is structured as follows: Chapter 2 serves as a short introduction to ASP, establishing some necessary definitions. In chapter 3 we go over the existing coverage metrics for answer set programs, explaining their functionality and purpose. Chapter 4 first introduces our approach to computing each coverage metric. Then, details are given on how this approach was implemented in a prototype tool. Finally, chapter 5 discusses how the coverage metrics might be adjusted to function with program classes beyond propositional normal programs and shows how our syntactic transformations can easily adapt these changes.

2 Preliminaries on Answer Set Programming

This chapter will provide a short introduction to ASP and lie down the basic definitions that will be used in the rest of the thesis. As mentioned in Chapter 1, the coverage metrics are restricted to propositional normal programs. Hence we will focus on defining these type of programs instead of the larger class of ASP programs.

2.1 Answer Set Programs

A normal logic program is a finite set of rules. These rules take the form

$$a \leftarrow b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_n,$$

where $a, b_1, \dots, b_m, c_1, \dots, c_n$ are propositional atoms with $m,\ n \geq 0$ and "not" denotes *negation-as-failure*. For such a rule, the *head* of the rule r is defined as $H(r) = \{a\}$, the *positive body* $B^+(r) = \{b_1, \dots, b_m\}$ and the *negative body* $B^-(r) = \{c_1, \dots, c_n\}$. Finally, the *body* is the union of the positive body and the default negation of each atom in the negative body $B(r) = B^+(r) \cup \{\text{not } c \mid c \in B^-(r)\}$. A rule r is called a *fact* if $B(r) = \emptyset$ and r is called a *constraint* if $H(r) = \emptyset$. All atoms in a program come from the same fixed alphabet A.

An *interpretation* I is a finite set of atoms from the alphabet A. This interpretation *satisfies* the body of a rule r, written as $I \models B(r)$, if and only if $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. Given this definition one can then say that I satisfies a rule r or $I \models r$, iff $I \cap H(r) \neq \emptyset$ whenever $I \models B(r)$. For a program P, an interpretation is a *model* of P, $I \models P$, iff for every rule $r \in P,\ I \models r$ holds.

Definition 2.1. The *reduct* of a program P relative to an interpretation I is the program $P^I = \{H(r) \leftarrow B^+(r) \mid r \in P,\ B^-(r) \cap I = \emptyset\}$ [GL88].

An interpretation I is called an *answer set* of P, iff it is a minimal model of the reduct P^I, i.e., $I \models P^I$ and there exists no subset $I' \subset I$ for which $I' \models P^I$ holds. The collection of all answer sets of a program P is denoted AS(P).

Example 2.1. Given the program P

$$\begin{aligned} &a \leftarrow \\ &c \leftarrow a, b \\ &d \leftarrow a,\ \text{not } b \end{aligned}$$

and the interpretation $I = \{a, d\}$, the *reduct* of P relative to I is $P^I = \{a \leftarrow\ ;\ c \leftarrow a, b\ ;\ d \leftarrow a\}$. It can then be verified, that the set $\{a, d\}$ is a minimal model of the reduct P^I as it satisfies every rule of the reduct and no subset of it is a model. Therefore the interpretation I is an *answer set* of the program P. Indeed it is the only answer set of the program.

An atom a is called a *brave consequence* of the program P iff it is contained in at least one answer set of P, i.e., there exists an answer set $X \in \text{AS}(P)$ such that $a \in X$. The set of brave consequences of P is $\mathcal{BC}(P) = \bigcup \text{AS}(P)$. Respectively, an atom a is called a *cautious consequence* of P iff it is contained in all the answers sets of P, i.e., for every answer set $X \in \text{AS}(P)$ it holds that $a \in X$. The set of cautious consequences of P is therefore $\mathcal{CC}(P) = \bigcap \text{AS}(P)$.

The *definition* of an atom in a program P is the set of *defining rules* for the atom $a \in A$, given by $\text{Def}_\text{P}(a) = \{r \in P \mid H(r) = \{a\}\}$ Furthermore, the set of *supporting rules* of P under an interpretation I is defined as $\text{SuppR}(P, I) = \{r \in P \mid I \models B(r)\}$

Example 2.2. Consider the program P

$$\begin{aligned} &a \leftarrow \\ &b \leftarrow a,\ \text{not } c \\ &b \leftarrow c,\ \text{not } a \end{aligned}$$

and the interpretation $I = \{a, b\}$, then the set of defining rules of the atom b is $\text{Def}_\text{P}(b) = \{b \leftarrow a,\ \text{not } c\ ;\ b \leftarrow c,\ \text{not } a\}$ and the set of supported rules under I is $\text{SuppR}(P, I) = \{a \leftarrow\ ;\ b \leftarrow a,\ \text{not } c\}$.

Finally we define the *positive atom dependency graph* of a program P as the directed graph $G = (V, E)$ where the vertices are all the atoms occurring in the program and the edge $(a, b) \in E$ exists, iff there is a rule $r \in P$ such that $a \in H(r)$ and $b \in B^+(r)$. For such a graph, a *loop* is a non-empty set L of atoms for which the subgraph of G induced by L is strongly connected. This means that for every pair of atoms $a, b \in L$ there is a path π in G from a to b such that each atom in π is in L. A *strongly connected component* (SCC) of a directed graph G is a loop that is maximal, meaning no additional vertices of G can be added to the set without breaking its property of being strongly connected. Consequently any subset of a strongly connected component is also a loop. A loop that contains exactly one atom is called singleton loop.

Example 2.3. Using the program given in example 2.2, the positive atom dependency graph is defined as $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{(b, a), (b, c)\}$. Therefore, the only loops this program contains are the singleton loops $\{a\}$, $\{b\}$, and $\{c\}$. These are also all strongly connected components. However, if the rules "$a \leftarrow b.$" and "$c \leftarrow a.$" are added to the program, the dependency graph of P will now be $G' = (V, E)$ with $V = \{a, b, c\}$ and $E = \{(b, a), (b, c), (a, b), (c, a)\}$ (see Figure 2.1). In this new graph the set of all loops is $L = \{\{a\}, \{b\}, \{c\}, \{a, b\}, \{a, b, c\}\}$. Thus $\{a, b, c\}$ is the only SCC.

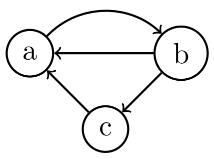

Figure 2.1: Dependency graph G' of the program in example 2.3

2.2 Testing in Answer Set Programming

In order to talk about testing answer set programs, some additional notions need to be established. Conventional testing generally consists of using a specific input to a program and observing whether the output of the program matches the expected output [AO16, pp. 71 sqq.].

In ASP it is common practice to distinguish between a *problem instance* that contains the specific description of one instance of the problem class, represented as a set of facts and a *uniform problem encoding* that describes the general problem class and its solutions. Thus when inputting a problem instance into the problem encoding it will find the corresponding solution and output it in form of answer sets that are usually filtered to dedicated output atoms. It is therefore natural to equate the problem instance to the conventional input and the answer sets encoding the solution to the conventional output. Based on this, we can define the *input alphabet* $\mathbb{I}_P \subseteq A$ as well as the *output alphabet* $\mathbb{O}_P \subseteq A$ of a program as subsets of the alphabet A. The input alphabet contains all the atoms that may occur in a problem instance and the output alphabet contains the atoms relevant for the output.

In conventional software testing, a test case for a program P always consists of an input I of P as well as the expected output of P given I [Sta22]. That way it can be verified, that a program produces the correct output given the input I. For an answer set program we therefore define a test case as a pair (I, O) where I is a set of atoms $I \subseteq \mathbb{I}_P$ represented as facts in a input program and $O \subseteq \mathbb{O}_P$ represents the expected output of P given the input I projected on the output atoms. When talking about code coverage, we are generally only interested in the input of a test case, also written as $\text{inp}(T)$, and not so much the expected output.

A *test suite* for P is a collection of individual test cases and the *exhaustive test suite* for P is the suite ε_P that contains every possible test case for P, meaning that for every possible input $I \subseteq \mathbb{I}_P$ there is a test case in the exhaustive test suite that consists of I and the corresponding output. There are a total of $2^{|\mathbb{I}_P|}$ test cases in the exhaustive test suite and the inputs of ε_P are the power set of the input alphabet $\text{inp}(\varepsilon_P) = 2^{\mathbb{I}_P}$.

Example 2.4. Consider the program P

$$\begin{aligned}
d &\leftarrow a,\ \text{not } b \\
e &\leftarrow b, c \\
f &\leftarrow a, b,\ \text{not } c
\end{aligned}$$

Assuming that $\mathbb{I}_P = \{a, b, c\}$ and $\mathbb{O}_P = \{d, e, f\}$ a possible test case could be the pair (I, O) where $I = \{a, b\}$ and $O = \{f\}$. The exhaustive test suite for this program would contain $2^3 = 8$ different test cases: $T_1 = (\emptyset, \emptyset)$, $T_2 = (\{a\}, \{d\})$, $T_3 = (\{b\}, \emptyset)$, $T_4 = (\{c\}, \emptyset)$, $T_5 = (\{a, b\}, \{f\})$, $T_6 = (\{a, c\}, \{d\})$, $T_7 = (\{b, c\}, \{e\})$, $T_8 = (\{a, b, c\}, \{e\})$

3 Coverage metrics

Coverage metrics in conventional software testing are a method of *structural testing*, also known as *white box testing*. The idea behind white box testing is to examine a programs logic and derive test cases based this information. Therefore, this approach requires explicit knowledge of the internal structure of the program. The more parts of a programs logic (source code) are executed during a test, the more likely it is, that a bug is found. This is where coverage metrics come into play as they measure how much of a program is *covered* – i.e., executed – by a test case.

There are several ways to measure this and thus, many different coverage metrics exist. Most of them take the control flow graph of a program as the basis of analysis. The most complete coverage metric is called *path coverage* and measures how many of the possible paths through the control flow graph of the program are explored by a test case. The problem with this is, that the number of paths is exponential in the number of branching statements in the program and therefore this coverage metric is very expensive. A less expensive alternative in conventional testing that approximates path coverage is called *branch coverage*. This entails executing each branch direction at least once, i.e., covering each edge in the control flow graph at least once [MSB12, pp. 41 sqq.].

While no real equivalent to the control flow graph exists in answer set programming, the coverage metrics that we define can be seen as the counterparts to path and branch coverage in conventional testing.

It is worth noting that due to its focus on the structure of a program, it is impossible to find errors relating to the specification of the program when using white-box testing – be it a faulty specification or the program not adhering to the specification.

3.1 Coverage functions

Simply speaking, the coverage of a test case is defined as the number of elements covered by the input of the test case relative to the total number of (coverable) elements. This general definition allows us to define a function schema, that can be used as the basis for different coverage metrics for different coverable elements (like programs, rules, loops etc.).

For a class X of elements the function $\mathrm{covered}_X(\mathcal{I}, P)$ describes the number of elements of X that are covered by some input collection $\mathcal{I}$ for the program P. It follows that $\mathrm{covered}_X(2^{\mathbb{I}_P}, P)$ describes the maximum number of elements of X in P that can be covered by inputs from the input alphabet $\mathbb{I}_P$. It is important to note that this number is often different from the total number of elements of X that exist in P. Based on this function we introduce the *basic coverage function schema* analogous to [Jan+10] as follows:

$$C_X(\mathcal{I}, P) = \begin{cases} \frac{\mathrm{covered}_X(\mathcal{I},P)}{\mathrm{covered}_X(2^{\mathbb{I}_P},P)}, & \text{if } \mathrm{covered}_X(2^{\mathbb{I}_P}, P) > 0 \\ 1, & \text{otherwise.} \end{cases} \tag{3.1}$$

This function is standardized between 0 and 1 with worse coverage leading to a lower score. We say a collection of inputs $\mathcal{I} \subseteq 2^{\mathbb{I}_P}$ for program P yields *total coverage* with respect to a class X of elements exactly if the coverage function $C_X(\mathcal{I}, P) = 1$.

In the following we will use this schema to define a total of 5 different coverage metrics. For some of these we introduce the notions positive and negative X coverage. Total X coverage holds if and only if both total positive and total negative X coverage hold.

3.2 Path-like coverage

As described above, path coverage in conventional testing is about covering every possible path through the control flow graph of the program. It is the most complete coverage metric in conventional testing, meaning that it includes any other coverage metric [Nta88]. It would therefore be the most ideal coverage metric if one were to disregard the cost of computing this metric. In practice, path coverage is almost never used. If the control flow graph of the program contains loops that can be executed arbitrarily often, the number of possible paths becomes infinite and finding total path coverage goes from being very computationally expensive to actually impossible.

We define here the metric *program coverage* following [Jan+10], that is analogous to conventional path coverage in that it is the most complete coverage metric for answer set programs and it includes all other metrics. It is also very expensive – often unreasonably so. However a clear difference to conventional path coverage is that the maximum coverable elements can only become infinite if the program itself is infinite, as it is dependent on the number of rules in the program.

3.2.1 Program coverage

The idea behind program coverage is to partition the program P into *subprograms* $P' \subseteq P$ and to measure how many of the possible subprograms are covered by a test case.

Definition 3.1. Let P be a program, $P' \subseteq P$ a subprogram of P and I an input for P. We say I *covers* P' iff there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that $P' = \mathrm{SuppR}(P, X)$.

We can thus define $\mathrm{covered}_P(\mathcal{I}, P)$ for a collection of inputs $\mathcal{I}$ for program P as the amount of subprograms of P that are covered by the inputs in $\mathcal{I}$ and consequently instantiate the basic coverage function schema defined in section 3.1 to define $C_P(\mathcal{I}, P)$ the *program coverage of $\mathcal{I}$ for P.*

Example 3.1. Consider the program given in example 2.4. The input $I = \{a, b\}$ has only one answer set: $\mathrm{AS}(P \cup I) = \{\{a, b, f\}\}$. The supported rules of this answer set are: $\mathrm{SuppR}(P, X) = \{f \leftarrow a, b,\ \mathrm{not}\ c\}$ so the given input only covers a single subprogram of P. The total amount of subprograms that exist is $2^{|P|} = 2^3 = 8$, however to calculate the program coverage of I for P we need to first calculate the number of coverable subprograms: $\mathrm{covered}_P(2^{\mathbb{I}_P}, P) = \mathrm{covered}_P(\{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}, P)$. To calculate this we have to compute all the answer sets produced by these inputs and then gather all the different sets of supported rules generated by them. In this case only four unique sets ($\emptyset, \{d \leftarrow a,\ \mathrm{not}\ b\}, \{f \leftarrow a, b,\ \mathrm{not}\ c\}, \{e \leftarrow b, c\}$) are produced and thus only half of the existing subprograms are actually coverable. The program coverage of I for P is therefore $C_P(I, P) = 1/4$.

This notion of program coverage can be very expensive to calculate as, similar to paths in a graph, the number of possible subprograms can be extremely high. In fact it is exponential in the number of rules in a program. However, it is also a very powerful metric that subsumes all other coverage metrics that we will define for answer set programs. This is clear due to the fact that total program coverage of a test suite S for a program P implies, that every obtainable answer set (modulo projection) of P with some input of P can be obtained by using only the inputs in S. This is shown in [Jan+10, Theorem 2]. We arrived at a slightly different result, however we are not conclusively sure which of the two considerations are correct. It is our belief that the theorem should read:

Theorem 3.1. Let P be a program, S a test suite for P and $\mathbb{I}_P$ the input alphabet for P.

Then, $C_P(\text{inp}(S), P) = 1$ implies that, for each input I of P and for each $X \in \text{AS}(P \cup I)$, S contains a test case T such that $X - \mathbb{I}_P = Y - \mathbb{I}_P$ for some $Y \in \text{AS}(P \cup \text{inp}(T))$.

This is because atoms that are both in the input alphabet and definable in P can appear in answer sets without being derived in the program. These edge cases are not covered by program coverage and can thus lead to counterexamples to the original theorem. Nevertheless, the general claim remains the same.

3.3 Branch-like coverage

Since in conventional testing, path coverage is often not realistically achievable, many more metrics have been developed that intend to approximate path coverage while keeping the cost much lower. They generally consist of defining some subset of all paths that needs to be covered. Depending on the size of this subset the metric can be more or less powerful and more or less expensive. One of the most widely used metrics that approximate path coverage is branch coverage, also known as edge coverage. It consists of covering each direction after a branching statement at least once. This is equivalent to requiring each edge of the control flow graph to be covered at least once [AO16, Chapter 7.2].

In the following, we use the definitions given by [Jan+10] to introduce four different coverage metrics that, like branch coverage in conventional testing, try to approximate the path-like metric of program coverage while being simpler to compute.

3.3.1 Rule coverage

We start by introducing the simplest one of the branch-like metrics called *rule coverage*. Here, the focus is on the rules of a program, which means the coverage of a test case T is decided by checking whether each individual rule in the program P is supporting or not for each answer set $X \in \text{AS}(P \cup \text{inp}(T))$.

Definition 3.2. Let P be a program and I an input for P. A rule $r \in P$ is *positively rule covered* by I iff there is an answer set $X \in \text{AS}(P \cup I)$ such that $X \models B(r)$, i.e., the rule is supported by the answer set. On the other hand, a rule r is *negatively rule covered* by I iff there is some answer set X such that $X \not\models B(r)$

For a collection of inputs $\mathcal{I}$ for a program P we define $\text{covered}_{R^+}(\mathcal{I}, P)$ and $\text{covered}_{R^-}(\mathcal{I}, P)$ as the number of rules that are positively (respectively negatively) covered by an input in $\mathcal{I}$. Again, the basic coverage function schema (3.1) is used to define $C_{R^+}(\mathcal{I}, P)$ and $C_{R^-}(\mathcal{I}, P)$ as the *positive and negative rule coverage* of

$\mathcal{I}$ for P. We call $C_R(\mathcal{I}, P)$ the *rule coverage* of $\mathcal{I}$ for P based on the function $\text{covered}_R(\mathcal{I}, P) = \text{covered}_{R^+}(\mathcal{I}, P) + \text{covered}_{R^-}(\mathcal{I}, P)$.

Example 3.2. We look at the program from example 2.4 with two different inputs $I_1 = \{a, b\}$ and $I_2 = \{b, c\}$. The corresponding answer sets are $\text{AS}(P \cup I_1) = \{\{a, b, f\}\} = \{X_1\}$ and $\text{AS}(P \cup I_2) = \{\{b, c, e\}\} = \{X_2\}$. Therefore the first input covers the third rule of the program positively, as $X_1 \models B(r_3)$ holds, and the two other rules negatively, while the second input covers the second rule positively ($X_2 \models B(r_2)$) and the first and third rule negatively. Considering all three rules can be both positively and negatively covered, a test suite S containing these two inputs would yield a positive rule coverage of $C_{R^+}(S, P) = 2/3$, a negative rule coverage of $C_{R^-}(S, P) = 1$ and a rule coverage of $C_R(S, P) = 5/6$.

Analogous to how some subprograms may not be coverable by any input (see example 3.1), it is also possible for rules to not be positively or negatively coverable. For example a fact can never be negatively covered, as any interpretation will always satisfy an empty body. On the other hand, rules of the form "$a \leftarrow \text{not } a.$" can never be positively covered. This is due to the fact that any interpretation that satisfies the body of this rule can not be an answer set of the program containing this rule.

Example 3.3. Consider the program $P = \{a \leftarrow \text{ not } a\}$. The interpretation $I = \emptyset$ satisfies the body or the rule ($I \models B(r)$) since $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$ both hold. However I does not satisfy the rule, because while $I \models B(r)$ holds, $I \cap H(r) \neq \emptyset$ does not hold. Therefore I is not a model for this program and no model can exists that satisfies the body of this rule.

Consequently, this also means that constraints can never be positively rule covered. This is obvious when considering that a constraint "$\leftarrow a.$" is nothing other than a shorthand notation for the rule "$x \leftarrow a, \text{ not } x.$", where x is a new atom, which matches the pattern of the rule in example 3.3 [Geb+12, p. 17]. It is also possible for a normal rule to not be positively or negatively coverable due to the program surrounding it. For example, if an atom that is never defined in the program and not part of the input alphabet occurs positively in the body of a rule, this rule can never be positively covered. Note however that any rule has to always be either positively or negatively covered.

Finally it is easy to see that rule coverage is indeed implied by program coverage, as rule coverage only considers the coverage of each rule individually, while program coverage looks at all possible sets of rules. As an example a program with a total of four rules has 16 coverable elements for program coverage, only four of which are coverable elements for rule coverage.

3.3.2 Loop coverage

Since rule coverage only considers individual rules, it can not explore possible connections between rules or atoms. On the other hand, a concept that has been shown to be very important when computing answer sets is the concept of loops in answer set programs [LZ04]. Therefore, a matching coverage metric is also introduced to capture these positive dependencies between atoms in the program.

Definition 3.3. Let P be a program and I an input for P. A loop L of P is *positively loop covered* by I iff there is some answer set $X \in \mathrm{AS}(P \cup I)$ such that for every atom $a \in L$ there is a rule $r \in \mathrm{SuppR}(P, X)$ with $r \in \mathrm{Def}_\mathrm{P}(a)$, i.e., every atom in the loop can be derived in the same answer set. The loop L is *negatively loop covered* by I iff there is some answer set X such that for at least one atom $a \in L$ for which $\mathrm{Def}_\mathrm{P}(a) \neq \emptyset$ holds, no rule $r \in \mathrm{SuppR}(P, X)$ exists with $r \in \mathrm{Def}_\mathrm{P}(a)$.

Again, we define for program P and the input collection $\mathcal{I}$ for P the (*positive* or *negative*) *loop coverage* of $\mathcal{I}$ for P as $C_{L^+}(\mathcal{I}, P)$, $C_{L^-}(\mathcal{I}, P)$ and $C_L(\mathcal{I}, P)$.

An atom a that is contained in a loop of size two or more is always defined in the program ($|\mathrm{Def}_\mathrm{P}(a)| \geq 1$) as it needs to have at least one outgoing edge in the positive atom dependency graph. This is however not true for singleton loops. Thus it is possible that a singleton loop $\{a\}$ is not positively or negatively coverable if $\mathrm{Def}_\mathrm{P}(a) = \emptyset$.

Example 3.4. Consider the program P

$$
\begin{aligned}
a &\leftarrow b, c \\
b &\leftarrow a, d \\
a &\leftarrow e,\ \mathrm{not}\ c
\end{aligned}
$$

with $\mathbb{I}_P = \{a, b, c, d, e\}$ and the positive atom dependency graph seen in Figure 3.1. We can achieve total rule coverage for P with the three test inputs $I_1 = \{b, c\}$, $I_2 = \{a, d\}$ and $I_3 = \{e\}$. However, while these inputs cover the two singleton loops $\{a\}$ and $\{b\}$ positively and negatively, the loop $L = \{a, b\}$ between the first two rules is only negatively covered. In order to positively cover L, a different input such as $I_4 = \{b, c, d\}$ is needed. Note that inputs I_1, I_2 and I_4 together yield total loop coverage, as the singleton loops $\{c\}$, $\{d\}$ and $\{e\}$ can not be positively or negatively loop covered.

The problem with loop coverage is that the number of loops in a program is, in the worst case, exponential in the number of rules of the program. Thus, while it is still an approximation of program coverage, i.e., total program coverage implies total loop coverage [Jan+10], it can be a rather expensive metric to compute. Therefore the next two coverage metrics are introduced to approximate loop coverage.

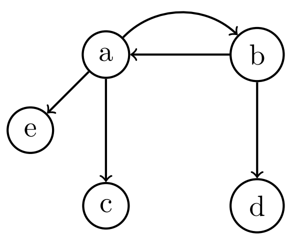

Figure 3.1: Dependency graph of the program in example 3.4

3.3.3 Definition coverage

There are two ways to think about the definition coverage metric. On one hand, it is, as mentioned above, an approximation of loop coverage as it simply looks at only the coverage of the singleton loops of a program. On the other hand one could say that definition coverage is a metric that looks at the *definable atoms* of a program P, i.e., the set of atoms $A_D = \{a \in A \mid \mathrm{Def_P}(a) \neq \emptyset\}$, as opposed to rule coverage looking at rules.

Definition 3.4. Let P be a program, I an input for P and A_D the set of definable atoms of P. An atom $a \in A_D$ is *positively definition covered* by I iff there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that there is a rule $r \in \mathrm{SuppR}(P, X)$ with $r \in \mathrm{Def_P}(a)$, i.e., the atom can be derived from the program and its input. The atom a is *negatively definition covered* by I iff there is an answer set X such that there is no rule $r \in \mathrm{SuppR}(P, X)$ for which $r \in \mathrm{Def_P}(a)$ holds.

We define $C_{D^+}(\mathcal{I}, P)$ and $C_{D^-}(\mathcal{I}, P)$ as the *positive* and *negative definition coverage* for an input collection $\mathcal{I}$ and a program P, as well as the *definition coverage* $C_D(\mathcal{I}, P)$, in the same way as with the other metrics.

It is important to note that total positive rule coverage implies total positive definition coverage but total negative rule coverage does not imply total negative definition coverage as can be seen in example 3.5. However, since definition coverage is effectively singleton loop coverage, total loop coverage implies total definition coverage.

Example 3.5. Consider the program and inputs from example 3.4. Inputs I_1, I_2 and I_3 together yield total rule coverage as well as total definition coverage since both singleton loops $\{a\}$ and $\{b\}$ are both positively and negatively covered. It is however also possible to achieve total definition coverage with just the two inputs I_1 and I_2 or I_2 and I_3. On the other hand while the two inputs $I_1 = \{b, c\}$ and $I_5 = \{a, d, e\}$ yield total rule coverage, they do not negatively definition cover the atom a and thus only yield total positive definition coverage.

Whether or not an atom is (positively or negatively) coverable always depends on the rules of the program it appears in. For example, an atom appearing as a fact in a program will always be positively definition covered and thus can not be negatively definition covered.

3.3.4 Component coverage

As seen above, definition coverage is a metric pertaining to the minimal loops of a program. Component coverage on the other hand is focused on the maximal loops of a program, i.e., its strongly connected components. As such it functions similarly to loop coverage with only a small difference in the definition of negative coverage.

Definition 3.5. Let P be a program and I an input for P. A strongly connected component C of P is *positively component covered* by I iff there is some answer set $X \in \mathrm{AS}(P \cup I)$ such that for every atom $a \in C$ there is a rule $r \in \mathrm{SuppR}(P, X)$ with $r \in \mathrm{Def}_{\mathrm{P}}(a)$, i.e., every atom in the strongly connected component can be derived in the same answer set from the program and its input. The component C is *negatively component covered* by I iff there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that for every atom $a \in C, \mathrm{Def}_{\mathrm{P}}(a) \neq \emptyset$ holds and there is no rule $r \in \mathrm{SuppR}(P, X)$ with $r \in \mathrm{Def}_{\mathrm{P}}(a)$.

The (*positive* or *negative*) *component coverage* for an input collection $\mathcal{I}$ and a program P is defined using the schema 3.1, yielding $C_{C^+}(\mathcal{I}, P)$, $C_{C^-}(\mathcal{I}, P)$ and $C_C(\mathcal{I}, P)$.

As with all the branch-like metrics, component coverage is implied by program coverage and just like definition coverage, it is an approximation of loop coverage and thus total positive loop coverage implies total positive component coverage. Because negative component coverage is defined slightly different than negative loop coverage (none of the atoms in the SCC can be supported as opposed to requiring just one atom being unsupported), negative component coverage is not implied by negative loop coverage. However, it does allow the following result: if a component C is positively (negatively) component covered, all loops that are contained in this component $L \subseteq C$ are also positively (negatively) loop covered [Jan+10, Theorem 7].

Example 3.6. Consider again the program and inputs from example 3.4. The input $I_4 = \{b, c, d\}$ yields total positive component coverage as the only coverable component is the SCC $C = \{a, b\}$. In order to achieve total negative component coverage, none of the rules of the program can be supported. Thus an input such as $I_6 = \emptyset$ works.

Notice how the input I also negatively covers the two singleton loops $\{a\}$ and $\{b\}$ that are contained in the SCC.

Now we have introduced two different coverage metrics that successfully approximate loop coverage in that they are significantly less expensive to compute while still trying to capture the essence of what loop coverage is about. Especially when used together, definition and component coverage come relatively close to the functionality of loop coverage, however for both metrics the number of coverable elements is always capped by the number of rules in the program. This is because in a propositional program, only one atom can be defined per rule and there can only be at maximum as many coverable SCCs as there are definable atoms. This means that both these metrics are significantly easier to compute than loop coverage.

4 Computing coverage metrics for propositional programs

One of the main contributions of this thesis is finding a way to implement the coverage metrics defined in the previous chapter and building a prototype tool for checking the coverage of answer set programs. Since the definitions in their current form are only for propositional programs, the implementation is also focused on that. However there is very little real world use for a coverage tool that can only work with propositional programs as almost no real world answer set program is exclusively propositional. Therefore it would be beneficial to choose an approach that can be easily extended to incorporate more complex programs. The attempt for such an extension is made in the next chapter.

4.1 General approach

The goal of this approach is to find a way to compute the different coverage metrics defined in the previous chapter using the power of ASP itself. In order to achieve this, we introduce syntactic transformations of the original program for each coverage metric.

The idea of these transformations is to add new rules to the program containing special *label atoms* for every coverage metric and every coverable element. These label atoms have to be atoms that are not contained in the original alphabet A of the program. Afterwards the program can be grounded and solved normally and the coverage can be calculated by checking which label atoms are contained in the answer sets. Since the rules that are added to the program can only produce label atoms that are not used anywhere else in the program and no existing rules are changed, the result of the transformation is still semantically equivalent to the original program. Another advantage of this approach is, that it is not necessary to compute every model of the program. Instead it is sufficient to only compute the brave consequences of the program, which is much more efficient.

The same idea of adding label atoms can also be used in order to add the test inputs to the program. This allows us to compute the coverage of all inputs contained in a test suite in the same solve call instead of using separate solve calls for each test input. However, according to the basic coverage functions schema, in order to calculate the coverage we need not only the elements covered by the test input but also the maximum coverable elements. This can easily be done in the same way by using a test input containing a choice rule of all atoms in the input alphabet $\mathbb{I}_P$ (see section 4.2). In the following, the specific transformations for each coverage metric are explained in detail.

4.2 Test input

First we will take a better look at how the test inputs of the test suite are added to the original program using the syntactic transformation. To implement this transformation, an additional language construct is required. It is called a choice rule which is shorthand for a rule containing an aggregate atom in the head. This aggregate atom is written as $s_1 \prec_1 \{a_1; \ldots; a_n\} \prec_2 s_2$, where s_1 and s_2 are in this case natural numbers, $\prec_1$ and $\prec_2$ are comparison operators such as $=, \neq, >, <, \geq, \leq$, which together form the *aggregate guards*, and the a_i are atoms. Such a rule signifies that any subset of $\{a_1, \ldots, a_n\}$ may be included in an answer set. The possible subsets can be limited by using the two aggregate guards to restrict the size of the subsets. Thus a rule of the form "$\{a_1; \ldots; a_n\} = 1.$" means that in each answer set exactly one of the a_i has to be true. For a formal definition of aggregate atoms, see Gebser et al. [Geb+15].

As mentioned in section 2.2, a test suite can contain multiple test cases and each test case contains one test input represented as a set of atoms. The transformation is done by adding a choice rule to the program, containing label atoms for each test case in a test suite. This choice rule represents "choosing" one of the test cases and additional rules will then add all the atoms contained in the chosen test input to the program.

Definition 4.1. Let P be a program and S a test suite containing test cases $T_1, \ldots, T_n$. The input of each test case T_i is a set of input atoms $\text{inp}(T_i) = \{a_{i,1}, \ldots, a_{i,m}\}$. The *input transformation* is defined as follows: Firstly, add a choice rule "$\{\_i_0; \ldots; \_i_n\} = 1.$" to program P where each $\_i_i$ is a label atom associated with the input of test case T_i. Secondly, for every test case $T_j \in S$ and for every atom $a_{j,i} \in \text{inp}(T_j)$ add a rule "$a_{j,i} \leftarrow \_i_j.$" to P where $\_i_j$ is the label atom corresponding to test case T_j.

When the test suite contains only one test case, the result of this transformation will be the same as concatenating the original program with the test input – minus the label atoms which do not change the behavior of the program. This is because the choice rule "$\{\_i_0\} = 1.$" is equivalent to the fact "$\_i_0.$" and thus all rules of the form "$a_i \leftarrow \_i_0.$" are equivalent to the fact "$a_i.$". This means that every input atom contained in the test input has to be present in all answer sets of the program in the same way as it would be when concatenating the program and the input.

On the other hand when the test suite contains multiple test cases, there can never be more than one of the input label atoms in an answer set. This means when considering each answer set individually, the above still holds. However, the results of all test cases can be calculated with just one solve call thanks to the choice rule.

Example 4.1. Consider the simple program P

$$\begin{aligned} c &\leftarrow a \\ d &\leftarrow b \end{aligned}$$

and the test suite $S = \{T_1, T_2\}$ where $\text{inp}(T_1) = \{a\}$ and $\text{inp}(T_2) = \{b\}$. When using the input transformation to add S to P, rules will be added according to the definition above and the transformed program P' will look like this:

$$\begin{aligned} c &\leftarrow a \\ d &\leftarrow b \\ \{\_i_1; \_i_2\} = 1 &\leftarrow \\ a &\leftarrow \_i_1 \\ b &\leftarrow \_i_2 \end{aligned}$$

The resulting answer sets are $\text{AS}(P') = \{\{\_i_1, a, c\}, \{\_i_2, b, d\}\}$ thus yielding the same answer sets – not counting label atoms – as adding the two inputs separately to the program, i.e., $\text{AS}(P \cup \text{inp}(T_1)) = \{\{a, c\}\}$ and $\text{AS}(P \cup \text{inp}(T_2)) = \{\{b, d\}\}$. Note that the input atoms indicate which test case was chosen in each answer set.

4.3 Rule coverage

Next we will look at the syntactic transformation used to compute rule coverage for a program P. In order to check the coverage of each rule we will introduce a new label atom $\_r_i$ for each rule $r_i \in P$. The label atom corresponds to positive coverage of that rule. Additionally, the strong negation [GL91] of the label atom (written as $-\_r_i$) is used to represent negative coverage. The strong negation of an atom a can be seen as a new atom a' with an added implicit constraint $\leftarrow a, a'$.

To realize this, we will add new rules to the program with the label atoms in the head. The atoms can then be derived whenever the corresponding rule is positively covered. The strong negation of the atom will be true whenever the rule is negatively covered. Using this approach, calculating the rule coverage is as simple as counting the label atoms contained in the answer sets of P.

As seen in section 3.3.1, a rule is positively rule covered iff the body of the rule is satisfied. Otherwise, it is negatively covered. We add the rule labels according to this definition.

Definition 4.2. Let P be a program containing rules $r_1, \ldots, r_n$. The *rule coverage transformation* works as follows: For every rule $r_i \in P$ add two new rules to P. "$\_r_i \leftarrow B(r_i)$." for positive coverage and "$-\_r_i \leftarrow$ not $\_r_i$." for negative coverage, where each $\_r_i$ is a label atom associated with the rule r_i.

The following equivalence establishes the formal correctness of this transformation.

Theorem 4.1. For a program P and an input I, the rule r_i corresponding to the label atom $\_r_i$ is positively covered by I iff the label atom $\_r_i$ is contained in an answer set $X \in \mathrm{AS}(P \cup I)$. The rule is negatively covered by I iff the label atom $-\_r_i$ is contained in an answer set $X \in \mathrm{AS}(P \cup I)$.

Proof. We will first prove the theorem for positive coverage, given a program P which has been transformed using the rule coverage transformation and an interpretation I. For the first direction, assume I positively covers a rule $r_i \in P$. Then there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that $X \models B(r_i)$. This means that the rule "$\_r_i \leftarrow B(r_i)$." is supported and thus the label atom $\_r_i$ has to be in the answer set X.

For the second direction, assume there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that $\_r_i \in X$. This is only possible if the rule "$\_r_i \leftarrow B(r_i)$." is supported, i.e., $X \models B(r_i)$ as no other way to derive the atom $\_r_i$ exists. Therefore, according to definition 3.2, the rule r_i is positively rule covered by I.

The proof for negative coverage is similar. Assume that I negatively covers the rule r_i. Then there in no answer set X such that $X \models B(r_i)$. This means that the rule "$\_r_i \leftarrow B(r_i)$." is not supported and thus the label atom $\_r_i$ can not be in the answer set X. Hence, the rule "$-\_r_i \leftarrow$ not$_r_i$." is supported and the atom $-\_r_i$ is contained in X.

On the other hand, assume there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that $-\_r_i \in X$. This is only possible if the rule "$-\_r_i \leftarrow$ not$_r_i$." is supported, i.e., $\_r_i \notin X$ as no other way to derive the atom $-\_r_i$ exists. This in turn means that the rule "$\_r_i \leftarrow B(r_i)$." can not be supported, since $\_r_i$ is not derived and $X \not\models B(r_i)$ follows. Therefore, according to definition 3.2, the rule r_i is negatively rule covered by I. □

With that, we have shown that this translation guarantees a correct calculation of the positive and negative rule coverage by simply counting the rule label atoms in the produced answer sets. In fact, it is not even necessary to compute every answer set. Computing just the brave consequences of the augmented program P for an input I will yield every rule label atom and every strong negation that has been derived in at least one answer set of P. The positive and negative rule coverage of I for P can therefore be calculated by counting these label atoms.

Example 4.2. Consider the program P and the test suite S from example 4.1. When using the rule transformation to prepare the program for a rule coverage check, four new rules will be added to P:

$$\begin{aligned} \_r_1 &\leftarrow a \\ -\_r_1 &\leftarrow \text{not } \_r_1 \\ \_r_2 &\leftarrow b \\ -\_r_2 &\leftarrow \text{not } \_r_2 \end{aligned}$$

If we then add the test suite S to the program using the input translation like before and solve the program normally the following answer sets are produced $\mathrm{AS}(P') = \{\{\_i_1, a, c, \_r_1, -\_r_2\}, \{\_i_2, b, d, \_r_2, -\_r_1\}\}$. We notice that the first answer set contains the label for the first rule and the negation for the second rule, while the second contains the label of the second rule and the negation of the first. This means the first test input positively covers the first rule and negatively covers the second rule and the second test input does the opposite. Thus, the entire test suite yields total rule coverage. It is easily verified that this is the correct result.

It is possible to achieve the same functionality by only adding one new rule per original rule of P, but this approach requires an additional solve call as well as more work outside of ASP. Considering the transformation is still linear in the size of P, our chosen approach is likely more efficient.

4.4 Definition coverage

From here on, the following transformations will each be based on a previous transformation. The transformation used to compute definition coverage in particular is based on the rule coverage transformation. This keeps each individual transformation very simple and straightforward. Because of this, the following transformations do not need to directly interact with the original program and only work off of the previously introduced label atoms. The disadvantage is that sometimes a transformation has to be made even if the corresponding coverage metric will not be computed.

The following transformation follows the same principle as the rule coverage transformation. It introduces new label atoms as well as their strong negations to represent positive and negative coverage.

As a reminder, definition coverage checks whether at least one of the defining rules of an atom is supported by a program under an interpretation. Checking whether a rule is supported is already done by the rule coverage transformation, so we can use the rule label atoms to derive new definition label atoms.

Definition 4.3. Let P be a program containing rules $r_1, \ldots, r_n$, A its alphabet and A_D the set of definable atoms of P. The *definition coverage transformation* works as follows: For every definable atom $a_i \in A_D$ and for every rule $r_j \in \mathrm{Def}_{\mathrm{P}}(a_i)$ add a rule "$\_d_i \leftarrow \_r_j$." to P where $\_r_j$ is the label atom for rule r_j and each $\_d_i$ is a new label atom associated with the atom a_i. Additionally, the rule "$-\_d_i \leftarrow \mathrm{not}\ \_d_i$." is added for every atom a_i.

It is important to note that the alphabet A of program P is that of the original program and thus does not contain any label atoms. Already introduced label atoms will not create additional rules during the definition coverage transformation and will therefore not affect the coverage calculation. As mentioned, this transformation only requires the label atoms derived from the rule coverage translation. It does not interact directly with the original program. The following theorem shows that it correctly represents the definition coverage metric.

Theorem 4.2. For a program P and an input I, the atom a_i corresponding to the label atom $\_d_i$ is positively covered by I iff the label atom $\_d_i$ is contained in an answer set $X \in \mathrm{AS}(P \cup I)$. The atom is negatively covered by I iff the atom $-\_d_i$ is contained in an answer set $X \in \mathrm{AS}(P \cup I)$.

Proof. We will prove the theorem for positive coverage only. The proof for negative coverage is similar and works in the same way as in section 4.3. First, given a program P which has been transformed using both the rule and the definition coverage transformation, an interpretation I and a definable atom $a_i \in A_D$, assume that I positively covers a_i. Then there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that there is a rule $r_j \in \mathrm{SuppR}(P, X)$ with $r_j \in \mathrm{Def}_{\mathrm{P}}(a_i)$. This means that the definition transformation added the rule "$\_d_i \leftarrow \_r_j$." to P. It also means that r_j is supported and thus positively rule covered. Therefore, according to theorem 4.1, $\_r_j \in X$ must hold, and $\_d_i$ is derived, i.e., it has to also be in the answer set X.

Second, assume there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that $\_d_i \in X$. This is only possible if $\_d_i$ has been derived from a rule added by the definition transformation of the form "$\_d_i \leftarrow \_r_j.$", meaning that $r_j \in \mathrm{Def}_{\mathrm{P}}(a_i)$ holds. For the rule to yield $\_d_i$, $\_r_j \in X$ has to hold. According to theorem 4.1, this means the rule r_j is positively rule covered, i.e., $r_j \in \mathrm{SuppR}(P, X)$. This satisfies definition 3.4 such that a_i is positively definition covered. □

Note that atoms that are not definable can not be (positively or negatively) definition covered. This is reflected in the transformation by the fact that no label atoms are added for non-definable atoms $a \notin A_D$.

Example 4.3. Consider once more the program P and test suite S from example 4.1. We use the rule coverage transformation as described in example 4.2. If we now add the definition coverage transformation there will again be four new rules added, as the program P only contains two definable atoms which are each only defined once in the program. Assuming that the atom c will be associated with the label atom $\_d_1$ and the atom d with $\_d_2$, the added rules will look like this:

$$\begin{aligned} \_d_1 &\leftarrow \_r_1 \\ -\_d_1 &\leftarrow \text{not } \_d_1 \\ \_d_2 &\leftarrow \_r_2 \\ -\_d_2 &\leftarrow \text{not } \_d_2 \end{aligned}$$

In this trivial case the definition coverage is very similar to the rule coverage, as any input that achieves total rule coverage will also achieve total definition coverage and vice versa. However, if we add a third rule to the original program such as "$d \leftarrow a.$", running both transformations will yield the program P':

$$\begin{array}{lllll} c \leftarrow a & \_r_1 \leftarrow a & -\_r_1 \leftarrow \text{not } \_r_1 & \_d_1 \leftarrow \_r_1 & -\_d_1 \leftarrow \text{not } \_d_1 \\ d \leftarrow b & \_r_2 \leftarrow b & -\_r_2 \leftarrow \text{not } \_r_2 & \_d_2 \leftarrow \_r_2 & -\_d_2 \leftarrow \text{not } \_d_2 \\ d \leftarrow a & \_r_3 \leftarrow a & -\_r_3 \leftarrow \text{not } \_r_3 & \_d_2 \leftarrow \_r_3 & \end{array}$$

In this example, we can see that the same atom can be covered by two different rules which is reflected by having two rules with the same label atom in the head. Yet, one rule per atom to derive the strong negation is enough. In this case, the test suite T_1 will yield the answer set $AS(P' \cup \mathrm{inp}(T_1)) = \{\{a, c, d, \_r_1, \_r_3, \_d_1, \_d_2, -\_r_2\}\}$ meaning it positively covers both definable atoms c and d but has a negative definition coverage of 0.

Note that in the example above, no rules are added for the atoms a and b as they are not definable. Consequently, they can not be definition covered. In the same way as with the rule coverage transformation, it is now possible to calculate the definition coverage by applying both the rule and definition coverage transformations, computing the brave consequences of the resulting program and then counting the label atoms in the resulting answer set.

4.5 Loop coverage

The loop coverage transformation is based on the previously introduced transformation for definition coverage. This is an obvious choice, as the basis of loop coverage is the same as that of definition coverage, namely checking whether specific atoms are defined in a program. The only difference with loop coverage is, that all atoms in a loop have to be defined in the same answer set. This makes the actual transformation again rather simple.

The problem with loop coverage is that, in order to calculate it, one needs to first find the loops of the program. To do that, we need to build the positive atom dependency graph of the program and then find all loops in the resulting graph. Only once all loops are found, the following transformation can be done.

Definition 4.4. Let P be a program and L the set of all loops of P. The *loop coverage transformation* works as follows: For every loop $l_i \in L$, $l_i = \{a_1, \dots, a_n\}$, that only contains definable atoms, add a rule "$\_l_i \leftarrow \_d_1, \dots, \_d_n$." to P where $\_l_i$ is a new label atom associated with the loop l_i and each $\_d_i$ is the label for atom a_i. Additionally, the rule "$-\_l_i \leftarrow \text{not } \_l_i$." is added for every loop l_i of definable atoms.

Similarly to the definition transformation, we only add rules for loops that are definable. This is, of course, only relevant for singleton loops (see section 3.3.2) but it ensures once again that no label atoms can be derived for loops that are not coverable due to them containing non-definable atoms. Also, this transformation only uses previously introduced label atoms and does not interact directly with the original program. We will again show the correctness of the transformation with the following equivalence.

Theorem 4.3. For a program P and an input I, the loop l_i corresponding to the label atom $\_l_i$ is positively covered by I iff the label atom $\_l_i$ is contained in an answer set $X \in \text{AS}(P \cup I)$. The loop is negatively covered by I iff the atom $-\_l_i$ is contained in an answer set $X \in \text{AS}(P \cup I)$.

Proof. Again, we will only prove the theorem for positive coverage, given a program P which has been transformed using the rule, definition and loop coverage transformations, an interpretation I and a loop $l_i \in L$. First, assume that I positively covers the loop $l_i = \{a_1, \dots, a_n\}$. Then there is some answer set $X \in \mathrm{AS}(P \cup I)$ for which every atom a_i contained in the loop is positively definition covered. As shown in theorem 4.2, this means that all the corresponding label atoms $\_d_1, \dots, \_d_n$ have to be contained in X. Thus the rule "$\_l_i \leftarrow \_d_1, \dots, \_d_n.$" is supported by the answer set X and the atom $\_l_i$ is contained in X.

For the other direction of the implication, assume there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that $\_l_i \in X$. This is only possible if the atom has been derived from the rule "$\_l_i \leftarrow \_d_1, \dots, \_d_n.$", meaning that $\_d_1, \dots, \_d_n \in X$ has to hold. Again using the theorem 4.2, we know that the atoms $a_1, \dots, a_n$ have to be positively definition covered in X. Therefore, all atoms contained in the loop l_i are positively definition covered in the same answer set and thus l_i is positively loop covered for P given I. □

Example 4.4. Consider the program from example 3.4. This program has a total of six loops (five singleton loops and the loop $\{a, b\}$). However, the three loops $\{c\}$, $\{d\}$ and $\{e\}$ all only contain atoms that do not appear in any rule head and are thus not definable. Because of this, no rules will be added for these loops during the loop coverage transformation. Therefore a total of six new rules is added to the program when the loop coverage transformation is used:

$$
\begin{array}{ll}
\_l_1 \leftarrow \_d_1 & -\_l_1 \leftarrow \text{not } \_l_1 \\
\_l_2 \leftarrow \_d_2 & -\_l_2 \leftarrow \text{not } \_l_3 \\
\_l_3 \leftarrow \_d_1, \_d_2 & -\_l_3 \leftarrow \text{not } \_l_3
\end{array}
$$

Once both the rule and definition coverage transformation have also been executed, it is easy to see that using the inputs I_1, I_2 and I_4 from example 3.4 will produce answer sets containing all three label atoms $\_l_1$, $\_l_2$ and $\_l_3$ and their strong negations as expected, since they yield total loop coverage for this program.

4.6 Component coverage

The definition for the component coverage metric is very similar to that of loop coverage and this is reflected also in the transformation. However, component coverage is the only metric where the absence of positive coverage does not imply negative coverage. Because of this, the rule to derive the strong negation of each component label atom is different from the previous transformations. Additionally, similar to the loop coverage transformation, it is necessary to find all the SCCs of the program first, which means building the positive atom dependency graph of the program.

Definition 4.5. Let P be a program and C the set of all strongly connected components of P. The *component coverage transformation* works as follows: For every SCC $c_i \in C$, $c_i = \{a_1, \dots, a_n\}$ that only contains definable atoms, add a rule "$\_c_i \leftarrow \_d_1, \dots, \_d_n$." to P where $\_c_i$ is a new label atom associated with the SCC c_i and each $\_d_i$ is the label for atom a_i. Additionally, the rule "$-\_c_i \leftarrow -\_d_1, \dots, -\_d_n$." where each $-\_d_i$ is the strong negation of the label for atom a_i, is added for every SCC c_i of definable atoms.

In this case the label atoms $-\_c_i$ are not derived based on the absence of the non-negated atom but rather on the absence of the relevant definition label atoms. This is in line with the definition of negative component coverage. All the points made for the loop coverage transformation also apply here as both are very similar. Meaning that rules are only added for definable strongly connected components to avoid problems with singleton loops and the transformation relies solely on label atoms. The following theorem shows the correctness of this transformation.

Theorem 4.4. For a program P and an input I, the strongly connected component c_i, corresponding to the label atoms $\_c_i$, is positively covered by I iff the label atom $\_c_i$ is contained in an answer set $X \in \mathrm{AS}(P \cup I)$. The component is negatively covered by I iff the atom $-\_c_i$ is contained in an answer set $X \in \mathrm{AS}(P \cup I)$.

Proof. We will only prove the theorem for negative coverage, as the proof for positive coverage is almost identical to that in section 4.5. First, given a program P which has been transformed using the rule, definition and component coverage transformations, an interpretation I and a strongly connected component $c_i \in C$, assume that I negatively covers the component $c_i = \{a_1, \dots, a_n\}$. Then there is some answer set $X \in \mathrm{AS}(P \cup I)$ in which every atom a_i contained in the loop is negatively definition covered. As shown in theorem 4.2, this means that all the corresponding label atoms $-\_d_1, \dots, -\_d_n$ have to be contained in X. Thus the rule "$-\_c_i \leftarrow -\_d_1, \dots, -\_d_n$." is supported by the answer set X and the atom $-\_c_i$ is contained in X.

For the other direction of the implication, assume there is an answer set $X \in \mathrm{AS}(P \cup I)$ such that $-\_c_i \in X$. This is only possible if the atom has been derived from the rule "$-\_c_i \leftarrow -\_d_1, \dots, -\_d_n$.", meaning that $-\_d_1, \dots, -\_d_n \in X$ has to hold. Again using the theorem 4.2, we know that the atoms $a_1, \dots, a_n$ have to be negatively definition covered in X. Therefore, all atoms contained in the strongly connected component c_i are negatively definition covered in the same answer set and thus c_i is negatively component covered for P given I. $\square$

Example 4.5. Consider again the program from example 3.4. The program has four strongly connected components (the loop $\{a, b\}$ and the singleton loops $\{c\}$, $\{d\}$ and $\{e\}$). However, as explained in example 4.4, the three singleton loops are not definable and no new rules will be added for them. Therefore, the component coverage transformation will only add two rules to the program:

$$\_c_1 \leftarrow \_d_1, \_d_2 \qquad\qquad -\_c_1 \leftarrow -\_d_1, -\_d_2$$

This reflects the fact that only the component $\{a, b\}$ can be positively or negatively covered in P. Once the rule and definition coverage transformations have also been applied the program looks as follows:

$$\begin{array}{rcr}
a \leftarrow b, c & \_r_1 \leftarrow b, c & -\_r_1 \leftarrow \text{not } \_r_1 \\
b \leftarrow a, d & \_r_2 \leftarrow a, d & -\_r_2 \leftarrow \text{not } \_r_2 \\
a \leftarrow e,\ \text{not } c & \_r_3 \leftarrow e,\ \text{not } c & -\_r_3 \leftarrow \text{not } \_r_3 \\
\_d_1 \leftarrow \_r_1 & -\_d_1 \leftarrow \text{not } \_d_1 & \_d_1 \leftarrow \_r_3 \\
\_d_2 \leftarrow \_r_2 & -\_d_2 \leftarrow \text{not } \_d_2 & \\
\_c_1 \leftarrow \_d_1, \_d_2 & -\_c_1 \leftarrow -\_d_1, -\_d_2 &
\end{array}$$

We can see that, as described in example 3.6, the inputs $I_4 = \{b, c, d\}$ and $I_6 = \emptyset$ yield total component coverage for P as they each cover one of the rules that were added using the component coverage transformation. Note that the set of inputs $\{I_1, I_2, I_4\}$ that yields total loop coverage for P (see example 3.4) does not negatively cover the SCC.

4.7 Program coverage

The final coverage metric that needs to be discussed is program coverage. As a reminder, program coverage looks at subprograms of the main program. These subprograms are again just sets of rules that need to be supported. This means we can base the calculation of program coverage on the rule coverage transformation, since that transformation is used to check if a rule is supported. The big difference for program coverage is, that it checks whether entire sets of rules are supported in the same answer set.

There are two different ways how we can approach computing program coverage with the help of the rule coverage transformation. One is closer to the transformation approach used for the other coverage metrics. The other takes the results we already get from the rule coverage transformation and uses outside resources to calculate the program coverage based on that. We will shortly present both approaches here, however we will refrain from trying to formally prove the correctness of these methods in this thesis.

For the first approach the idea is again to add new rules to the program that are used to derive specific label atoms. These atoms are then used to calculate the coverage. As with the other transformations, one new label atom is added for each element. In this case this means one label atom for each possible subprogram. Note that there is no such thing as negative program coverage. Thus there is no need to work with the strongly negated label atoms like in the other transformations.

Definition 4.6. Let P be a program and $P_S = \{P' \mid P' \subseteq P\}$ the set of all possible subprograms of P. The *program coverage transformation* works as follows: For every subprogram $P_i \in P_S$, $P_i = \{r_1, \dots, r_n\}$, add a rule to P "$\_p_i \leftarrow \_r_1, \dots, \_r_n$." where $\_p_i$ is a new label atom associated with the subprogram P_i and each $\_r_i$ is the label for the rule r_i.

The issue with this approach is, that the number of possible subprograms is exponential in the number of rules in the program. This means that the number of rules added by this transformation also grows exponentially with the length of the program. As a result, this approach is very inefficient for larger programs. We showed in example 3.1 that it is possible for some subprograms to not be coverable. In fact for most programs a majority of subprograms are not coverable. However, whether a subprogram is coverable depends on the semantics of the program and is therefore very difficult to figure out. Because of this, we can not use this knowledge to limit the amount of rules added by the program coverage transformation. Every rule for every subprogram has to be added and most will remain unused.

Example 4.6. Consider the program P with the input alphabet $\mathbb{I}_P = \{a, b, c, d\}$

$$\begin{aligned} a &\leftarrow d \\ b &\leftarrow a \\ c &\leftarrow b,\ \text{not } d \end{aligned}$$

The program has a total of $2^3 = 8$ subprograms so the program coverage transformation will add eight new rules to P:

$$
\begin{array}{ll}
\_p_1 \leftarrow & \_p_2 \leftarrow \_r_1 \\
\_p_3 \leftarrow \_r_2 & \_p_4 \leftarrow \_r_3 \\
\_p_5 \leftarrow \_r_1, \_r_2 & \_p_6 \leftarrow \_r_1, \_r_3 \\
\_p_7 \leftarrow \_r_2, \_r_3 & \_p_8 \leftarrow \_r_1, \_r_2, \_r_3
\end{array}
$$

It is clear to see that these rules represent each possible subset of P. Depending on which rules are covered by a test case, exactly one of these rules will be supported by the answer set and the label atom derived by it represents which subprogram has been covered. Even for a program this small, the blow-up introduced by this transformation is very obvious. Additionally, it is clear to see by looking at P that the subprogram p_2 can never be covered. This is because the derivation of the atom a in the first rule will always mean that the second rule is also supported. In fact, the only subprograms of P that can be covered are p_1, p_4, p_5 and p_7. Meaning that in this case, half of the rules added by the program coverage transformation will never be used.

The big advantage of this transformation approach is that it works perfectly fine alongside the transformations for the other coverage metrics. It makes it possible to compute program coverage with the same single solve call using brave consequences that is used for all the other metrics. This is not at all true for the second approach.

The other possible approach to calculate program coverage does not require a syntactic transformation beyond the rule coverage transformation. The idea behind this approach is that each answer set X of the program P has exactly one set of supporting rules $\mathrm{SuppR}(P, X)$. This set of rules corresponds to exactly one subprogram $P' \subseteq P$. Consequently, to calculate the program coverage of some input I of P, it is sufficient to count how many different sets of supporting rules are produced by the answer sets of $P \cup I$. We make use of the fact that according to definition 3.2 and theorem 4.1, a rule that is supported by an answer set X is positively rule covered in X and consequently the corresponding rule label atom is contained in X.

Definition 4.7. Let P be a program that has been transformed using the rule coverage transformation and I an input for P. The program coverage is calculated using a set $P_{covered}$. For each answer set $X \in \mathrm{AS}(P \cup I)$, $X = \{\_r_i, \ldots, \_r_j, -\_r_k, \ldots, -\_r_l, a_n, \ldots, a_m\}$ the tuple $(\_r_i, \ldots, \_r_j)$ containing all the positive rule label atoms in X represents the subprogram covered by that answer set. Thus, it is added to $P_{covered}$. At the end, $P_{covered}$ will contain all the unique subprograms covered by I.

Example 4.7. Consider again the program from example 4.6 with the input alphabet $\mathbb{I}_P$. Consider also a test suite with test cases containing the inputs $I_1 = \{d\}$, $I_2 = \{b, c\}$, $I_3 = \{a, c\}$ and $I_4 = \{a, d\}$. Each of these inputs yields exactly one answer set and each answer set corresponds to exactly one subprogram of P. Assuming that the rule coverage transformation has already been applied to P the resulting answer sets are $\text{AS}(P \cup I_1) = \{\{\_r_1, \_r_2, -\_r_3, a, b, d\}\}$, $\text{AS}(P \cup I_2) = \{\{\_r_3, -\_r_1, -\_r_2, b, c\}\}$, $\text{AS}(P \cup I_3) = \{\{\_r_2, \_r_3, -\_r_1, a, b, c\}\}$ and $\text{AS}(P \cup I_4) = \{\{\_r_1, \_r_2, -\_r_3, a, b, d\}\}$. The set $P_{covered}$ can then be built based on the rule label atoms contained in each answer set, yielding $P_{covered} = \{(\_r_1, \_r_2), (\_r_3), (\_r_2, \_r_3)\}$. Note that I_4 produces the same answer set as I_1 and therefore no new tuple is added to $P_{covered}$. Thus, this test suite covers three different subprograms out of the eight existing subprograms. As discussed in example 4.6, only four subprograms of P are coverable meaning the test suite S yields a program coverage $C_P(S, P) = 3/4$.

The clear advantage of this approach is that it is not necessary to calculate all possible subprograms. Instead, only the covered subprograms are gathered by going through all answer sets. This also guarantees that no time is spend on subprograms that are not coverable. On the other hand, the main disadvantage is, that this approach works very differently than the other coverage metrics. While the coverage transformations only require the computation of the brave consequences, this method needs to look at each answer set individually. Therefore it does not integrate well into the process of calculating coverage. This means an additional solve call with different options is necessary.

Overall, the first approach would presumable be more efficient for very small programs where the number of subprograms is small. For bigger programs the second approach should be a lot faster. Due to the scope of this work and the fact that program coverage is probably one of the less interesting coverage metrics for real world use due to its complexity, we decided to only implement the second approach and thus a proper comparison of the two methods is not possible at this time.

4.8 Implementation details

In this section we will take a brief look at an implementation[1] of the above transformations. As part of this thesis, a prototype tool for checking coverage for answer set programs was built that applies the defined semantic transformations and uses the power of ASP to calculate all five coverage metrics for any propositional nor-

[1] `https://github.com/ElectroChicken/Clincover`

mal logic program. The implementation is done using the integrated ASP system *clingo* [Geb+19] and its Python API [Pot21].

There are three main steps in order to implement the transformations and use them to calculate coverage. First, information on the program needs to be gathered. This includes for example collecting the atoms that appear in the program and finding which of them are definable. Second, new rules are added to the original program based on the information that has been gathered and the defined transformations. Finally, the modified program is solved using the clingo solver and the coverage is calculated based on the results of the solving process.

4.8.1 Information gathering

The information gathering is done using the *abstract syntax tree* (AST) of the program. This is a representation of a non-ground program that can be generated using the clingo.ast module of the clingo Python API. It contains all the necessary information about the rules in the program and the atoms in the head and the body of each rule.

The `parse_files` function is used to parse the original program statement by statement and pass each statement as an abstract syntax tree to the `gather_info` function. This function will analyze each AST node and save all relevant info in an appropriate data structure. The main one is a dictionary that uses atom names and arity as the keys and saves the location of the atom in the original program, a unique label associated with that atom and the labels of all the rules that define the atom in the values. The latter two make it easy to later construct the rules for the definition coverage transformation, the location is only used to allow for a better output. Additionally, the AST nodes containing an entire rule are saved in a list. This list is later used during the rule coverage transformation for easy access to the bodies of each rule. It is also used to build the positive atom dependency graph of the program and for printing results.

Example 4.8. Consider the simple program P:

$$\begin{aligned} c &\leftarrow a \\ d &\leftarrow b \end{aligned}$$

A simplified abstract syntax tree of the first rule can be seen in figure 4.1. The atom dictionary for this program looks like this after information gathering:

```
  atoms = {(c, 0): [[0], [loc], [0]], (d, 0): [[1], [loc], [1]], (a, 0): [[2],
[loc], []], (b, 0): [[3], [loc], []]}
```

Where the first value is the atom label, the second is the location of the atom in the program (here abbreviated by the keyword "loc") and the third value is a list of the labels of the rules it is defined in. Atoms a and b have an empty list as last value since they are not defined in this program.

It is obvious even from this small example that the abstract syntax tree can quickly become rather complicated. Different language constructs also have differently structured AST nodes. Thus some work has to be done to make sure that the correct information is saved for every possible type of AST node.

What is also done in the `gather_info` function is removing `#show` statements from the original program. This is necessary because show statements can interfere with the calculation of coverage. Since they have no impact on the functionality of the program, removing them is not problematic. Since every AST node of the original program is analyzed in the `gather_info` function anyways, it is easy to take care of the removal here. Each show statement is replaced by a simple `#true` statement that has no impact on the program whatsoever.

Since the information gathered in this function is relevant for each coverage metric, this function is always executed, no matter which metric will be computed. Another part of the information gathering is building the positive atom dependency graph of the program and finding all its loops and strongly connected components. This is done by going through every rule of the program and adding nodes and edges one by one to the graph using the networkX package [HSS08]. Then, the `strongly_connected_components` method of networkX is used to find all the SCCs in the constructed graph. Finally, the remaining loops that are not SCCs are found by going through every subset of every SCC and checking, whether each set is strongly connected or not. Building the dependency graph this way is not very efficient but it only needs to be done if loop or component coverage will be computed and can be skipped otherwise.

4.8.2 Adding rules

There are multiple ways how one can add new rules to an non-ground program using the clingo Python API. In this case, the rules are built using functions from the AST module of clingo and the `ProgramBuilder` class is used to add the rules to the non-ground program.

There is a separate function for each transformation. That way it is possible to only perform the transformations that are required. Each of these functions works very similarly. Thanks to the work done during information gathering, data structures

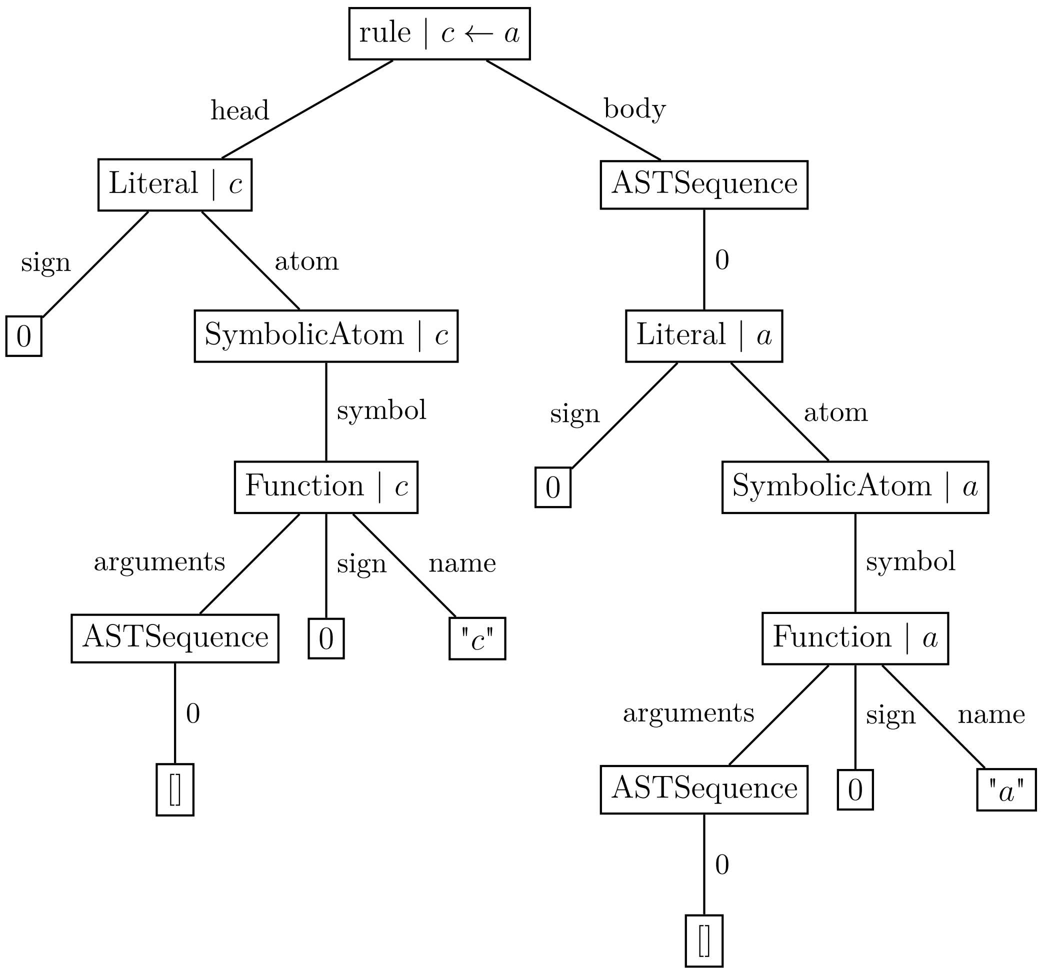

Figure 4.1: Simplified abstract syntax tree of the first rule of the program in example 4.8

already exist that contain everything needed for each transformation. All that is left to do is go through the appropriate structure for each transformation, build the rules accordingly and use a program builder to add them to the program.

As an example you can see the function used for the loop coverage transformation in listing 4.1. It uses the set `self.loops`, which is a set of tuples. Each tuple represents a loop of the program and contains the keys (name and arity) of every atom in the loop. The function goes through every loop and creates a label atom and its strongly negated counterpart for each loop. Next, it builds the body for the new rule depending on the atoms contained in the loop. In line 95, a check is implemented to see if each atom in the loop is definable in the program. If this is not the case, no rule will be added to the program. At the end, the two new rules are added to the program using the program builder – one for positive and one for negative coverage.

The only transformation that is a little different is the input transformation. The `parse_files` function is used to go through the input files statement by statement and each statement is passed to the `add_input` method. There, a new rule is created by taking the head of the original fact in the input file and adding the label atom of the current test case as the body. This corresponds to step two of the input transformation. Once that is done for every input file, the choice rule can easily be added by building a string of all the existing input label atoms.

Since we only implemented the second approach to calculating program coverage discussed in section 4.7, no new rules have to be added for this metric.

4.8.3 Calculating coverage

When all the necessary rules have been added to the program, we can work on actually calculating the code coverage. This is done in two steps. The first is to calculate how many elements are covered by the inputs, i.e., $\text{covered}_X(I, P)$ and the second is to check the maximum coverage, i.e., $\text{covered}_X(2^{\mathbb{I}_P}, P)$.

For the first step, the program has to be ground and we have to enable brave consequences by setting clingos `enum_mode` parameter to "brave". It is also important to disable any optimization by setting the `opt_mode` parameter to "ignore", as these could change the output and therefore falsify the results. Then, the ground program can be solved and all the label atoms – identified by starting with either an underscore or "-_" for the strong negation – contained in the last model are saved in the `atoms` set. We are only interested in the last model as this is the one containing the full brave consequences (as per the definition of the brave enumeration mode of clingo [GKS09]) and we can drop all the non-label atoms as those are not required to calculate the coverage. From here we build sets representing the positive coverage of each metric.

```
def add_loops(self):
    with ProgramBuilder(self.ctl) as bld:
        for idx, loop in enumerate(self.loops):
            body = []
            pos = ast.Position('<string>', 1, 1)
            loc = ast.Location(pos, pos)
            fun = ast.Function(loc, '_l{}'.format(idx), [],
                False)
            fun2 = ast.Function(loc, '-_l{}'.format(idx), [],
                False)
            atm = ast.SymbolicAtom(fun)
            atm2 = ast.SymbolicAtom(fun2)
            head = ast.Literal(loc, ast.Sign.NoSign, atm)
            head2 = ast.Literal(loc, ast.Sign.NoSign, atm2)
            for key in loop:
                if self.atoms[key][2]:
                    fun3 = ast.Function(loc, '_d{}'.format(self.
                        atoms[key][0][0]), [], False)
                    atm3 = ast.SymbolicAtom(fun3)
                    lit = ast.Literal(loc, ast.Sign.NoSign, atm3
                        )
                    body.append(lit)
            if body:
                bld.add(ast.Rule(loc, head, body))
                bld.add(ast.Rule(loc, head2, [ast.Literal(loc,
                    ast.Sign.Negation, atm)]))
```

Listing 4.1: Function implementing the loop coverage transformation (taken from CoverageCheck_final_final.py)

They contain all the labels of the elements that are positively covered. For example, the set representing positive rule coverage will contain all the $\_r_i$ labels that were in the brave consequences. In the same way, sets representing the negative coverage are built using the strongly negated label atoms.

The second step works largely the same. The difference being that instead of the input files from the test cases, a choice rule containing all possible input atoms is added to the program. This means, if the input alphabet of the program P is $\mathbb{I}_P = \{a_1, \ldots, a_n\}$, then the rule "$\{a_1, \ldots, a_n\}$." is added to the program instead of the input transformation described in section 4.2. This will allow us to compute every possible model that can be produced by any input of P. By gathering the coverage in the same way as in the first step, we receive the maximal positive and negative coverage for all metrics.

Now all that is left is to calculate the length of the set obtained in the first step divided by the length of the set obtained in the second step to get the coverage $C_X(I, P)$. On top of that the tool can print additional information such as the location in the original program of all uncovered elements.

5 Coverage for further program classes

All the coverage metrics as well as the prototype tool presented up to this point are only made for propositional programs. This is a simplification that made defining these a lot easier. Unfortunately, this approach is not very practical, as propositional programs are extremely rare in a real world setting. Many more complex language constructs exist in answer set programming that are used regularly. Some examples are variables, choice rules and aggregates, as well as conditional literals. A coverage tool that has real value for ASP developers would need to work properly for all possible answer set programs. In order to achieve this, the definitions would have to be adjusted to account for the additional language constructs. Unfortunately, this theoretic work would go beyond the scope of this thesis.

The goal of this chapter is thus not to completely redefine all the coverage metrics but rather to explore some ideas of how coverage might be handled for further program classes. We will also see how the practical approach of using a syntactic transformation, that was introduced in chapter 4, might still be used and requires very few adjustments to function properly. This is a testament to how simple yet effective this approach is.

5.1 Language constructs in ASP

In this section we will go over most of the language constructs that exist in ASP and that can be computed by the clingo solver. For each we will try to establish an idea of how it should be handled during the computation of each coverage metric. We will only briefly introduce the different constructs here. For detailed definitions refer to [Geb+19; Geb+15]

Variables

The most important extension to propositional programs is the introduction of variables. At the same time, this is also the most problematic extension for the current approach to computing coverage. This is due to the potentially infinite blowup in unique atoms that can exist inside of a program. For example, the propositional rule

"$a \leftarrow b.$" can only produce the single atom a depending on whether b is true. The rule "$a(X) \leftarrow b(X).$" on the other hand can produce as many unique atoms as there are possible values for the variable X. This causes multiple problems.

The first problem is with the definition coverage metric. Depending on the domain of the variable, the above example rule can expand into a potentially infinite amount of unique *ground rules*, i.e., rules where the variables are instantiated with values. Each of these ground rules has a different unique *ground atom* in the head. This means that according to the current definition, a potentially infinite amount of atoms has to be checked for definition coverage. This is obviously neither feasible nor desirable. The proposed solution would be to only check coverage for each unique atom signature instead of each unique ground atom. This means instead of checking coverage for $a(1)$, $a(2)$ etc., the coverage will only be checked for $a/1$. This way we go back to only having one atom to check. The same difficulty exists with loop and component coverage, as they also rely on checking for defined atoms, but the same solution also applies there.

The other problem that is introduced with variables concerns the computation of the maximal coverage for any of the metrics. The maximal coverage for elements X in program P, i.e., $\text{covered}_X(2^{\mathbb{I}_P}, P)$, is computed by using the exhaustive test suite ε_P as input for the program P. However, such is only possible if the exhaustive test suite, or rather $\text{inp}(\varepsilon_P)$, is finite. This can already be violated by the simple example above. Assuming that the domain for X is the natural numbers $\mathbb{N}$, $\text{inp}(\varepsilon_P)$ is infinite and thus the maximal coverage for a program containing only this one rule can not be computed according to the current definition. A remedy similar to what was used for definition coverage is not possible here as just choosing one variable instantiation as input for each atom will in most cases not yield the maximal coverage. One way this problem could be solved is by specifying a finite domain for each variable in the program. Although depending on the program this might not fulfill the specification.

The only easy fix to this problem is to redefine the basic coverage function schema so that coverage is not calculated relative to the maximal coverable elements but rather the total existing elements. Thus the *basic coverage function schema for programs with variables* will be defined for a class X of elements, the program P and some input I as

$$CV_X(I, P) = \frac{\text{covered}_X(I, P)}{\text{number of elements of } X \text{ in } P} \tag{5.1}$$

The obvious disadvantage here is that in most cases reaching 100% coverage will be impossible due to some elements not being coverable. The users will then have to discern themselves if elements are not covered because the test case is not optimal or because it is not possible to cover the element.

The rule coverage metric is not influenced by the existence of variables as using the suggested rule transformation to check if the body of a rule is supported by an answer set works the same whether there are variables or not. Due to this, program coverage which relies on the rule coverage transformation is also not affected.

Constraints

Constraints were already discussed shortly in section 3.3.1 as a type of rule that can never be positively covered. In order to solve this, Janhunen et al. introduce an additional coverage metric called constraint coverage in [Jan+11]. It involves removing the constraint from the program prior to computing the coverage to allow models that satisfy the body of the constraint. This can be easily implemented using a similar approach to the rule coverage transformation. However, since the constraint needs to be removed in the process, this interferes with the functionality of the original program in a way that none of the other coverage metrics do. Because of this we chose to not implement this approach at this time.

Disjunctions

Disjunctions can appear in the head of a rule by separating multiple atoms by a semicolon. They signify that, if the body of the rule is true, exactly one of the atoms in the disjunction will be added to the answer set. The simple program "$a; b.$" has therefore two answer sets with one atom each.

Since disjunctions only appear in the heads of rules, they do not influence rule and program coverage at all. Their introduction does however require some definition adjustments. In propositional normal programs, the head of a rule only contains a single atom. This changes with the introduction of disjunctions. Therefore the definition of the set of defining rules of an atom a has to be adjusted to $\mathrm{Def}_{\mathrm{P}}(a) = \{r \in P \mid a \in H(r)\}$. This means that every atom contained in a disjunction is defined there. The definition of the positive dependency graph does not need to be adjusted in this case. Thus, this adjustment should be enough to make sure that all five coverage metrics work without a problem for disjunctive programs.

Note that this way of handling disjunctions means that a test case may positively definition cover an atom a even though a is not contained in any of its answer sets. This behavior seems counter-intuitive but it makes sense considering the definition of definition coverage has nothing to do with whether an atom is contained in an answer set. Also, even though it ended up not creating an answer set, the program was solved once with the assumption that a is true.

Example 5.1. Consider the program P

$$\begin{aligned} a \;;\; b &\leftarrow c \\ &\leftarrow a \end{aligned}$$

with the input $I = \{c\}$. The only resulting answer set is $\mathrm{AS}(P \cup I) = \{\{c, b\}\}$. Because the first rule is supported, both atoms a and b are positively but not negatively definition covered however atom a is not contained in any answer set.

Head aggregates

Aggregate atoms were already shortly introduced in section 4.2 as a part of so called choice rules. These atoms can appear both in the head and the body of a rule. When they are in the head of a rule, we run into similar problems as with disjunctions, meaning they cause problems for definition, loop and component coverage but change nothing for rule and program coverage. They also have the added difficulty that they are much more flexible than disjunctions, which makes coming up with a uniform solution problematic. For example the simple disjunctive program "$a \;;\; b.$" always has two answer sets. If we replace the disjunction with an aggregate atom while keeping the atoms a and b, we have now the possibility to create anywhere from zero to four distinct answer sets depending on how the aggregate atom is constructed. "$\{a \;;\; b\}.$" has four answer sets while "$\{a \;;\; b\} = 3.$" is unsatisfiable. The difference between the two has nothing to do with the atoms contained in the rule head. On top of this, different functions can be applied to aggregate atoms such as `#count`, `#sum`, `#max` or `#min`.

For the sake of keeping the definition coverage metric relatively simple and straightforward, our suggested approach is the same as with disjunctions. Any atom contained in the head aggregate is defined in that rule. Regardless of any aggregate guards or functions that may be present. This simple solution obviously causes even more weird interactions than it does for disjunctions, but it is mostly in line with the idea behind the coverage metric. Also, any solution that would take all possibilities into account would end up being extremely complicated.

Another problem lies in the definition of the positive atom dependency graph. Its definition states, that an edge exists from every atom in the head of a rule to every atom in its positive body. This implies for a rule with a head aggregate, edges should go from the aggregate atom to the body atoms. However we are more interested in the atoms contained in the aggregate than the aggregate itself. Thus, we suggest only looking at the atoms inside the aggregate when building the atom dependency graph.

This is also in line with the proposed approach for head aggregates and definition coverage.

Example 5.2. Consider the program P

$$\{a; b; c\} \leq 2 \leftarrow d$$

The positive atom dependency graph for P would be $G = (\{a, b, c, d\}, \{(a, d), (b, d), (c, d)\}$ and all three atoms a, b and c are defined in the rule.

Body aggregates

Body aggregates take the same form as head aggregates but since they appear in the body of a rule they serve a different purpose and have to be handled differently with respect to code coverage. They have no influence on definition coverage and also rule and program coverage are once again unaffected. The problem here is again with the dependency graph. As with head aggregates, we are interested in the atoms contained in the body aggregate when calculating coverage. Thus the solution is also the same as with head aggregates: Edges in the positive atom dependency graph should go to all the positive atoms contained in a positive body aggregate instead of the aggregate atom.

Like all the other adjustments proposed in this chapter, this is a simplification that we chose as it keeps mostly in line with the idea of the coverage metrics and it is easily integrated into our approach of computing coverage. Whether it is the best – or even formally correct – in all situations requires proper testing and verification.

Conditional literals

Conditional literals are of the form $l_0 : l_1, \ldots, l_n$ where all the l_i are literals. The ":" can be seen like the mathematical set notation "|" or an implication "$\leftarrow$" and $l_1, \ldots, l_n$ is called the condition. For example the rule "$a \leftarrow b : c$." can be interpreted as the propositional formula $(c \rightarrow b) \rightarrow a$ meaning that a is derived if either c is false or both b and c are true. These literals can appear both in the head and the body of a rule as well as inside of aggregates.

As such these literals have an impact on the computation of definition coverage as well as the construction of the dependency graph and thus the loop and component coverage metrics. Multiple ways of defining the positive atom dependency graph for programs with conditional literals exist [FLL06; FL22]. We would adopt the graph $G^{sp}(T)$ defined by Fandinno and Lifschitz [FL22]. It states that for a rule $B \rightarrow H$

edges from every strictly positive atom in H to every atom that has at least one strictly positive occurrence in B will be added to the graph. Strictly positive meaning that the atom does not belong to the antecedent of any implication in the formula. Simply speaking, this means that an atom appearing in the condition of a conditional literal will never add a new edge to the graph.

Example 5.3. Consider the rule r:

$$a(X) : b(X, Y) \leftarrow c(X), d(Y) : e(Y)$$

The dependency graph for this rule has five nodes and two edges: $G^{sp}(r) = (\{a/1, b/2, c/1, d/1, e/1\}, \{(a/1, c/1), (a/1, d/1)\}$. Atoms b and e do not contribute any edges as they are not strictly positive.

By using this definition for the dependency graph when computing loop and component coverage, conditional literals will not cause any problems.

As for definition coverage the same notion of strictly positive can be used by saying an atom is defined in a rule if there is a strictly positive occurrence of the atom in the head of the rule. This means in the rule from example 5.3, only the atom $a/1$ is defined.

With these small definition changes, all coverage metrics can be computed even for programs containing conditional literals in the heads or bodies of rules. The suggested changes also work well for the previously described notions of (head and body) aggregates, and disjunctions as they are only slightly more strict by requiring strict positivity. This has the added benefit of correctly handling negated atoms in the heads of rules. It is also easily shown that the changed definition for the positive atom dependency graph is equivalent to the original definition when applied to propositional normal programs. Consider the form of rules of such programs shown in section 2.1. The head consists of the single strictly positive atom a and the body of the strictly positive atoms $b_1, \ldots, b_m$ and the negated atoms $c_1, \ldots, c_n$ which are thus not strictly positive. The resulting graph is therefore identical no matter which definition is used.

Intervals and pools

Intervals are a shorthand notation used by clingo to represent integers between two bounds. For example the fact "$a(1..4)$." is shorthand for the four facts "$a(1)$.", "$a(2)$.", "$a(3)$." and "$a(4)$.". As was mentioned in section 5.1, we are only interested in the signature of an atom when computing coverage. The actual value a variable takes or how many atoms with the same signature but different arguments exist in a program is not relevant. Therefore the introduction of intervals has no impact on the computation of coverage.

The same can be said about pools. These are another shorthand which, instead of defining intervals, gives a complete list of possible values. An example would be the fact "$a(1; 3; 5)$." that expands into the three facts "$a(1)$.", "$a(3)$." and "$a(5)$.". Again the same reasoning applies as with intervals and thus they do not influence the computation of coverage.

Meta statements

The clingo solver also supports many meta statements that are used to go beyond the scope of the program. They can for example suppress certain atoms from appearing in the output or allow the user to specify constants via command line options. These do not influence the calculation of the coverage metrics on a definition level. However, on the implementation level they have to be taken under consideration. Examples for this were already mentioned in section 4.8.1 and section 4.8.3.

The mentioned `#show` and optimization statements are meta statements that can have an impact on the computation of coverage. Because of this they are removed from the program prior to solving. Other meta statements such as `#const` and `#program` do not interfere with the calculation of coverage, however they are currently not supported by the coverage tool as they require additional attention to work as intended when using the tool. For example the `#const` statement allows the user to set the values of certain constants through the command line when calling clingo. This functionality is currently not implemented, as it is a usability feature that does not have any direct impact on the code coverage.

Theory solving and constraint programming

There are other extensions to simple answer set programs that are supported by clingo such as theory solving and constraint programming. These are not covered in this thesis as it would surpass the scope of this work. However, thanks to the simplicity of the label approach, computing rule coverage for such programs should be no problem. This is again because the metric only requires checking whether the body of a rule is satisfied. The label approach achieves this regardless of the actual content of the body. The program coverage metric should therefore also work as it is based on rule coverage.

Deciding how the theory atoms and constraint terms should behave when calculating definition, loop and component coverage should fall to experts in these fields and is thus left for future work. The current implementation of the coverage tool will simply ignore such terms when calculating these metrics.

5.2 Syntactic transformations for further program classes

In the previous section, four main suggestions were made on how to adjust some definitions in order to make calculating coverage for further program classes possible. The first one is to only consider non-ground atoms instead of ground atoms and the second was a change to the basic coverage function schema. These two together make handling variables possible. The third adjustment is a slight change in the definition of the defining rules of an atom a $\mathrm{Def}_{\mathrm{P}}(a)$. And finally, the definition of the positive atom dependency graph was adjusted. In this section, we want to discuss how to make these changes work with our proposed syntactic transformations and thus allow the computation of coverage for almost all types of answer set programs.

As a matter of fact, it is quite obvious that none of these changes have to do directly with one of the syntactic transformations. This means that the only things that need to be adjusted are the information gathering that is being done prior to the transformation as well as the calculation of the coverage in the end.

The information gathering has to be updated according to the changed definitions. As described in section 4.8.1, atoms are already differentiated based on their signature and not their ground representation meaning the first adjustment is already implemented. Most of the work in this case is actually taking into account every possible type of AST node to make sure that the correct information is saved at all times. Doing this will also take care of the third adjustment. Finally, the construction of the dependency graph can be easily adjusted to the new definition.

The only thing that is left is to implement the new basic coverage function schema. For this we need to remove the additional step of calculating the maximal coverage that was previously required. Instead the calculated number of covered elements is simply divided by the total number of elements in the program. This number is already being saved during the information gathering process and can thus be plugged in easily. With this we also remove one solve call from the whole process, going down to a total of just one solve call needed for the calculation of program coverage and another for all the other coverage metrics.

Because the bulk of the calculation of coverage is done by the clingo solver, these small changes are all that is needed to allow the coverage tool to work for almost all possible answer set programs. The complete definition of the different coverage metrics and transformations for programs other than propositional normal programs is however left for future work. Therefore it is not possible at this time to guarantee that all the ideas described in this chapter will behave as intended in every scenario. Further adjustments might be needed.

6 Conclusion

In this thesis, a prototype coverage testing tool for propositional normal answer set programs was developed by implementing the coverage metrics introduced by Janhunen et al. [Jan+10]. The chosen approach was to use a syntactic transformation on the program for which the coverage should be tested. This transformation then made it possible to utilize the power of ASP to calculate the covered elements in the program. To implement this approach, the ASP solver clingo was used together with its Python API. With the help of an additional transformation used to add the input to the program, the four main coverage metrics can be calculated at the same time, using just two solve calls regardless of the number of test cases. Computing program coverage requires two additional solve calls.

We then discussed how these coverage metrics and the syntactic transformations used to compute them can be extended in order to be applied to further program classes beyond just propositional normal programs. Some simple ideas were introduced and then implemented to yield a coverage testing tool capable of handling almost all possible answer set programs. The fact that this was possible with only minimal changes to the original tool is a testament to how simple yet effective the chosen approach is.

Unfortunately the scope of this bachelor thesis makes it impossible to fully cover everything this topic has to offer and much is left to be explored in future work. As such, both the coverage metrics and the transformations are only formally defined for propositional normal programs. More work is required to do the same for further program classes. Additionally, some concepts like theory solving have not been discussed at all. However based on the simple approach that makes use of the strength of answer set solvers, it should be possible to extend the prototype tool to work with all answer set programs. Also while the current tool can only check the code coverage given a program and a test suite, it should also be possible to use the syntactic transformations in order to automatically generate test suites with high code coverage.

Future work is also required in testing the effectiveness of the five coverage metrics. Janhunen et al. [Jan+11] did some work for rule and definition coverage but data for the other three metrics is missing. Therefore it is unclear how well these metrics perform and which combination of metrics is the most efficient at finding errors in the program. Some of the definitions might need adjustments to properly fulfill their purpose. For example, positive loop coverage for a loop l currently does not guarantee that the loop is actually executed. It only means that each atom in the loop is defined somewhere but the loop itself might have additional conditions and thus require a different test case to be executed.

Additional work is also necessary to determine if the suggested adjustments for further program classes are indeed the best approach. It remains to be seen if the potential problems pointed out in section 5.1 limit the functionality of the coverage metrics.

Finally, the prototype tool is only supposed to be a proof of concept. Because of this, some features of clingo such as external atoms or the use of program parts are not supported. Additionally, the tool has not been tested for efficiency and the code is not very optimized or especially user friendly. Such work is left to a time when all the coverage metrics are fully defined for all program classes.

Even though much is left to do on the topic, we managed to lay the ground work for a coverage tool for ASP. This makes checking the code coverage of test suites for answer set programs possible as well as potentially allow for coverage based test generation. This is one step into the direction of having a full unit testing tool for ASP. Such a tool will help make the development process of answer set programs more accessible and more efficient by allowing workflows that have been proven effective in many other programming paradigms.

List of Figures

Listings

Bibliography

[ABR21] Giovanni Amendola, Tobias Berei, and Francesco Ricca. “Testing in ASP: Revisited Language and Programming Environment”. In: *Logics in Artificial Intelligence - 17th European Conference, JELIA*. Ed. by Wolfgang Faber et al. Vol. 12678. Springer International Publishing, 2021, pp. 362–376. DOI: 10.1007/978-3-030-75775-5_24 (cit. on p. 2).

[AO16] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2nd ed. Cambridge University Press, 2016. DOI: 10.1017/9781316771273 (cit. on pp. 5, 10).

[BJ98] Fevzi Belli and Oliver Jack. “Declarative Paradigm of Test Coverage”. In: *Software Testing, Verification Reliability* 8.1 (1998), pp. 15–47. DOI: 10.1002/(SICI)1099-1689(199803)8:1<15::AID-STVR146>3.0.CO;2-D (cit. on p. 1).

[EGL16] Esra Erdem, Michael Gelfond, and Nicola Leone. “Applications of answer set programming”. In: *AI Magazine* 37.3 (2016), pp. 53–68. DOI: 10.1609/aimag.v37i3.2678 (cit. on p. 1).

[FL22] Jorge Fandinno and Vladimir Lifschitz. “Positive Dependency Graphs Revisited”. In: *Theory and Practice of Logic Programming* (2022), pp. 1–10. DOI: 10.1017/S1471068422000333 (cit. on p. 41).

[FLL06] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. “A generalization of the Lin-Zhao theorem”. In: *Annals of Mathematics and Artificial Intelligence* 47.1-2 (2006), pp. 79–101. DOI: 10.1007/s10472-006-9025-2 (cit. on p. 41).

[Fra+03] Steven Fraser et al. “Test Driven Development (TDD)”. In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Michele Marchesi and Giancarlo Succi. Vol. 2675. Berlin, Heidelberg: Springer, 2003, pp. 459–462. DOI: 10.1007/3-540-44870-5_84 (cit. on p. 1).

[Geb+12] Martin Gebser et al. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. DOI: 10.2200/S00457ED1V01Y201211AIM019 (cit. on p. 11).

[Geb+15] Martin Gebser et al. “Abstract gringo”. In: *Theory and Practice of Logic Programming* 15.4-5 (2015), pp. 449–463. DOI: 10.1017/S1471068415000150 (cit. on pp. 18, 37).

[Geb+19] Martin Gebser et al. *Potassco User Guide*. Version 2.2.0. 2019. URL: https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf (visited on 05/18/2023) (cit. on pp. 31, 37).

[GJG14] Rahul Gopinath, Carlos Jensen, and Alex Groce. “Code Coverage for Suite Evaluation by Developers”. In: *Proceedings of the 36th International Conference on Software Engineering*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. Association for Computing Machinery, 2014, pp. 72–82. DOI: 10.1145/2568225.2568278 (cit. on p. 1).

[GKS09] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. “The Conflict-Driven Answer Set Solver clasp: Progress Report”. In: *Logic Programming and Nonmonotonic Reasoning, 10th International Conference LPNMR*. Ed. by Esra Erdem, Fangzhen Lin, and Torsten Schaub. Vol. 5753. Berlin, Heidelberg: Springer, 2009, pp. 509–514. DOI: 10.1007/978-3-642-04238-6_50 (cit. on p. 34).

[GL88] Michael Gelfond and Vladimir Lifschitz. “The Stable Model Semantics for Logic Programming”. In: *Proceedings of the Fifth International Logic Programming Conference and Symposium*. Ed. by Robert Kowalski, Bowen, and Kenneth. MIT Press, 1988, pp. 1070–1080. URL: http://www.cs.utexas.edu/users/ai-lab?gel88 (cit. on p. 3).

[GL91] Michael Gelfond and Vladimir Lifschitz. “Classical Negation in Logic Programs and Disjunctive Databases”. In: *New Gener. Comput.* 9.3/4 (1991), pp. 365–386. DOI: 10.1007/BF03037169 (cit. on p. 19).

[GOT17] Alexander Greßler, Johannes Oetsch, and Hans Tompits. “\mathsf Harvey : A System for Random Testing in ASP”. In: *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR*. Ed. by Marcello Balduccini and Tomi Janhunen. Vol. 10377. Springer International Publishing, 2017, pp. 229–235. DOI: 10.1007/978-3-319-61660-5_21 (cit. on p. 2).

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15 (cit. on p. 32).

[Jan+10] Tomi Janhunen et al. “On testing answer-set programs”. In: *ECAI 2010 - 19th European Conference on Artificial Intelligence*. Ed. by Helder Coelho, Rudi Studer, and Michael J. Wooldridge. Vol. 215. IOS Press, 2010, pp. 951–956. DOI: `10.3233/978-1-60750-606-5-951` (cit. on pp. 2, 8–10, 12, 14, 47).

[Jan+11] Tomi Janhunen et al. “Random vs. Structure-Based Testing of Answer-Set Programs: An Experimental Comparison”. In: *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR*. Ed. by James P. Delgrande and Wolfgang Faber. Vol. 6645. Berlin, Heidelberg: Springer, 2011, pp. 242–247. DOI: `10.1007/978-3-642-20895-9_26` (cit. on pp. 39, 47).

[LGL10] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. “On Testing Constraint Programs”. In: *Principles and Practice of Constraint Programming – 16th International Conference CP*. Ed. by David Cohen. Berlin, Heidelberg: Springer, 2010, pp. 330–344. DOI: `10.1007/978-3-642-15396-9_28` (cit. on p. 1).

[Lif19] Vladimir Lifschitz. *Answer Set Programming*. 1st ed. Springer, 2019. DOI: `10.1007/978-3-030-24658-7` (cit. on p. 1).

[LZ04] Fangzhen Lin and Yuting Zhao. “ASSAT: Computing answer sets of a logic program by SAT solvers”. In: *Artificial Intelligence* 157.1-2 (2004), pp. 115–137. DOI: `10.1016/j.artint.2004.04.004` (cit. on p. 12).

[MSB12] Glenford J Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd ed. Wiley Online Library, 2012. DOI: `10.1002/9781119202486` (cit. on p. 7).

[MT99] Victor W. Marek and Miroslaw Truszczynski. “Stable Models and an Alternative Logic Programming Paradigm”. In: *The Logic Programming Paradigm - A 25-Year Perspective*. Ed. by Krzysztof R. Apt et al. Springer, 1999, pp. 375–398. DOI: `10.1007/978-3-642-60085-2_17` (cit. on p. 1).

[Nie99] Ilkka Niemelä. “Logic programs with stable model semantics as a constraint programming paradigm”. In: *Annals of Mathematics and Artificial Intelligence* 25.3 (1999), pp. 241–273. DOI: `10.1023/A:1018930122475` (cit. on p. 1).

[Nta88] Simeon C. Ntafos. “A comparison of some structural testing strategies”. In: *IEEE Transactions on Software Engineering* 14.6 (1988), pp. 868–874. DOI: `10.1109/32.6165` (cit. on p. 8).

[Oet22] Johannes Oetsch. “Testing for ASP-ASP for Testing”. PhD thesis. Technische Universität Wien, 2022. DOI: `10.34726/hss.2022.102508` (cit. on p. 2).

[Pot21] Potsdam Answer Set Solving Collection Potassco. *clingo Python-API reference.* Version 5.5. 2021. URL: `https://potassco.org/clingo/python-api/5.5/clingo/` (visited on 05/18/2023) (cit. on p. 31).

[Sta22] International Organization for Standardization. *Software and systems engineering — Software testing — Part 1: General concepts.* standard ISO / IEC / IEEE 29119-1:2022. Vernier, Geneva, Switzerland: International Organization for Standardization, 2022. URL: `https://www.iso.org/standard/81291.html` (cit. on p. 6).

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Zuhilfenahme anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen sind als solche kenntlich gemacht.

Die „Richtlinie zur Sicherung guter wissenschaftlicher Praxis für Studierende an der Universität Potsdam (Plagiatsrichtlinie) - Vom 20. Oktober 2010“ habe ich zur Kenntnis genommen.

Berlin, den 25. Mai 2023

Jakob Westphal