

Testing for ASP—ASP for Testing

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Johannes Oetsch, Bakk.techn.

Matrikelnummer 00025631

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: A.o.Univ.Prof. Dr.techn. Hans Tompits

Diese Dissertation haben begutachtet:

Dr. Marina De Vos

Prof. Dr. Francesco Ricca

Wien, 22. September 2021

Johannes Oetsch

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Johannes Oetsch, Bakk.techn.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. September 2021

Johannes Oetsch

Acknowledgements

First and foremost, I would like to thank Hans Tompits who supervised this thesis and served as a mentor throughout the years. In fact, working together on numerous papers and projects since 2006 deeply shaped my understanding of scientific research.

I also want to thank the external reviewers of this thesis, Marina De Vos and Francesco Ricca, for the time and effort they spent to evaluate this work.

Most results of this thesis stem from of a project titled “Methods and Methodologies for Developing Answer-Set Programs” that was funded by the Austrian Science Fond (FWF) under grant P21698. In this context, thanks are also due to Jörg Pührer. He was an excellent colleague and collaborator in this project with whom I discussed and developed many ideas that lead to numerous successful publications.

I am furthermore grateful to several researchers who helped in my academic development not only by co-authoring various publications but also by investing a lot of their time by serving as a host for fruitful research visits. In particular, I would like to thank Tomi Janhunen and Ilkka Niemelä from Aalto University in Helsinki, Esra Erdem from Sabanci University in Istanbul, Marina De Vos from the University of Bath, Nicola Leone from the University of Calabria in Rende, and Katsumi Inoue from the National Institute of Informatics in Tokyo.

In the course of the years, I had the opportunity to work together with quite a number of people on papers and articles that contain ideas which are further investigated in this thesis. In particular, this includes Martin Brain, Doga Gizem Kisa, Christian Kloimüllner, Michael Prischink, Martin Schwengerer, Martina Seidl, Stefan Woltran, Cemal Yilmaz, Patrick Zwickl, and Alexander Greßler. I thank all of them for being part of this interesting journey. I also want to express my gratitude towards Richard Kuhn for providing me with helpful information on the particular topic of event-sequence testing.

It has been a pleasure to be affiliated with the Knowledge-Based Systems group in Vienna, led by Thomas Eiter. There, I am especially thankful for all the administrative help from Eva Nedoma and the technical support by Matthias Schlögel throughout the years.

Last, I want to thank my parents Elke and Hans Oetsch. Without all the love and support they gave, doing this would not have been possible.

Kurzfassung

Unter *Answer-Set Programming* (ASP) versteht man einen logikbasierten Ansatz zum deklarativen Problemlösen. Anders als beim imperativen Programmieren werden Eigenschaften von Problemlösungen mit Regeln beschrieben und Problemlösungen (die “Answer Sets”) von ASP-Solvern automatisch berechnet. Der Erfolg von ASP spiegelt sich in einer stetig wachsenden Zahl von Anwendungen in verschiedenen Domänen, wie Planung, Diagnose, Systembiologie, Konfiguration oder Sprachverarbeitung, wieder. Bislang richtete sich die Forschung primär auf Grundlagen, Solvetechnologie und Anwendungen und weniger auf Fragen zur systematischen Programmentwicklung. Ein wesentlicher Aspekt bei jeder Form von Softwareentwicklung ist *systematisches Testen*. Obwohl die deklarative Natur von ASP den Bedarf an Testmethoden reduziert ist der Modellierungsprozess dennoch anfällig für Fehler und adäquate Testmethoden sind ebenso erforderlich. Die Entwicklung von Methoden zum *systematischen Testen von ASP Programmen* ist das vorrangige Ziel dieser Arbeit. Wir entwickeln Methoden und Methodologien zum Testen und evaluieren die vorgeschlagenen Ansätze. Viele der betrachteten Methoden können selbst mittels ASP umgesetzt werden. Über Testen für ASP hinausgehend betrachten wir auch ASP-basierte Methoden zum allgemeinen Softwaretesten. Im Speziellen verwenden wir ASP zum *Testen von Ereignissequenzen*, welche elegant in ASP modelliert werden können.

Testen ist das Ausführen von Software mit der Absicht sie zum Scheitern zu bringen, und die Literatur ist umfangreich was imperative Sprachen betrifft. Beim *Zufallsbasierten Testen* werden beispielsweise Testeingaben einer uniformen Verteilung über dem Eingaberaum entnommen. Beim *Strukturbasierten Testen* richtet sich die Auswahlstrategie auf Eingaben, die den Kontrollfluss auf definierte Programmelemente richten. Ein anderer erfolgreicher Ansatz ist *Bounded-Model-Checking*, ein leichtgewichtiger Verifikationsansatz bei dem getestet wird, ob eine gegebene Eigenschaft mit einer beschränkten Anzahl von Ausführungsschritten verletzt werden kann. Bei *Mutationsanalyse* geht es darum, eine Sammlung von Testeingaben daraufhin zu testen, ob diese ein Programm von mutierten Versionen unterscheiden kann, die durch Einfügen kleiner Fehler erzeugt wurden. Wir untersuchen wie sich solche bekannten traditionellen Testmethoden auf ASP übertragen lassen. Dabei müssen verschiedene Herausforderungen, wie das Fehlen eines expliziten Kontrollflusses und andere Aspekte im Zusammenhang mit der deklarativen Natur von ASP, überwunden werden.

Während das Hauptaugenmerk dieser Arbeit auf Testmethoden für ASP liegt, gehen wir darüber hinaus und betrachten auch andere Testprobleme. In vielen Anwendungen, wie Testen von Benutzerschnittstellen oder Programmen mit Nebenläufigkeiten, werden Fehler durch Ereignisse, welche in einer gewissen Reihenfolge auftreten, ausgelöst. Das erschöpfende Testen mit allen möglichen Ereignisfolgen ist im Allgemeinen aufgrund der Größe des Suchraumes nicht möglich. Ein vielversprechender Lösungsansatz basiert auf der Idee eine Stichprobe mittels eines geeigneten kombinatorischen Entwurfes zu generieren, der einen gewissen Grad an Interaktion von Ereignissen garantiert. Eine Herausforderung liegt darin, anwendungsspezifisches Wissen in eine Lösung zu integrieren. Wir demonstrieren die Vorteile von ASP, als ausdrucksstarke Sprache zur Wissensrepräsentation, gegenüber Ad-hoc-Ansätzen, um solche kombinatorischen Entwürfe zu modellieren.

Die Hauptbeiträge dieser Arbeit lassen sich wie folgt zusammenfassen:

- (i) Zunächst entwickeln wir ein Mutationsmodell für ASP, wobei wir ähnlichen Ideen aus anderen Sprachen folgen, und implementieren einen entsprechenden Mutationsgenerator. Dieses Werkzeug ermöglicht Mutationsanalyse in ASP. Darauf basierend evaluieren wir eine Small-Scope-Hypothese für ASP, die aussagt, dass die meisten Fehler durch erschöpfendes Testen mit Eingaben mit beschränkter Größe gefunden werden können. Unsere Evaluierung basiert auf einer repräsentativen Auswahl von ASP Benchmarkproblemen.
- (ii) In weiterer Folge studieren wir zufallsbasiertes Testen für ASP und stellen eine entsprechende Implementierung vor. Sowohl das Generieren von Testeingaben als auch das Bestimmen des Testausgangs werden mittels ASP realisiert. Unser Ansatz erlaubt über Testen in ASP hinaus auch das Testen von Systemen, welche strukturell komplexe Eingaben erfordern.
- (iii) Wir entwickeln Methoden zum strukturbasierten Testen von Answer-Set Programmen, d.h. wir adressieren das Testen mit Eingaben, die im Hinblick auf die Struktur eines Programms ausgewählt werden. In Analogie zu Pfad und Zweigüberdeckungstests im konventionellen Testen werden verschiedene ASP-spezifische Überdeckungsbegriffe eingeführt. Wir präsentieren Strategien zur Erzeugung von Eingaben und evaluieren unseren Ansatz.
- (iv) Um das systematische Entwickeln von Programmen zu unterstützen, führen wir eine Annotationssprache für ASP ein, um Programme mit zusätzlicher Metainformation zu erweitern. Dies ist für verschiedene Belange des Testens und der systematischen Programmentwicklung sinnvoll. Diese Sprache ist in *SeaLion*, einer Eclipse-basierten integrierten Entwicklungsumgebung für ASP, realisiert. Wir stellen zwei Entwicklungswerkzeuge vor, die auf Programmannotationen aufbauen: *ASPDoc* zur Erzeugung von HTML-Dokumentation für ASP, und *ASPUnit* zur Verwaltung von Unittests in ASP.

- (v) Schlussendlich gehen wir über das Testen für ASP hinaus und präsentieren einen ASP-basierten Ansatz zum Testen von Ereignissequenzen. Unser Ansatz erlaubt es, komplexe Testabdeckbedingungen knapp auszudrücken und Verfeinerungen und Variationen in der Problemstellung leicht zu inkorporieren.

Abstract

Answer-set programming (ASP) is a prominent approach for declarative problem solving with roots in non-monotonic reasoning, knowledge representation, and logic programming. As a problem solving paradigm, it means that properties of problem solutions are modelled using declarative rules so that an ASP solver can then be used to search for solutions which are referred to as *answer sets*. The success of ASP is witnessed by a large and indeed ever growing number of applications in various domains including planning, diagnosis, systems biology, configuration, language processing, and many more. While research so far focused more or less on theoretical foundations, solver technology, and applications, there is comparably little work on development support although this is a clear desideratum of the community. Testing is an essential part of every software development process, and no system for quality management can spare testing completely. Although it can be argued that the declarative nature of ASP reduces the need for testing to some extent, errors sneak also into ASP specifications and adequate methods for testing are required for ASP no less than for conventional imperative languages. Developing methods for *systematic testing of ASP programs* is therefore the foremost objective of this thesis. We lay down the foundations for testing answer-set programs, develop respective methods and methodologies, and evaluate the proposed approaches. As it turns out, many methods for testing ASP can be effectively realised using ASP itself. We go beyond testing for ASP and apply ASP-based methods to tackle challenging problems from the field of general software testing. In particular, we use ASP to address the problem of *event-sequence testing* which requires the generation of certain combinatorial designs that can be expressed using ASP quite conveniently.

Testing can be defined as executing software with the intent to make it fail, and there exists a vast body of literature on this subject for imperative languages. This includes *random testing* and *structure-based testing*. In random testing, test inputs are drawn from a uniform distribution over the test-input space. In structure-based testing, the objective is to cover the structure of a program by directing the control flow towards certain elements. Another successful testing approach is *bounded model checking*, which is a light-weight verification approach to check whether a defined property can be violated with a bounded number of execution steps. In *mutation analysis*, a test suite is evaluated with respect to its ability to distinguish a program under test from mutated versions of that program which are generated by injecting small errors. We investigate how such well-known methods from traditional testing can be adapted and applied for ASP. To do

this, we have to overcome challenges related to the declarative nature of ASP like the lack of an explicit control flow.

While the main focus of this thesis is on testing for ASP, we also go beyond this topic and address testing problems outside of ASP. In particular, we deal with the problem of *event-sequence testing*. In many applications, like testing of user-interfaces or multi-threaded programs, faults are triggered by events that occur in a particular order. In general, the search space of all possible event sequences will be too large for exhaustive testing. Promising solution approaches are based on sampling the search space in a more sophisticated way using dedicated combinatorial designs that guarantee a defined degree of interactions between events. A challenge here is to incorporate application specific knowledge that necessitates variations of the original problem statement. We aim at demonstrating the advantages of ASP as a highly-expressive knowledge-representation language over more ad-hoc approaches to generate the required designs.

We summarise our main contributions as follows:

- (i) To begin with, we develop a mutation model for ASP following similar ideas for other languages and implement a respective mutation generator. This tool facilitates mutation analysis in ASP which is useful for evaluations of other testing methods. Based on mutation analysis, we evaluate the small-scope hypothesis for ASP, i.e., the idea that most bugs can be found by testing exhaustively with inputs of relatively small size. Our evaluation is based on a representative set of ASP benchmark problems.
- (ii) We then introduce random testing for ASP and present a respective tool. Both test-input generation and determining test verdicts rely on ASP itself. This approach is potentially useful for testing systems that involve structurally complex test inputs outside of ASP.
- (iii) In the next step, we develop notions for structural testing of answer-set programs, i.e., we address testing based on inputs that are chosen with respect to the structure of an encoding. In particular, we introduce ASP counterparts to path and branch coverage from conventional testing. Also, we show how test inputs can be automatically generated, and experimentally evaluate the structure-based approach against simple random testing.
- (iv) Towards a systematic development methodology, we introduce an annotation language for ASP that allows to augment answer-set programs with additional meta-information that is useful for testing and systematic program development. This language is implemented within *SeaLion*, an Eclipse-based integrated-development environment for ASP. We furthermore introduce particular tools that are based on program annotations: *ASPD*oc, for generating an HTML documentation for a program, and *ASP*Unit, for running and monitoring unit tests on program blocks.

- (v) Finally, we go beyond testing for ASP and present an ASP-based approach to event-sequence testing. We propose ASP for computing relevant combinatorial structures and compare our method against related work. Our approach allows to concisely state complex coverage criteria in an *elaboration tolerant* way, i.e., small variations of a problem specification require only small modifications of the ASP representation.

Contents

Kurzfassung	vii
Abstract	xi
Contents	xv
1 Introduction	1
1.1 Testing for ASP	2
1.2 ASP for Testing	4
1.3 Contribution of the Thesis	5
1.4 Organisation of the Thesis	7
1.5 Publications	8
2 Background	11
2.1 Answer-Set Programming	11
2.2 Complexity Theory	17
3 Testing Uniform ASP Encodings	21
3.1 Basic Concepts for Testing	21
3.2 A Small-Scope Hypothesis for ASP	26
3.3 Random Testing	42
3.4 Structure-Based Testing	47
4 An Annotation Language for ASP	67
4.1 Basic Elements of the Annotation Language LANA	67
4.2 ASPDoc	76
4.3 ASPUnit	78
4.4 Related Work	80
5 ASP for Event-Sequence Testing	83
5.1 Sequence-Covering Arrays	83
5.2 Prelude: Complexity of SCA Generation	86
5.3 SCA Computation	89
5.4 Problem Elaborations	96
	xv

5.5 Related Work	103
6 Conclusion	105
Appendix A: ASP Benchmark Problems	109
Problems in P	109
Problems in NP with a Tight Encoding	123
Problems in NP with a Non-Tight Encoding	143
Bibliography	161

Introduction

Answer-set programming (ASP) [115, 120, 10, 109, 54, 69] is not only one of the currently most widely-used computational approaches for realising non-monotonic reasoning but also constitutes a viable paradigm for declarative problem solving. Indeed, the development of increasingly efficient solver technology [27, 25, 72] allowed ASP to become an important host language for computing reasoning problems from diverse areas including systems biology [82], cladistics [58, 21], planning [108, 51, 52], diagnosis [50, 123], configuration [160], multi-agent systems [11], music composition and production [60, 15, 148, 59], super optimisation [18], semantic-web reasoning [152], and natural language processing [157]—an overview and further references can be found in related articles [56, 62].

The basic idea of ASP is to encode a problem instance as a logic program such that its models, referred to as *answer sets*, provide the solutions to the given instance. So far, research in ASP can basically be classified into three categories:

- (i) theoretical foundations of ASP including language extensions,
- (ii) performance development and evaluations of ASP solvers, and
- (iii) case studies as well as applications involving ASP.

Besides these traditional research areas, methods and methodologies to support an ASP programmer are becoming an increased focus of interest [40, 41, 134].

Testing is an essential part of every software development process. No system for software quality management can spare testing completely, and there exists a vast body of literature on this subject [118, 13]. It is a particular strength of ASP that the high-level specification languages supported by state-of-the art ASP solvers allow to develop concise specifications close to the problem statement which reduces the need for debugging and testing methods to a minimum. Despite this, to err is human, and faults can and

usually do sneak in even into such high-level specifications. It hence can be argued that although there is less need for testing than in conventional imperative software development, it is still required when more sophisticated methods like correctness proofs or equivalence checks are not viable. This can be due to a lack of skills of an ASP engineer or simply because of limited time resources. Also, the ongoing development of solver languages makes testing, as a flexible light-weight verification approach, especially appealing. We conclude that adequate means for testing are therefore clearly needed for ASP as well. In fact, *systematic testing of programs*, constituting a key element in a typical software development process, has been recognised in the academic community as a crucial step in an ASP program development methodology [17]. Yet, only few dedicated methods and tools for testing answer-set programs have been introduced so far [93, 94, 64, 66, 39, 127, 83, 4].

In this thesis, we lay down the *foundations for testing answer-set programs* in more detail, develop *methods and methodologies for systematic testing* of answer-set programs and *evaluate the proposed methods*. As demonstrated in previous work [93, 94], methods for testing ASP can be effectively realised using ASP itself. Going beyond testing for ASP, we apply ASP-based methods also to tackle challenging problems from the field of software testing in general. In particular, we use ASP to address the problem of *event-sequence testing* which requires the generation of certain combinatorial designs that can be expressed using ASP quite conveniently.

1.1 Testing for ASP

Testing can be understood as a light-weight verification approach that aims at increasing the reliability of a software component by executing the code with the intent to make it fail [118]. Hence, testing is inherently incomplete and—contrary to full verification—cannot be used in general to guarantee the correctness of a program.

The prevalent mode ASP is used for declarative problem solving is by modelling properties of problem solutions by means of ASP rules. This results in so-called *uniform problem encodings* where we distinguish between a *fixed program* that specifies solutions to a problem at hand and *problem instances* that are represented as sets of facts over some input signature.

For illustration, consider the following reviewer-assignment problem, in which we deal with reviewers and papers that need to be reviewed. The objective is to find an assignment subject to the following constraints:

- (i) every paper is assigned to three reviewers,
- (ii) no reviewer gets more than four papers assigned, and
- (iii) no paper gets assigned to a reviewer who has declared a conflict of interest with that paper.

We consider the following encoding in `gringo` syntax [154]¹.

```
{ assigned(R,P) : reviewer(R) } = 3 :- paper(P).    % Condition (i)
:- { assigned(R,P) : paper(P) } > 4, reviewer(R).  % Condition (ii)
:- assigned(R,P), conflict(R,P).                  % Condition (iii)
#show assigned/2.
```

Each rule directly expresses one of the three conditions from the above. Now, a problem instance is described using the predicates `paper/1`, `reviewer/1`, and `conflict/2`, serving as *input* for the problem encoding, while the *output* is given by the answer sets, projected to the output predicate `assigned/2`, of the ASP program joined with the input. Testing a uniform problem encoding in ASP means to select different problem instances (inputs) and verify that the computed answer sets indeed correspond to the solutions that we would expect.

Different strategies for selecting test inputs have been studied and used in practice for conventional software testing. Probably the conceptually most simple one is *random testing*. As the name suggests, random testing means that test inputs are drawn from a uniform distribution over the test-input space [86]. Although this seems quite straightforward, random testing can be quite effective. Particular advantages are that it is simple and unbiased, that is, no parts of the test-input space are preferred over other ones. Random testing lends itself as a baseline for comparison with more sophisticated testing methods. In ASP, there are certain challenges to overcome. On the one hand, we need means to define the test-input space. On the other hand, generating a uniform sample is not trivial when we deal with test inputs that have a complex structure as it is often the case for testing uniform problem encodings.

A more sophisticated approach to select test inputs is *structure-based testing*, also known as *white-box testing*. Here, the goal is to “cover” different structural components of the program under test like execution paths, branches, statements, or conditions [90, 118, 124]. To obtain complete path coverage means to have enough test cases so that each possible execution path is worked through at least once. Structure-based testing seems to be a promising concept for ASP as well. However, ASP is a declarative programming paradigm and lacks any notion of explicit control flow. Thus, coverage notions from imperative programming cannot be used directly, and novel declarative concepts that reflect the structure of answer-set programs are required.

Another successful testing approach is to exhaustively test a software component with all inputs within a given bound on their size. The underlying idea is that bugs can usually be triggered using relatively small test inputs. This concept is often referred to as *the small-scope hypothesis* [92] and is the basis of prominent light-weight verification techniques such as bounded model checking [30]. In ASP, program comparisons under certain notions of equivalence can be used for exhaustive testing within a fixed scope [95, 144]. However,

¹At this point, we trust in the reader’s intuition, details about syntax and semantics follow later.

a respective version of a small-scope hypothesis needs to be formulated and evaluated in ASP as well.

An interesting concept to evaluate the quality of a collection of test-inputs with respect to its potential for fault detection is *mutation analysis* [42]. In a nutshell, the idea is to simulate a sloppy programmer and to generate a number of “mutated” versions for a program under test. A mutation is a small error that is introduced at random at some place. We then run our test inputs on the mutants and assess the number of errors that we can find. A good test suite should be able to identify a high number of faults, otherwise it should be extended or the test inputs need to be improved. While this idea seems feasible for ASP as well, we need to define and implement a suitable mutation model that considers typical modelling errors that an ASP engineer can make.

With larger programs being written for real-world applications, it is vital to support a programmer with the right tools. Few software developer would consider to realise a larger project without the help of a dedicated *integrated development environment* (IDE). Such IDEs usually incorporate features for programming convenience like code completion or syntax highlighting. Also, some allow to write simple test cases for small components, so-called *unit tests*, and provide frameworks to run and monitor them, cf., e.g., JUnit [98]. This facilitates *test-driven development* as a model to write software in small increments, each of which is preceded by a new unit test for some yet unimplemented functionality. Other features related to testing and validation include the possibility to express assertions and report if they are violated or to generate a documentation for the program based on annotated comments in the code, cf., e.g., JavaDoc [97]. Existing IDE support for ASP includes the system *Aspide* [150, 65] that also offers features for unit testing [64, 4] and *SeaLion* [135], which is based on Eclipse [49]. However, further progressing this direction of tool support, especially towards testing and validation features, is an important goal in ASP.

1.2 ASP for Testing

While the main focus of this thesis is on testing for ASP, we also make contributions to the field of *combinatorial-interaction testing*. There already exists a considerable body of literature regarding this subject [22, 163, 37, 32, 104, 84, 168, 113, 114, 34, 43, 167]. We address the specific problem of *event-sequence testing* [170, 171, 103, 169]. Testing different classes of event-driven software applications is becoming very important and this may involve generating and executing sequences of events in such applications. Common examples are user-interface testing [170], dynamic web applications [166], and testing of multi-threaded programs [87, 113]. What the mentioned challenges have in common is that the search space in terms of event sequences or thread interleavings is usually prohibitively large for exhaustive testing. However, promising solution approaches can be based on combinatorially sampling the search space in a sophisticated way based on a solid fault model as demonstrated for related testing problems [22, 32, 37, 104, 84, 113, 34]. The generation of dedicated combinatorial structures is therefore an important issue. A

particular challenge is to incorporate application specific knowledge that necessitates variations of the original problem statement into a solution approach.

ASP, as a language for knowledge representation that allows to concisely specify solutions to combinatorial search problems and, more importantly, allows to explicitly incorporate application-specific knowledge into some solution approach, has the potential for mastering aforementioned challenges in combinatorial testing by generating the needed combinatorial designs. This has been demonstrated to some extent in related work on combinatorial-interaction testing [167, 7] and event-sequence testing [9, 19]. In particular, we investigate *ASP as a method for testing even-driven software*, develop dedicated ASP-based methods, and compare them with existing approaches. Our results demonstrate the advantages of using a highly-expressive knowledge-representation language over more ad-hoc approaches in the light of application-specific knowledge and change.

1.3 Contribution of the Thesis

To achieve our envisaged goals, both theoretical research and practical work like implementations and experiments were done. In fact, development support for ASP was the objective of an FWF-funded research project on methods and methodologies for developing answer-set programs conducted at the Technische Universität Wien [134]. The goal of this project was not only to study the theoretical foundations of suitable programming support methods, including systematic testing of answer-set programs, but also to develop practical tools. Most of the results of this thesis stem from this project.

Regarding testing for ASP, our main contribution is a *methodology for testing in ASP*. This includes foundational issues like basic definitions and concepts tailored to specific testing scenarios and needs in ASP. In particular, we investigate how well-known methods from traditional testing can be defined and applied for ASP. In what follows, we summarise our main contributions in more detail.

A mutation model for ASP. We develop a mutation model for ASP following similar ideas for other languages like Java or C++, and we implement a respective mutation generator. This tool takes an ASP encoding as input and produces faulty versions of that encoding as output. This is done by injecting small errors that simulate a sloppy programmer like the omission of atoms or rules. This mutation generator facilitates mutation analysis in ASP which is needed for empirical evaluations of other testing methods, or it may serve as a testing method for itself.

A small-scope hypothesis for ASP. We evaluate the small-scope hypothesis for ASP. To this end, we follow work in traditional testing and base our evaluation on mutation analysis in order to examine the relation of input size and mutant-catching rate for a representative set of ASP benchmark problems. In fact, we show that a rather limited scope is sufficient for testing ASP encodings from the benchmarks, i.e., a high proportion of errors can be found by testing a program exhaustively with small inputs.

Our experimental evaluation enables effective methods for testing in ASP like bounded-exhaustive testing or testing methods inspired by model checking. These methods allow to verify that certain properties hold for one or all answer-sets of a program under test, respectively. We describe how to use ASP itself for realising this. Also, it is the small-scope hypothesis that gives some justification to analyse programs at the propositional level after grounding them over a small domain.

Random testing. Random testing is recognised as a simple yet effective testing approach. We discuss challenges and develop methods for random testing in ASP. In particular, we present **Harvey**, a tool to realise random testing of answer-set programs. In our approach, both test-input generation and determining test verdicts is done using ASP itself: The test-input space is defined using ASP rules and uniformity of test-input selection is achieved by using XOR sampling. This allows to go beyond simple random testing by adding further ASP constraints in the process. Our random-testing approach has the potential to serve as a basis for new testing methods beyond ASP, like random testing of systems that involve structurally-complex test inputs.

Structure-based testing. We develop notions enabling structural testing of answer-set programs, i.e., we address testing based on inputs that are chosen with respect to the structure of a given answer-set program. More specifically, we introduce different notions of coverage that measure to what extent a collection of test inputs covers certain interesting structural components of the program. In particular, we introduce metrics corresponding to path and branch coverage from conventional testing, and we study their interrelations including organising different coverage notions within a subsumption hierarchy. Also, we discuss complexity aspects of the considered notions and give strategies how test inputs that yield increasing (up to total) coverage can be automatically generated.

We furthermore devise an experimental setting where structure-based and random testing are compared on different benchmarks chosen from an ASP competition. The comparison is based on mutation analysis. Results indicate that random testing is quite ineffective for some benchmarks, while structure-based techniques catch faults with a high rate more consistently also in these cases.

An annotation language, unit testing, and program documentation. We propose to augment answer-set programs with additional meta-information formulated in a dedicated annotation language, called **LANA**. The design and implementation of **LANA** is a further contribution of this thesis. The motivation is that different testing methods necessitate the specification of additional meta information like inputs and output signatures, pre- and post-conditions, and so on. **LANA** is implemented within the Eclipse-based IDE **SeaLion** for ASP [135].

This language allows to group rules into coherent blocks and to specify language signatures, types, pre- and postconditions, as well as unit tests for such blocks similar to other programming languages. While these annotations are invisible to an ASP solver, as they

take the form of program comments, they can be interpreted by tools for documentation, testing, and verification purposes, as well as to eliminate sources of common programming errors by realising syntax checking or code completion features. To demonstrate its versatility, we introduce two such tools, viz.

- (i) ASPDoc, for generating an *HTML documentation for a program* based on the annotated information, and
- (ii) ASPUnit, for running and monitoring *unit tests* on program blocks.

LANA is also exploited in the IDE *SeaLion*. Based on this framework for unit testing, a *test-driven development methodology for ASP* is discussed.

An ASP-based approach to event-sequence testing. We finally go beyond testing for ASP and also investigate ASP for event-sequence testing. In many applications, faults are triggered by events that occur in a particular order. In fact, many bugs are caused by the interaction of only a low number of such events. Based on this assumption, *sequence covering arrays* (SCAs) have been proposed as suitable designs for event sequence testing. In practice, directly applying SCAs for testing is often impaired by additional constraints, and SCAs have to be adapted to fit application-specific needs. Modifying precomputed SCAs to account for problem variations can be problematic, if not impossible, and developing dedicated algorithms is costly. We propose ASP for computing SCAs and compare our method against related work. Our approach allows to concisely state complex coverage criteria in an *elaboration tolerant* way, i.e., small variations of a problem specification require only small modifications of the ASP representation [117]. Employing ASP for computing SCAs is further justified by new complexity results related to event-sequence testing that are established in this work.

1.4 Organisation of the Thesis

This thesis is organised as follows. In Chapter 2, we provide some background and a short introduction to ASP, and we review formal syntax and semantics of logic programs under the answer-set semantics. Also, we give a brief introduction to relevant concepts from complexity theory.

Chapter 3 is dedicated to testing methods for uniform ASP encodings. We start in Section 3.1 to formally introduce basic terminology and definitions needed for testing answer-set programs. This includes in particular the notions of test case, test suite, input and output signatures, specifications, and failing and passing of test cases. We consider both propositional programs and programs with variables and also discuss issues related to test automation like the oracle problem, i.e., how to verify that some actual output matches the expected outcome.

Then, in Section 3.2, we empirically evaluate a small-scope hypothesis for ASP. For our evaluation, we follow work in traditional testing and base our evaluation on mutation

analysis. We introduce a respective mutation model for ASP and discuss its implementation after reviewing general principles of mutation testing and mutation operations. We then show that a rather limited scope is sufficient for testing ASP encodings from a representative set of benchmark problems.

We next address random testing in Section 3.3 and describe a method to generate answer sets in a randomised fashion that thus allows to sample combinatorial search spaces in a uniform way. We implemented this method as a dedicated random-testing tool. Besides the tool’s relevance for random testing in ASP and beyond, it is the basis to evaluate more advanced testing concepts that follow later.

Finally, in Section 3.4, we address structure-based testing for ASP, i.e., testing based on inputs that are chosen with respect to the structure of a given answer-set program. We devise an experimental setting where structure-based and random testing are compared on different benchmarks chosen from an ASP competition.

Chapter 4 is dedicated to the topic of testing and of systematic development support in ASP in which we introduce the annotation language LANA for ASP. To wit, in Section 4.1, we explain the basic language features and illustrate them using a running example. In Section 4.2 we describe the basic features of ASPDoc for automatically generating a documentation document for an ASP encoding, and in Section 4.3 we do the same for ASPUnit, a tool that implements a unit-testing framework for ASP.

After having dealt with testing for ASP, we move on to explore testing challenges outside of ASP in Chapter 5. In particular, we address challenges in the field of combinatorial interaction testing. We introduce ASP encodings for computing sequence-covering arrays and analyse their computational complexity. Then, we discuss how to deal with problem elaborations and experimentally compare our method against a related approach.

The relevant related work is discussed at the end of the respective chapters. We conclude in Chapter 6, where we summarise our contributions and discuss future work.

1.5 Publications

Most results of this thesis emerged as an outcome of the Austrian Science Fund (FWF) Project “Methods and Methodologies for Developing Answer-Set Programs” (P21698)—a brief project description of it is published in the technical communications of the 26th International Conference on Logic Programming (ICLP 2010) [134]. Work on basic testing terminology and structure-based testing for ASP was done in close cooperation with Tomi Janhunen and Ilkka Niemelä and first presented at the 19th European Conference on Artificial Intelligence (ECAI 2010) [93], a subsequent paper on an empirical comparison between structure-based testing and random testing for ground answer-set programs was published in the proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011) [94].

LANA, the annotation language for answer-set programs, was presented at the 14th International Workshop on Non-Monotonic Reasoning (NMR 2012) [165]. Later, a conference

version was accepted at the 28th International Conference on Logic Programming (ICLP 2012) and was published as part of a special issue of Theory and Practice of Logic Programming [39].

A mutation model for ground answer-set programs was first introduced in the context of structure-based testing [94]. A generalisation of it and its application to evaluate a small-scope hypothesis for ASP was introduced in a paper presented at the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012) [127].

SeaLion, an IDE for ASP in which **LANA** is incorporated, was first presented at the 25th Workshop on Logic Programming (WLP 2011) [135]. An extended version of that paper was accepted for the workshop's formal post-conference proceedings [140].

Work on ASP for event-sequence testing is the result of joint work involving the Sabanci University in Istanbul, the National Institute of Informatics (NII) in Tokyo, where respective research was carried out to a large extent, and the Technische Universität Wien. Results were first presented at the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011) [57]. Later, an extended version was published in the International Journal on Advances in Software [19].

While a first randomised method for testing in ASP was already introduced at LPNMR 2011 [94], a more elaborated system for random testing in ASP based on XOR streamlining has been presented at LPNMR 2017 [83].

Other project results not covered in this thesis include work on debugging answer-set programs [141, 132, 133, 136, 138, 139], a **SeaLion** plugin for visualising answer sets [102], work on program transformations [155] and program verification [143], plagiarism detection for ASP [128, 129], model-driven development support for ASP [130, 131], as well as extensions of object-oriented languages for ASP [137].

Background

The aim of this chapter is to provide a short introduction to ASP, the basic language, and common extensions. Also, we review concepts from complexity theory that are relevant for this work.

2.1 Answer-Set Programming

Answer-set programming (ASP) [115, 120, 10, 109, 54, 69] has its roots in knowledge representation and non-monotonic reasoning. It is a logic-based declarative problem solving paradigm in which a logic program is used to describe the search space as well as the requirements that must be fulfilled by the solutions of a given problem. Thus, a computational problem is declaratively represented as a logic program whose models, appropriately referred to as “answer sets”, correspond to the solutions of that problem [77]. Answer sets are usually defined through a variant or extension of the stable-model semantics [76, 77].

Algorithms and tools for obtaining answer sets of logic programs are referred to as *answer-set solvers*. Popular and widely used solvers are DLV [55, 1], WASP [3], and the SAT inspired `clasp` [71]. Alternatives are `SModels` [121] and `CModels` [78], a solver based on translating the program to SAT. ASP solver competitions provide an additional overview of current technology [27, 25, 72].

To illustrate problem solving in ASP, we use the 3-colourability problem (3COL): Given a graph consisting of nodes and edges between nodes, we want to find an assignment of three colours to the nodes such that no nodes connected by an edge have the same colour. We use the syntax that is supported by the solver `clasp` along with the grounding tool `gringo` [85] to represent an ASP program for 3COL. Note that, at term positions, upper-case letters denote variables, while lower-case letters denote constant symbols.

```

colour(red;green;blue).
1 {asgn(N,C) : colour(C)} 1 :- node(N).
:- edge(X,Y), asgn(X,C), asgn(Y,C).

```

The first rule abbreviates three facts that state that red, green, and blue are colours, respectively. The second rule is a choice rule. Its intuitive reading is that if N is a node, then both an upper bound and a lower bound on the number of colours assigned to this node, expressed by $\text{asgn}(N, C)$, is 1. This means that each node gets assigned precisely one colour from the set of available colours defined by $\text{colour}/1$. The last rule is a constraint that forbids that there is an edge between any two nodes with the same colour. If the above program is joined with facts over $\text{edge}/2$ and $\text{node}/1$ that represent a graph G , the answer sets correspond one-to-one to the valid 3-colourings of G .

In the following, we briefly introduce the essential concepts of ASP; for an in-depth coverage, we refer to well-known textbooks on this subject [10, 69, 75].

2.1.1 Syntax and Semantics of Normal Logic Programs

To begin with, we review the original class of logic programs that was introduced by Gelfond and Lifschitz [76]. Later, we will introduce common language extensions. The language of *normal logic programs* is defined over a countable set \mathcal{U} of *atoms*, each of which can be assigned a truth value. An atom can be negated using *default negation* “not”. A *literal* is an atom a or a negated atom $\text{not } a$. The intuitive idea behind default negation is that $\text{not } a$ is true if we cannot find evidence supporting the truth of a .

Definition 1. A normal logic program is a finite set of rules of form

$$a \leftarrow b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_n, \quad (2.1)$$

where $a, b_1, \dots, b_m, c_1, \dots, c_n$ are atoms from \mathcal{U} .

The intuitive meaning of a rule is the following: if all atoms b_1, \dots, b_m are known to be true, and there is no evidence for any atom c_i ($1 \leq i \leq n$) to be true, then a must be true as well.

Definition 2. For a rule r of form (2.1), define

- (i) the head of r as $H(r) = \{a\}$,
- (ii) the positive body of r as $B^+(r) = \{b_1, \dots, b_m\}$, and
- (iii) the negative body of r as $B^-(r) = \{c_1, \dots, c_n\}$.

Furthermore, the body of r is $B(r) = B^+(r) \cup \text{not } B^-(r)$, where

$$\text{not } B^-(r) = \{\text{not } c \mid c \in B^-(r)\}.$$

Definition 3. A rule r of form (2.1) is a fact if $B(r) = \emptyset$.

For facts, the symbol “ \leftarrow ” will usually be omitted, and the body of a fact is the empty set. Intuitively, facts state what is known to be unconditionally true.

An *interpretation* is an assignment of truth values to the elements of \mathcal{U} . It can be conveniently specified as a the set of atoms from \mathcal{U} that are true, all other atoms are assumed to be false.

Definition 4. An interpretation is a finite set of atoms from \mathcal{U} .

Definition 5. Given a rule r of form (2.1) and an interpretation I ,

1. I satisfies $B(r)$, denoted $I \models B(r)$, iff both $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$; and
2. I satisfies r , denoted $I \models r$, iff $I \cap H(r) \neq \emptyset$ whenever $I \models B(r)$.

Definition 6. An interpretation I is a model of a program P , in symbols $I \models P$, if $I \models r$ for each rule $r \in P$.

The semantics of normal programs is defined in terms of *answer sets*,¹ i.e., assignments of true and false to all atoms in the program that satisfy the rules in a minimal and consistent fashion.

Following Gelfond and Lifschitz [76], answer-sets are defined as models that satisfy the following fixed-pointed condition:

Definition 7. The reduct of a program P with respect to an interpretation I is the program

$$P^I = \{H(r) \leftarrow B^+(r) \mid r \in P \text{ and } B^-(r) \cap I = \emptyset\}.$$

Definition 8. An interpretation I is an answer set of P iff I is a subset-minimal model of the reduct P^I . By $AS(P)$, we denote the collection of all answer sets of P .

Programs can yield no answer set, one answer set, or several answer sets. For instance, the program

$$\begin{aligned} p &\leftarrow \text{not } q, \\ q &\leftarrow \text{not } p. \end{aligned} \tag{2.2}$$

has two answer sets: $\{p\}$ and $\{q\}$.

When we represent a problem in ASP, some rules “generate” answer sets corresponding to “possible solutions”, and some “eliminate” the answer sets that do not correspond to solutions. The rules in program (2.2) are of the former kind; *constraints* are of the latter kind.

¹In the original work of Gelfond and Lifschitz [76], the term *stable model* was used; when they subsequently generalised the concept, the name “answer set” was employed [77].

In general, a constraint is an expression of the form

$$\leftarrow b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_n, \quad (2.3)$$

which is an abbreviation for the rule

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

where a is a globally new atom.

The semantics of a constraint is that it eliminates any answer set candidates that satisfies the body of the constraint. For instance, adding the constraint

$$\leftarrow p$$

to a program P eliminates all answer sets of P containing p . In particular, adding $\leftarrow p$ to program (2.2) eliminates the answer set $\{p\}$.

We will make use of the following technical definitions for our subsequent elaborations. As usual, let HB_P denote the *Herbrand base* of a program P , i.e., the atoms occurring in P .

Definition 9. Given an atom $a \in \text{HB}_P$,

$$\text{Def}_P(a) = \{r \in P \mid \text{H}(r) = \{a\}\}$$

is the set of defining rules of a in P , i.e., the definition of a .

Furthermore,

$$\text{SuppR}(P, I) = \{r \in P \mid I \models \text{B}(r)\}$$

is the set of supporting rules of P under an interpretation I .

Definition 10. The positive dependency graph of a program P is the directed graph $\langle V, A \rangle$, where

- (i) $V = \text{HB}_P$ and
- (ii) $\langle a, b \rangle \in A$ iff there is a rule $r \in P$ such that $a \in \text{H}(r)$ and $b \in \text{B}^+(r)$.

A non-empty set L of atoms is a loop of a program P iff, for each pair $a, b \in L$ of atoms, there is a path π of length greater than or equal to 0 from a to b in the positive dependency graph of P such that each atom in π is in L .

We say that an atom a is a *brave consequence* of P if a is in some answer set of P . Atom a is a *cautious consequence* of P if a is in each answer set of P .

2.1.2 Language Extensions

Different extensions of the basic ASP language have been proposed. On the one hand, we have syntactic extensions, providing mere, but very useful, syntactic sugar. On the other hand, we have semantic extensions where the formalism itself is generalised.

Strong Negation

Strong negation, introduced by Gelfond and Lifschitz in 1991 [77], allows to explicitly state that something is not true. In a program with strong negation, any atom a in a rule of form (2.1) may be strongly negated, denoted by $\neg a$. Answer sets are defined as for normal logic programs except that an interpretation may also contain strongly negated atoms. However, for any interpretation I and any atom a , $\neg a \in I$ implies $a \notin I$.

Choice Rules

From a programmer's perspective, *choice rules* [122] are probably the most commonly used extension. Many problems require choices between a set of atoms to be made. Although this can be modelled in the basic formalism, it tends to increase the number of rules and increases the possibility of errors.

When we represent a problem in ASP, we often use *cardinality expressions* [159] of the form

$$l \{a_1, \dots, a_k\} u$$

where each a_i is an atom and l and u are non-negative integers denoting the *lower bound* and the *upper bound* of the cardinality expression, respectively. Programs using these constructs can be viewed as abbreviations for particular normal programs [67]. Such an expression describes the subsets of the set $\{a_1, \dots, a_k\}$ whose cardinalities are at least l and at most u . Bound l defaults to 0 while u defaults to k . In heads of rules, cardinality expressions generate answer sets containing subsets of $\{a_1, \dots, a_k\}$ whose cardinality is at least l and at most u . Rules of this kind are called *choice rules*. When used in constraints, they eliminate answer sets that contain such respective subsets.

Disjunction

One of the major extensions to the language was the introduction of disjunction in the head of rules [77]. A disjunctive rule is of form

$$a_1 \vee \dots \vee a_l \leftarrow b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_n. \quad (2.4)$$

When the body of such a rule is true, we need to have at least one head atom a_i , for $1 \leq i \leq l$, to be true. Once we define $H(r) = \{a_1, \dots, a_l\}$ for a rule r of form (2.4), our definition of answer-sets of normal programs applies to programs with disjunctive rules as well.

Although at first glance it may seem that disjunctive programs can easily be translated to non-disjunctive programs, this is not the case in general. Both types of programs are

in different complexity classes (under the usual complexity-theoretic proviso that the polynomial hierarchy does not collapse).

Programs with Variables

A group of rules that follow a particular pattern can often be described in a compact way using *schematic variables*. For instance, we can write the program

$$p_i \leftarrow \text{not } p_{i+1} \quad (1 \leq i \leq 7)$$

as follows:

$$\begin{aligned} & \text{index}(1), \text{index}(2), \dots, \text{index}(7), \\ & p(i) \leftarrow \text{not } p(i+1), \text{index}(i). \end{aligned}$$

ASP solvers compute an answer set for a given program that contains variables after “grounding” the program, e.g., by the grounder **gringo** [73]. A grounder systematically replaces each rule r with variables by its ground instances that result from r by uniformly replacing each variable by constants from the program. Variables can also be used “locally” to describe a list of literals. For instance, the rule

$$1 \{p_1, \dots, p_7\} 1$$

can be represented as

$$1 \{p(i) : \text{index}(i)\} 1.$$

To represent actual programs with variables, we will often use the basic Prolog-style rule syntax that corresponds to the input language of the ASP grounder **gringo** [73, 70] as seen in the 3COL example. In this language, an *ordinary atom* is an atom from some fixed first-order language determined by sets of predicate, function, and constant symbols, possibly preceded by the symbol “-” representing *strong negation*. An *ordinary literal* is an ordinary atom possibly preceded by “not”, the symbol for *default negation*. An *atom* is either an ordinary atom or an *aggregate atom* which is an expression of form

$$\begin{aligned} & l \# \text{count} \{ l_1, \dots, l_m \} u \quad \text{or} \\ & l \text{ op } [l_1 = w_1, \dots, l_m = w_m] u, \end{aligned}$$

where l_1, \dots, l_m form a multiset of ordinary literals, each w_i is an integer weight assigned to l_i , $\text{op} \in \{\# \text{sum}, \# \text{min}, \# \text{max}, \# \text{avg}\}$, and $0 \leq l \leq u$ are two integer bounds. Intuitively, an aggregate atom is true if “op” maps the (weighted) satisfied literals l_i to a value within bounds l and u . Aggregate names $\# \text{count}$ and $\# \text{sum}$ may be omitted. A *literal* is either an atom or a default negated atom.

A rule r is *safe* if all variables occurring in r also occur in the positive body of r , and we assume all rules to be safe. A *program* is a finite set of (safe) rules. An expression (program, rule, atom, etc.) is *ground* if it does not contain variables. Given a program P , we refer to the set of constant symbols occurring in P as the *active domain* of P .

For a program with variables, an *interpretation*, I , is a set of ordinary atoms that does not include an atom and its strong negation. The semantics of this class of logic programs is defined by a two step procedure: First, a program P is *grounded*, i.e., it is transformed into a corresponding ground program, basically by replacing each non-ground rule by its propositional instances that are obtained by uniformly replacing each variable by constant symbols from the active domain of P . We denote the grounding of P by $grnd(P)$. Then, answer sets are defined for $grnd(P)$ corresponding to the semantics for normal logic programs [76] augmented with the *choice semantics* for aggregate atoms in rule heads [67, 159].

Sometimes, one is not only interested in arbitrary solutions to a problem but in solutions that are optimal according to some preference relation. Minimise statements allow to express such preferences. For illustration, assume that, e.g., we want to minimise the number of blue nodes in the above 3COL example. This can be expressed by simply adding the following minimise statement:

```
#minimize[asgn(N,blue) : node(N)].
```

The meaning of such a statement is that the ASP solver computes answer sets where the sum of literals $asgn(N,blue)$, where N is a node, is minimal among all answer sets.

In addition to the constructs we have discussed so far, current state-of-the-art ASP solvers support many further language extensions like *functions*, *built-in arithmetic*, *comparison predicates*, *aggregate atoms*, *maximisation statements*, as well as *weak constraints*. For a more detailed description of current solver languages, we refer to respective user manuals [45, 153].

2.2 Complexity Theory

We proceed by recapitulating some basic facts about complexity theory; for more information, we refer to the well-known treatise by Papadimitriou [149]. Problems are examined with respect to their structural, i.e., problem inherent, complexity. A *problem class* is defined as a set of problems that can be solved using a certain *machine model* under bounded resources.

The classical machine model is the *Turing machine* [164]. Resources for computation are usually limited by running time or memory space. We distinguish between *deterministic* Turing machines (DTMs) and *non-deterministic* Turing machines (NDTMs). Both have the same computational power, but the NDTM branches non-deterministically from one state to a set of successor states and can execute computation in some sense in parallel. Whether a NDTM can be simulated by a DTM in polynomial time is one of the most prominent open question in theoretical computer science so far.

A *problem description* (or simply a *problem*) is a pair (L, Y) , where L is a formal language and $Y \subseteq L$, representing the *positive instances*. The *decision problem* associated with (L, Y) is the problem of determining whether $I \in Y$, for some *problem instance* $I \in L$.

A *complexity class* is a collection of decision problems. We consider the following prominent complexity classes:

- P, the class of problems that can be solved by a DTM in polynomial time with respect to the size of the problem instance;
- NP, the class of problems that can be solved by a NDTM in polynomial time with respect to the size of the problem instance; and
- PSPACE, the class of problems that can be solved by a DTM using polynomial space with respect to the size of the problem instance.

For a problem $A = (L, Y)$, the *complementary problem*, \bar{A} , is defined as $(L, L \setminus Y)$. For a problem class C , the complementary class, $\text{co-}C$, is defined as $\{\bar{A} \mid A \in C\}$.

The following set inclusions hold:

$$P \subseteq NP \subseteq PSPACE.$$

It is unknown so far whether these inclusions are proper or not.

Given two problems $P_1 = (L_1, Y_1)$ and $P_2 = (L_2, Y_2)$, P_1 is *polynomial-time many-to-one reducible* to P_2 , denoted by $P_1 \preceq P_2$, iff there exists a function f that maps each problem instance $I_1 \in L_1$ to an instance $I_2 \in L_2$ in polynomial time such that $I_1 \in Y_1$ iff $f(I_1) \in Y_2$. For a class C , a problem A is called *hard with respect to C* , or *C -hard*, iff for each $B \in C$, it holds that $B \preceq A$. A is called *complete with respect to C* , or *C -complete*, iff it is hard for C and a member of C , i.e., $A \in C$. To prove completeness for a problem A with respect to a class C , it is sufficient to

1. show that $B \preceq A$ for some problem B that is hard for C , and
2. show membership of A in C .

Many problems, especially in the field of propositional non-monotonic reasoning, are hard for NP (resp., co-NP) and in the class PSPACE without being PSPACE-hard. For those problems, it is appropriate to make use of the concept of *oracle classes*. An oracle for a class C can be regarded as a procedure that solves problems $A \in C$ in constant time. For a complexity class C , the class P^C (resp., NP^C) denotes the class of problems that can be solved by a DTM (resp., NDTM) with the help of an oracle for class C in polynomial time.

The *polynomial hierarchy* (PH) [161] consists of classes $\Sigma_k^P, \Pi_k^P, \Delta_k^P$, $k \geq 0$, and is inductively defined as follows: for $k = 0$,

$$\Sigma_0^P = \Pi_0^P = \Delta_0^P = P,$$

and, for $k > 0$,

$$\begin{aligned}\Delta_{k+1}^P &= P^{\Sigma_k^P}, \\ \Sigma_{k+1}^P &= NP^{\Sigma_k^P}, \\ \Pi_{k+1}^P &= \text{co-}\Sigma_{k+1}^P.\end{aligned}$$

The following relations hold:

$$\begin{aligned}\Delta_k^P &\subseteq (\Sigma_k^P \cap \Pi_k^P), \\ (\Sigma_k^P \cup \Pi_k^P) &\subseteq \Delta_{k+1}^P, \text{ and} \\ \bigcup_{k=0}^{\infty} \Sigma_k^P &\subseteq \text{PSPACE}.\end{aligned}$$

We also mention complexity class D^P which consists of decision problems that can be characterised by a conjunction of an NP and an independent co-NP problem.

2.2.1 Parameterised Complexity

Some problems can be solved by algorithms that are exponential only in the size of a fixed parameter while polynomial in the size of the input. They are called *fixed-parameter tractable*; more details on parameterised complexity can be found elsewhere [47, 68].

In more formal terms, a parameterisation of a decision problem is obtained by assigning a natural number to each problem instance. A parameterised decision problem is fixed-parameter tractable if a problem instance x with parameter k can be decided in running time $f(k) \cdot |x|^{O(1)}$, where f is a computable function which is independent of $|x|$.

In standard complexity theory, problems are classified and ordered into hierarchies using polynomial-time reductions. Under parameterised complexity, parameterised reductions, so-called *fpt-reductions*, are used for this purpose. An fpt-reduction from a parameterised problem P to a parameterised problem Q is a function φ such that for any instance x of P

- (i) $\varphi(x)$ can be computed in time $f(k) \cdot |x|^{O(1)}$, where k is the parameter of x ,
- (ii) $\varphi(x)$ is a yes-instance of Q iff x is a yes-instance of P , and
- (iii) if k is the parameter of x and k' is the parameter of $\varphi(x)$, then $k' \leq g(k)$, for some computable function g .

The class FPT contains all fixed-parameter tractable problems. Note that FPT is closed under fpt-reductions. Similar to the polynomial hierarchy in standard complexity theory, a hierarchy of classes $W[i]$ has been introduced with FPT at its lowest level. In particular, $\text{FPT} = W[0]$ and $W[i] \subseteq W[j]$, for all $i \leq j$. All classes $W[i]$ are closed under fpt-reductions. Moreover, analogous to $P \subseteq NP$, it is not known whether the inclusions $W[i] \subseteq W[j]$ are proper or not.

Testing Uniform ASP Encodings

This chapter is dedicated to methods to test uniform problem encodings in ASP. First, in Section 3.1, we lay down the foundation for a formal and systematic study of testing for ASP. Based on input and output signatures, we introduce formal definitions of test cases and test suites and the conditions under which a program under test passes or fails them. We also discuss the relevant topics of preconditions, admissible inputs, test-input generators, and test oracles for ASP. These definitions and considerations form the basis for later sections on strategies for actually testing programs. The idea to exhaust the test-input space within a fixed scope is discussed in Section 3.2, random testing in Section 3.3, and selecting inputs that “cover” different elements of the program in Section 3.4.

3.1 Basic Concepts for Testing

In a conventional setting, testing aims at increasing the reliability of a software component by executing the code with the intent of finding errors, i.e., mismatches between the actual and expected program output for some given input [118]. Contrary to verification, testing is in general unsuitable to establish total correctness of a program relative to some specification but can be seen as a computationally lighter approach where programming mistakes can be detected from the actual code. In this section, we introduce a general framework for testing answer-set programs, define conditions under which a particular test case passes or fails, and review related issues like the test-oracle problem.

We are mainly concerned with testing *uniform problem encodings* which is the prevalent mode ASP is used. That is, we distinguish between a *fixed program* that specifies solutions to a problem at hand and *problem instances* that are represented as sets of facts. Let us recapitulate the reviewer-assignment problem from the introduction for illustration: We have reviewers and papers and the objective is to find an assignment subject to the following constraints: (i) every paper is assigned to three reviewers, (ii) no reviewer gets

more than four papers, and (iii) no paper gets assigned to a reviewer who has declared a conflict of interest. The following program is a uniform problem encoding of the above problem in `gringo` syntax [154]:

```
{ assigned(R,P) : reviewer(R) } = 3 :- paper(P).
:- { assigned(R,P) : paper(P) } > 4, reviewer(R).
:- assigned(R,P), conflict(R,P).
#show assigned/2.
```

Each rule directly encodes one of the three conditions from above. The last rule specifies that the ASP solver only outputs the projection of the answer sets on atoms over the predicate `assigned/2`. Now, a problem instance is described using the predicates `paper/1`, `reviewer/1`, and `conflict/2`. We consider this as *input* for the problem encoding. The *output* is given by the answer sets projected to the output predicate `assigned/2` of the encoding joined with the facts. Testing means to select different problem instances as inputs and verify that the computed answer sets indeed correspond to all the solutions that we would expect. We will investigate this idea in more formal terms.

3.1.1 Test Cases for ASP

Our notion of test case is designed to reflect the common practice in ASP to write a program as a uniform problem encoding that, given a problem instance as input in form of a set of facts, has answer sets filtered to dedicated output atoms encoding the desired solutions. We will refer to the program to be tested as the *program under test*.

To apply systematic testing to answer-set programs, we make the notions of input and output for a program explicit. Recall that we assume \mathcal{U} to be the universe of atoms used to construct programs from. Here, we are not particular about the considered class of programs and \mathcal{U} might contain atoms from a first-order language if programs with variables are considered or atoms that are preceded by the symbol for strong negation if required. As done in other work [74, 96], we assume two finite sets $\mathbb{I}_P, \mathbb{O}_P \subseteq \mathcal{U}$ of atoms for each program P , where \mathbb{I}_P is the *input alphabet* of P and \mathbb{O}_P is the *output alphabet* of P . Sometimes, we will refer to input and output alphabets as *input signatures* and *output signatures*, respectively; the terms are used synonymously.

If we want to specify input and output signatures of programs with variables, like the one for the reviewer-assignment problem, it is convenient to use sets of predicate symbols instead of sets of atoms. We will identify a set S of predicate symbols with the set of atoms from \mathcal{U} constructible from S in that case. For example, if P is the encoding for the reviewer-assignment problem,

$$\begin{aligned} \mathbb{I}_P &= \{\text{paper}/1, \text{reviewer}/1, \text{conflict}/2\}, \\ \mathbb{O}_P &= \{\text{assigned}/2\}. \end{aligned}$$

Note that as there is no bound on the number of papers and reviewers, sets \mathbb{I}_P and \mathbb{O}_P represent infinite subsets of an appropriately defined universe \mathcal{U} .

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

Definition 11. For a program P with input and output alphabets \mathbb{I}_P and \mathbb{O}_P , a test input, or simply input, of P is any finite set $I \subseteq \mathbb{I}_P$ of atoms.

The output of P with respect to some input I of P is the set

$$P[I] = \text{AS}(P \cup I)|_{\mathbb{O}_P},$$

where $P \cup I$ stands for P augmented by a fact $a \leftarrow$ for each $a \in I$.

Example 1. Consider program P defined as the following set of rules:

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

Assume that $\mathbb{I}_P = \{a, b, c\}$ and $\mathbb{O}_P = \{e, f, g\}$. Let $I_1 = \{a, b\}$ and $I_2 = \{c\}$ be two inputs of P . Then, the resulting sets of outputs are

$$P[I_1] = \text{AS}(P \cup \{a, b\})|_{\mathbb{O}_P} = \{\{e\}\}$$

and

$$P[I_2] = \text{AS}(P \cup \{c\})|_{\mathbb{O}_P} = \{\{f\}\}.$$

Following custom from conventional software testing [118], a test case for a program P consists of a precise description of the correct output of P given some input of P . Since answer-set programs are inherently non-deterministic, there can be several outputs or even no output for an actual input $I \subseteq \mathbb{I}_P$ supplied to a program P . We will use specifications to determine the correct output of a program for any input.

Definition 12. Define a specification for a program P as a mapping σ from sets over \mathbb{I}_P to collections of sets over \mathbb{O}_P .

Then, the correct outputs for P for a given input I are determined by $\sigma(I)$. A specification σ can be seen as a *test oracle* determining the correct outputs for any input.

Definition 13. Program P is correct with respect to its specification σ if $P[I] = \sigma(I)$ for each test input $I \subseteq \mathbb{I}_P$ of P .

Definition 14. Let P be a program and σ a specification for P . Then, a test case T for P and σ is a pair $\langle I, O \rangle$, where $I \subseteq \mathbb{I}_P$ is an input of P and $O = \sigma(I)$. The sets I and O are denoted by $\text{inp}(T)$ and $\text{out}(T)$, respectively.

Definition 15. A test suite \mathcal{S} for some program P and some specification σ for P is a collection of test cases for P and σ . The collection of inputs of \mathcal{S} is given by

$$\text{inp}(\mathcal{S}) = \{\text{inp}(T) \mid T \in \mathcal{S}\}.$$

Definition 16. The exhaustive test suite for P and σ is the suite

$$\mathcal{E}_{P,\sigma} = \{\langle I, O \rangle \mid I \subseteq \mathbb{I}_P \text{ and } O = \sigma(I)\}.$$

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

Definition 17. Let P be a program and $T = \langle I, O \rangle$ a test case for P . Then, P passes T if $P[I] = O$, otherwise P fails T .

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

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

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

If the input alphabet of a program is finite, it is possible to use exhaustive testing to establish program correctness. Unfortunately this is usually not the case.

Proposition 1. Let P be a program and σ a specification for P . Then, P is correct with respect to σ if P passes the test suite $\mathcal{E}_{P,\sigma}$.

Admissible Test Inputs

For testing in practice, usually not all possible inputs over a program's input signature are of interest. In general, there will be different assumptions and restrictions regarding what we consider as valid input for some ASP encoding. They can be regarded as preconditions and are often implicit. That is, inputs of practical use will in general not be arbitrary sets of facts over the input alphabet but structures that adhere to certain constraints.

The output of a program under test is only required to match the expected output for inputs that satisfy a program’s precondition and we would like to only use such inputs for testing—we refer to them as *admissible*.

For example, assume the problem at hand is to decide whether a given tree defined over predicates `node/1` and `edge/2` is balanced. Now, being a tree is a precondition of the program under test and other structures that are not trees are of little use to test this program. In the case of our reviewer-assignment problem, we would expect that `conflict/2` is a relation between reviewers and papers, and that both are actually mentioned in the input instance.

A convenient way to specify admissibility constraints for testing in ASP and thus making them explicit is by using ASP itself. Hence, we can define the test input space of the program under test using a program whose answer sets represent all valid inputs. We refer to such a program as a *test-input generator*. The output signature of the test-input generator is the input signature of the program under test. An input generator will often have a parameter to impose a bound on the size of the test-input space. The following program is an example of a test-input generator for the reviewer-assignment problem:

```
#const n=4.
{ noR(R) : R=0..n } = 1.    % set number of reviewers
{ noP(P) : P=0..n } = 1.    % set number of papers

reviewer(R) :- noR(M), R <= M, R=1..n.
paper(P)     :- noP(M), P <= M, P=1..n.
{ conflict(R,P) } :- reviewer(R), paper(P).

#show reviewer/1. #show paper/1. #show conflict/2.
```

We use a constant `n` as a size limit for the inputs. The actual number of reviewers is non-deterministically set to a value between 0 and `n`. The same holds for the number of papers. Facts `reviewer/1` and `paper/1` are then derived to represent the actual input instance. Also, for any pair of reviewers and papers, we non-deterministically choose if there should be a conflict of interest. Finally, `#show` statements are used to define the input signature, i.e., the predicates that are used to describe a problem instance. Smaller values for `n` result in a smaller size of the input space and inputs can be generated faster; larger values yield more inputs for a more complete testing. Adequate values in practice are the smallest values such that all interesting structural variations of input instances are included while the generation of inputs is still feasible—we will elaborate on this in the next section.

The Oracle Problem

If testing is to be automated, verifying that the actual output matches the intended output requires a test oracle, i.e., some computational means to determine the correct

outcome. Designing test cases by hand in ASP is not only tedious but also difficult and error prone as input structures can be quite complex and a complete test case would also require to specify all outputs for a given input. Also, verifying that an output of a program is correct by hand is not easy as it is presented by the solver to the user as an unstructured list of atoms. Visualisation techniques for ASP can help a great deal in this regard [31, 102]. We will address the problem of computing test outcomes for test case based on ASP itself in Section 3.4.

Fortunately, it is often sensible to assume that an oracle is given by an answer-set program itself: It is usually rather simple to come up with an initial ASP encoding that is easily seen to be correct but lacks other properties like scalability. In the course of refining and rewriting this first design towards a more efficient and thus useful program, different versions will emerge that have to be tested against the original. We will refer to the original program as the *test oracle*. The only assumption is that the program under test and the test oracle have the same input and output signatures. The idea of using an ASP encoding as specification for a program under test will occur frequently in later sections.

3.2 A Small-Scope Hypothesis for ASP

The *small-scope hypothesis* in traditional testing states that a high proportion of errors can be found by testing a program for all test inputs that are taken from some relatively small scope [92], i.e., by restricting the number of objects a test input is composed of. This suggests that it can be quite effective to test a program exhaustively for some restricted small scope instead of deliberately selecting test inputs from a larger one. For illustration, assume we encoded by means of ASP, using the rules below, the graph problem of testing whether a graph is disconnected, where problem instances are represented by facts over an input signature with predicates `edge/2` and `node/1`:

```
reachable(X, Y) :- edge(X, Y) .  
reachable(X, Z) :- reachable(X, Y), reachable(Y, Z) .  
disconnect      :- node(X, Y), not reachable(X, Y) .
```

Applying the small-scope hypothesis would mean that if our encoding is faulty then it is rather likely that an error becomes apparent when testing with small problem instances that consists of not more than, say, three or four nodes.

A small-scope hypothesis in ASP would be a matter of interest for two reasons. First, it would allow to devise effective testing methods for uniform problem encodings, the prevailing representation mode used in ASP. The second reason is that ASP specifications are formulated at the first-order level, while ASP solvers operate at the propositional one and rely on an intermediate grounding step. In fact, a considerable body of literature deals with ASP at the propositional level. This is justified by asserting that first-order ASP representations are only a short-hand for propositional ones via grounding. However, a uniform problem encoding like the one above stands, in general, for a propositional

program of infinite size even when no function symbols are used. This is because the size of problem instances, e.g., the number of nodes in the example, is usually not bounded. Hence, to ground the rules from above such that they can be joined with arbitrary large sets of facts over `edge/2` and `node/1`, we need an infinite supply of constant symbols. This can lead to a rather grave gap between methods and theories, like for equivalence testing or debugging, that operate on the propositional level and their applicability for real-world ASP. The small-scope hypothesis helps to reconcile the propositional and the first-order level by establishing that the grounding of a uniform encoding with respect to a small domain of constants can be, in a sense, representative for the entire infinite program.

Although it is plausible that the small-scope hypothesis is a valid principle in ASP, an investigation whether this conjecture can be answered affirmatively or not has not been studied so far. As well, there is no solid ground that allows to determine an adequate scope for testing.

Being an empirical principle, the small-scope hypothesis is evasive to a formal proof but it can be empirically and experimentally evaluated. Performing such an undertaking is the goal of this section. To this end, we follow Andoni et al. [5] who evaluated the small-scope hypothesis for Java programs that manipulate complex data structures (a more thorough version of their work is published as technical report [116]). Their evaluation is based on *mutation analysis* [42] in which small changes that should simulate typical programmer faults are introduced into a program. This way, a number of faulty versions of a program, so-called *mutants*, are generated. Then, these mutants are exhaustively tested with all inputs from some fixed scope. The original program serves as test oracle in this step; in particular, a mutant is said to be *caught* by some input if its output on that input differs from the output of the original program. Finally, it is investigated how the catching rates change with respect to the size of the scope.

In our work, we adopt the methodology of Andoni et al. [5] for ASP. In particular, based on mutation analysis, we evaluate the small-scope hypothesis using benchmark problems used for the third ASP competition [28, 27]. We show that a rather restricted scope is often sufficient to catch all mutants. Furthermore, we provide concrete hints how to determine a suitable scope and deal with other aspects, like input preconditions, that are of practical relevance.

3.2.1 Prelude: A Key Observation

We first formally define the scope of an input of an answer-set program as follows:

Definition 18. *Let P be a program. Then, an input of P is within scope n iff at most n different constant symbols occur in I .*

For illustration, the following facts

$$\text{node}(1..3) . \text{edge}(1,2) . \text{edge}(2,1) . \text{edge}(2,3) .$$

are an input for the short program from the beginning of this section that is within scope 3, since it contains not more than three different constant symbols.

One could ask whether we could prove the small-scope hypothesis for restricted but still interesting classes of answer-set programs, thus effectively establishing a small-scope *theorem*. However, the answer is negative already for Horn programs, i.e., programs which do not use default negation, strong negation, or aggregates. Also, we assume that Horn programs do not contain function symbols.

Theorem 1. *Given a Horn program P and a number n , determining whether positively testing P with all inputs within scope n implies correctness of P is undecidable, even if the output considered as correct for any input of P is determined by some Horn program itself.*

Proof. The result follows from the undecidability of query equivalence [158], which is the following task: Given two Horn programs P with output signature \mathbb{O}_P and Q with output signature \mathbb{O}_Q . We assume that $\mathbb{O}_P = \mathbb{O}_Q$ and both consists of a single dedicated goal predicate. Furthermore, P and Q have the same input signatures, i.e., $\mathbb{I}_P = \mathbb{I}_Q$, which contain all predicate symbols that only occur in rule bodies of P or Q . The task of deciding query equivalence is now to determine whether $P[I] = Q[I]$ for any I over \mathbb{I}_P .

Towards a contradiction, assume testing P positively with all inputs within scope n implies correctness of P is decidable. We let program Q determine the output that we consider as correct for any input over \mathbb{I}_P . Then, we check whether $P[I] = Q[I]$ for any I within scope n . Clearly, after a finite number of steps, we would conclude either that P is correct or that its output diverges from that of Q for some input—a contradiction to the undecidability of query equivalence. \square

This negative result provides further motivation of the necessity for resorting to an *empirical* evaluation of the small-scope hypothesis.

3.2.2 A Mutation Model for ASP

Mutation analysis [42] is an established approach to evaluate the quality of test suites. Given a program P , we apply certain *mutation operations* on P to obtain versions of P , so-called *mutants*, where small changes are introduced. Intuitively, mutation operations mimic typical programmer mistakes like omission of certain elements or misspelling of names. For ASP, typical mistakes are, e.g.,

- deletion of a single literal from a rule body,
- deletion of a single rule from a program,
- swapping of polarity of literals, i.e., adding or removing default negation in front of an atom,

Table 3.1: Mutation operations for ASP at the rule level.

Name	Operation	Example
RDP	delete proper rule	$p(X) :- q(X) . q(X) :- r(X) .$ $\implies p(X) :- q(X) .$
RDC	delete constraint	$good(x-men) . :- good(X), evil(X) .$ $\implies good(x-men) .$
RDF	delete fact	$good(x-men) . :- good(X), evil(X) .$ $\implies :- good(X), evil(X) .$

- increasing or decreasing a bound of an aggregate by one,
- picking an atom and replacing it with another atom used elsewhere in the program.

Note that these operations may never produce syntactically incorrect programs. Literal deletion simulates that the programmer forgot a literal or misspelled a negative body atom. Rule deletion does not only mimic that the programmer forgot to write a rule but also misspelling of head atoms or positive body atoms. Swapping the polarity of literals simulates the omission or the accidental use of default negation. The modifications of aggregate bounds by one mimic typical off-by-one errors. Replacing atoms simulates to accidentally use a wrong atom name. The stipulation that this name occurs somewhere else in the program makes this operation different from rule deletion. In what follows, we introduce a detailed mutation model for answer-set programs simulating simple programming errors.

Mutation operations were introduced for different programming languages [2, 100, 101, 146]. Here, we introduce a mutation model for ASP—in particular, for the `gringo` input language. Compared to most imperative languages, ASP languages are rather simply structured. Obtaining a complete set of mutation operations—complete in the sense that each language element is addressed by some operation—is thus more easily achieved. While more complex languages, like, e.g., Java, may require a more systematic approach to derive a complete set of mutation operations [100], our selection is more based on programming experience as in the approach of Agrawal et al. [2].

We start with discussing the general design principles underlying our mutation operations. Above all, mutation operations should model common programmer errors. We are interested only in simple errors and consider only single-step operations, i.e., operations that mutate only a single syntactic element at a time. In fact, one important justification of mutation testing is that test cases that are effective in revealing simple errors are, in many cases, also effective in uncovering complex errors. This is what is known as the *coupling effect* [42]. Also, all mutated programs have to be syntactically correct,

Table 3.2: Mutation operations for ASP at the literal level.

Name	Operation	Example
LDB	delete body literal	$\text{norm}(X) \text{ :- } p(X), \text{ not } ab(X).$ $\Rightarrow \text{norm}(X) \text{ :- } p(X).$
LDH	delete head literal	$\text{norm}(X) \text{ :- } p(X), \text{ not } ab(X).$ $\Rightarrow \text{ :- } p(X), \text{ not } ab(X).$
LAD	add default negation	$p \text{ :- } q(X, Y), t(Y).$ $\Rightarrow p \text{ :- } q(X, Y), \text{ not } t(Y).$
LRD	remove default negation	$p \text{ :- } q(X, Y), \text{ not } r(X).$ $\Rightarrow p \text{ :- } q(X, Y), r(X).$
LAS	add strong negation	$\text{person}(X) \text{ :- } \text{mutant}(X).$ $\Rightarrow \text{person}(X) \text{ :- } \text{-mutant}(X).$
LRS	remove strong negation	$\text{norm}(X) \text{ :- } \text{-mutant}(X).$ $\Rightarrow \text{norm}(X) \text{ :- } \text{mutant}(X).$
LRP	rename predicate	$r(X, Y) \text{ :- } e(X, Y). \text{ :- } e(a, b).$ $\Rightarrow r(X, Y) \text{ :- } e(X, Y). \text{ :- } r(a, b).$
LRC	replace comparison relation	$\text{ :- } \text{succ}(X, Y), Y > X.$ $\Rightarrow \text{ :- } \text{succ}(X, Y), Y \geq X.$

otherwise they cannot be used for testing. Mutants which cause syntax errors are sometimes called *stillborn* [146]. In ASP, the most common source for stillborn mutants are safety violations, hence mutation operations have to be designed in a way that they never result in unsafe rules. Another aspect that makes mutation testing in ASP different from, e.g., imperative languages is that some ASP solvers do not guarantee that the grounding of a program is always finite if function symbols or integer arithmetics are used. In our experiments, we do not consider function symbols other than arithmetic operations, still the grounding step is not guaranteed to terminate. Hence, we have to deal with another class of stillborn mutants, namely mutants which are syntactically correct but who cannot be finitely grounded for some input. This kind of mutants is harder to avoid than syntactically incorrect ones and requires a post-processing step where they are filtered out. Filtering out stillborn mutants is done by checking whether a mutant along with its input generator can be grounded for some sufficiently large fixed scope. If this is not possible due to some syntax error, the mutant is marked as

Table 3.3: Mutation operations for ASP at the level of arithmetic expressions.

Name	Operation	Example
ARO	replace arithmetic operator	$\begin{aligned} & \text{next}(X, Y) \text{ :- } r(X; Y), Y=X+1. \\ \Rightarrow & \text{next}(X, Y) \text{ :- } r(X; Y), Y=X*1. \end{aligned}$
ATV	twiddle variable domain	$\begin{aligned} & \text{:- } s(X, Y), X==(Y+X)*2. \\ \Rightarrow & \text{:- } s(X, Y), X==(Y+1)+X)*2. \end{aligned}$
ATA	twiddle aggregate bound	$\begin{aligned} & 1\{\text{guess}(X)\}N \text{ :- } \max(N). \\ \Rightarrow & 1\{\text{guess}(X)\}(N-1) \text{ :- } \max(N). \end{aligned}$
ATW	twiddle aggregate weight	$\begin{aligned} & \text{ok} \text{ :- } 2 [\text{val}(P, V)=V] 8. \\ \Rightarrow & \text{ok} \text{ :- } 2 [\text{val}(P, V)=(V+1)] 8. \end{aligned}$

Table 3.4: Mutation operations for ASP at the level of terms.

Name	Operation	Example
TST	swap terms in literals	$\begin{aligned} & \text{less}(X, Y) \text{ :- } n(X, Y), X < Y. \\ \Rightarrow & \text{less}(Y, X) \text{ :- } n(X, Y), X < Y. \end{aligned}$
TVC	change variable to constant	$\begin{aligned} & p(a;b). \text{fail} \text{ :- } p(X). \\ \Rightarrow & p(a;b). \text{fail} \text{ :- } p(a). \end{aligned}$
TRV	rename variable	$\begin{aligned} & \text{first}(X) \text{ :- } \text{rel}(X, Y). \\ \Rightarrow & \text{first}(Y) \text{ :- } \text{rel}(X, Y). \end{aligned}$
TCV	change constant to variable	$\begin{aligned} & \text{ok} \text{ :- } \text{exit}(1, Y), \text{grid}(X, Y). \\ \Rightarrow & \text{ok} \text{ :- } \text{exit}(X, Y), \text{grid}(X, Y). \end{aligned}$
TRC	rename constant	$\begin{aligned} & \text{first}(\alpha). \text{last}(\omega). \\ \Rightarrow & \text{first}(\omega). \text{last}(\omega). \end{aligned}$

stillborn. Also, if a time limit of a few seconds is reached (implying that the grounding is presumably not finite), the mutant is marked as stillborn and subsequently not considered for testing. We note that deciding whether or not the grounding is finite is undecidable in general. However, if grounding of the original program is fast and a mutant cannot be grounded within seconds, it is sensible to assume that grounding will not terminate at all in practice.

Another important design aspect of mutation operations is that mutants that are equivalent to their reference programs have to be avoided as far as possible. In more formal terms, if P is a program and P' is a mutated version of P , we say that P is the *reference program* (for P'). A mutant is *caught* if there is some admissible input I of P such that $P[I] \neq P'[I]$. If no such input exists, P' is *equivalent* (to the reference program). Note that any mutant has the same input and output signature, as well as the same program preconditions, as its reference program. By definition, equivalent mutants cannot be caught by any test input; ideally, one only takes non-equivalent mutants into account when assessing catching rates. We thus have to manually identify equivalent mutants which is a tedious task if the mutation model does not avoid them to a large extent already in the generation process. Besides mutants that are equivalent to their reference program, we aim for a low number of mutants that are pairwise equivalent. On the one hand, we avoid such mutants by keeping track of which mutations have been applied already to generate some mutant. So, we can avoid that the same operation is applied twice when generating a set of mutants for a program under test. Besides that, we define mutation operations in a way that makes it more unlikely that they semantically simulate each other.

Following Agrawal et al. [2], we classify mutation operations according to the level on which they operate. At the lowest level are mutations that simulate faults of a programmer when defining simple terms, like wrong names for constants or variables. Then, at the next level, we deal with mutations of more complex terms in the form of arithmetic expressions, like using a wrong mathematical operator. At the next level reside mutations that resemble faults when defining literals, like omitting a default negation or using a wrong predicate name. Finally, we consider mutations that take place at the rule level, e.g., omitting entire rules, constraints, or facts.

Table 3.1–3.4 summarise the mutation operations that we consider for ASP. Each operation is illustrated using a simple example, and each operation has a unique name consisting of three letters. The first letter abbreviates the category the operation belongs to: each operation mutates a program either at the rule level (R), at the literal level (L), at the level of arithmetic expressions (A), or at the level of simple terms (T). Most operations are quite straightforward, others need some explanation. For the operation LAS that adds strong negation, we require that any resulting strongly negated literal already occurs elsewhere in the program. Otherwise, this operation would resemble rule deletion operations if the mutated literal was chosen from the positive body of a rule, or literal deletion operations if the literal was taken from the negative body of a rule. Also, if a literal from the head of a rule is mutated, we would get something quite similar to rule deletion. Likewise, we require for removing strong negation (LRS) that the unnegated literal occurs elsewhere. For analogous reasons as for LAS and LRS, we only rename predicates (LRP) into predicates with the same arity that occur elsewhere already. Also, when renaming a constant symbol (TRC) or when changing a variable into a constant (TVC), the new constant has to occur somewhere else.

The idea of the domain twiddle operations (ATV, ATA, ATW) is to introduce off-by-one

faults into arithmetic expressions, aggregate bounds, and weights in aggregates. Hence, some variable or integer constant X is replaced by $(X \pm 1)$ at random.

To avoid equivalent mutants, we check if the affected terms are different before swapping them within a literal (TST). When changing comparison relations (LRC), we never pairwise interchange relations that express inequality, i.e., $<$, \neq , and $>$, otherwise resulting mutants tend to be equivalent due to symmetries.

Our mutation model will be used later for experimental evaluations of testing methods for ASP. Beyond that, a mutation model for ASP has its worth for itself, e.g., for mutation testing, where a test input collection that catches all mutants of a program under test is generated—such a test suite is called *mutation adequate*. In traditional software testing, mutation adequacy is regarded as a rather strong criterion for testing. A Java implementation of our mutation model is publicly available.¹ It takes as input programs written in the `gringo` input language. The tool is configured using an XML file that specifies the number of mutants to generate, the kind of mutation operations, the mode of their application (all or one), and so on. A more detailed description of the tool can be found online.

3.2.3 Experimental Setup

To evaluate the small-scope hypothesis for ASP, we follow the approach of Andoni et al. [5]. In particular, we apply our mutation model on a set of representative benchmark instances taken from the third ASP solver competition [28]. Then, we exhaustively test each mutant over some fixed scope, and we analyse how mutant catching rates change with the size of that scope.

In what follows, we outline the experimental setup. A detailed description of the considered benchmarks, including problem specifications and encodings, is available on the web² as well as in Appendix A. These benchmarks are a representative cross section of challenging problems for both modelling in ASP as well as solving. The benchmark collection contains both problem specifications, formulated in natural language, and uniform problem encodings that implement the respective specifications. In particular, we consider all problems in P and in NP for which reference encodings are published on the competition’s web page. This includes all problems from the system track of the competition, provided in the ASP-Core language, and two additional ones, provided in the ASP-RFC language [26]. The ASP-Core and the ASP-RFC language can be regarded as simple language fragments common to most ASP solvers, where the latter comprises shared aggregate expressions.

As a preparatory step, we rewrote all benchmarks into `gringo` syntax since we use `gringo` along with the ASP solver `clasp` for determining test verdicts. This step consists of only minor modifications of the programs and thus poses no problem for retaining the validity of our experiments for other classes of ASP language dialects. In particular, we

¹www.kr.tuwien.ac.at/research/projects/mmdasp.

²www.mat.unical.it/aspcomp2011.

- rewrote guesses expressed using disjunction into choice rules,
- replaced `<>` with `!=`,
- rewrote aggregate expressions from the ASP-RFC syntax into `gringo` syntax, and
- rewrote queries into constraints.

Of course, instead of rewriting disjunctive heads into choice rules, we could have used a simple shifting operation. However, we regard shifting as more invasive since it turns one disjunctive rule into several normal rules, and, moreover, choice rules better reflect common ASP practice of how to express a non-deterministic selection of objects. As well, confining to the class of normal programs without choices or disjunction would make our results less relevant. In any case, mutations of choices are directly comparable to modifications of disjunctive heads and thus do not restrict our results.

For each benchmark, we repeat the following steps:

1. we formalise the preconditions of the encoding in ASP,
2. we generate up to 300 mutants according to our mutation model from the previous section, and
3. we exhaustively test each mutant for fixed scopes and assess the catching rates.

We exemplify our methods using the simple benchmark problem `GRAPHCOLOURING`. The encoding appears in Figure 3.1. The input of `GRAPHCOLOURING` is defined over predicates `node/1`, `link/2`, and `colour/1`. An input is admissible if node names are consecutive, ascending integers that start from 1, and `link/2` represents a symmetric relation between nodes. The output signature consists of `chosenColour/2` which encodes a mapping from the nodes of the graph to available colours such that no two adjacent nodes are assigned the same colour.

The ASP formalisation of the preconditions of a program in Step 1 is needed to automatise exhaustive testing in Step 3 of our experimental setup. As discussed already in Section 3.1, we refer to such an encoding as *input generator*. For any program P with input signature \mathbb{I}_P , an input generator $IG[P, n]$ for P is an ASP program whose output signature equals \mathbb{I}_P , and whose output on the empty set is in one-to-one correspondence with the admissible inputs of P within scope n . We assume throughout that any input generator $IG[P, n]$ is defined only over \mathbb{I}_P and possibly globally new auxiliary predicates. Moreover, our testing methods require that atoms over \mathbb{I}_P are not defined in P , i.e., they do not occur in the head of any rule in P (which can always be achieved by slightly rewriting an encoding). A respective input generator for `GRAPHCOLOURING` is given in Figure 3.2. The encoding is parameterised by integer constants s and t . Constant s determines the maximal number of nodes in a graph and t fixes the maximal number of available colours. The sum of s

```

% Guess colours.
{ chosenColour(N,C) } :- node(N), colour(C).

% At least one colour per node.
:- node(X), not coloured(X).
coloured(X) :- chosenColour(X,Fv1).

% Only one colour per node.
:- chosenColour(N,C1),
   chosenColour(N,C2), C1 != C2.

% Adjacent nodes have different colours.
:- link(X,Y), X<Y, chosenColour(X,C),
   chosenColour(Y,C).

```

Figure 3.1: ASP encoding of GRAPHCOLOURING.

and τ gives the scope n .³ Regarding the definition of input generators, we sometimes avoided isomorphic inputs to reduce the size of the input space at the representational level, viz. by adding symmetry-breaking constraints. We provide detailed descriptions of all benchmark instances, input and output signatures, as well as input generators in Appendix A. Related approaches for test input generation, like Korat [16], avoid such isomorphic inputs already when computing test inputs.

In Step 2, each mutant is generated by applying one randomly selected mutation operation from Table 3.1–3.4 to the benchmark program. Stillborn mutants are filtered out in a post-processing step.

For Step 3, let P denote a benchmark encoding and P' a mutant of P . To check whether P' can be caught by an input within scope n , we need to check if $P' \cup IG[P, n]$ and $P \cup IG[P, n]$, or equivalently $grnd(P' \cup IG[P, n])$ and $grnd(P \cup IG[P, n])$, have the same answer sets projected to $\mathbb{I}_P \cup \mathbb{O}_P$. This kind of equivalence problem is actually a special case of a *propositional query equivalence problem* (PQEP) [144]. As an aside, we note that Offutt, Voas, and Payne [146] considered also the notion of weak equivalence for mutants of non-deterministic programs: A mutant is weakly equivalent to a program under test if any produced output is correct. The notion of weak equivalence between a mutant and a program corresponds to *propositional query inclusion problems* (PQIPs) [144] in our setting. However, it is not necessarily the case in ASP that a program yields some answer set. Therefore, an ASP mutant that is inconsistent with any test input corresponds to a non-deterministic mutant that is trivially weakly equivalent with the program under test. Also, the correctness notion for non-deterministic programs by Offutt, Voas, and

³Technically, the scope is given by the maximum of s and τ . However, we count both the integer range defined by s and τ since they are treated independently in the encoding due to different domain predicates domN and domC .

```

domN(1..s).
domC(1..t).

1 { maxN(X) : domN(X) } 1.
1 { maxC(X) : domC(X) } 1.

node(X)    :- domN(X), maxN(M), X <= M.
colour(X)  :- domC(X), maxC(M), X <= M.

{ link(X,Y) } :- node(X), node(Y).
link(X,Y)    :- link(Y,X).

#hide.
#show node/1. #show link/2. #show colour/1.
#show chosenColour/2.
#show maxN/1. #show maxC/1.

```

Figure 3.2: Input generator for GRAPHCOLOURING.

Payne [146] amounts to a PQIP. In the ASP setting, however, we are able to determine total correctness in terms of PQEPs, mainly because of the fixed small scope.

We assume a further syntactic property, called *EVA* (which stands for “enough visible atoms”), that allows to treat PQEPs as special case of *modular equivalence* [147, 95]. Roughly speaking, EVA states that there are no hidden guesses which means that all predicates involved in even cycles through negation also belong to the output signature of a program. The co-NP complexity of deciding modular equivalence allows a reduction approach to ASP itself while deciding PQEPs resides on the third level of the polynomial hierarchy. A respective reduction approach from modular equivalence to ASP is realised by the tool `lpeq` [147], which we used for our experiments.

The output signature $\mathbb{I}_P \cup \mathbb{O}_P$ that is needed for the equivalence tests is specified within the input generator by means of `#hide` and `#show` statements (cf. Figure 3.2). The mentioned assumptions on programs are not a limiting factor in practice: either they hold already, which is usually the case, or they can be satisfied by slightly rewriting the ASP encoding at hand. As EVA was indeed not satisfied by all benchmarks, the repair was to extend output signatures by making predicates involved in hidden guesses visible.

A related approach where PQEPs were used as underlying notion of program equivalence for testing programs was conducted for validating student assignment solutions in previous work [143]. In that case study, the correspondence checking tool `ccT` [142] was used.

Besides catching rates, we also studied how the size of the input space increases with scope. Recall that the admissible inputs of a benchmark problem are defined via respective input generators. However, even for comparably small scopes, the huge number of admissible

Table 3.5: Evaluation results for benchmark instances in P.

Problem Name	#Mutants	Scope	# Inputs	Time (Sec.)	Catch. Rate
REACHABILITY	31	1	2	0.2	0.25
		2	72	0.2	0.96
		3	4770	0.3	1.00
GRBSDINFEXTRCT	249	9	8	2.6	0.01
		10	72	3.5	0.01
		11	584	7.5	0.26
		12	4680	32.6	0.47
		13	37448	227.9	0.75
HYDRAULICLEAKING	55	4	12	1.6	0.65
		5	12	2.9	0.65
		6	816	10.5	0.78
		7	2096	36.5	0.78
		8	63476	148.1	0.83
HYDRAULICPLANNING	247	4	12	4.0	0.55
		5	12	4.7	0.55
		6	816	11.7	0.70
		7	2096	19.9	0.70
		8	63476	157.0	0.81
STABLEMARRIAGE	298	1	1	2.5	0.05
		2	257	5.1	0.82
		3	387420746	56.5	0.86
PARTNERUNITSPOLY	193	3	16	1.8	0.49
		6	65552	3.3	0.77
		9	$1.7 \cdot 10^{10}^\dagger$	12.3	0.90
		12	$5.7 \cdot 10^{17}^\dagger$	59.2	0.91

inputs makes exact counting of answer sets by exhaustive enumeration often infeasible. Hence, we had to resort to an approximation approach.

In fact, we adopted *XOR streamlining* [80] from model counting in SAT for ASP which is based on *XOR constraints*. An XOR constraint on a set S of atoms expresses that an even (or odd) number of atoms from S have to be true. If we add a random XOR constraint to a program, it cuts the solution space in half.

Assume U is a set of propositional atoms. An *XOR constraint over U* is an expression of form

$$a_1 \oplus \dots \oplus a_n,$$

where all a_i , $1 \leq i \leq n$, are from U . Such an XOR constraint is *satisfied* in an interpretation I iff an odd number of atoms from $\{a_1, \dots, a_n\}$ is true in I . We translate an XOR constraint into ASP rules as follows:

$$x_1 \leftarrow a_1. \quad \dots \quad x_n \leftarrow a_n, \text{not } x_{n-1}. \quad x_n \leftarrow \text{not } a_n, x_{n-1}.$$

Then, we add the constraint $\perp \leftarrow \text{not } x_n$ to enforce that the XOR constraint is satisfied, or we add $\perp \leftarrow x_n$ to express that it must not be satisfied.⁴

We add s XOR constraints on the output atoms to an input generator and test whether it yields some answer set. After t such trials, if all trials yield some answer set, we know that $2^{s-\alpha}$ is a lower bound of the exact number of answer sets with at least $1 - 2^{-\alpha t}$ confidence ($\alpha \geq 1$ serves as slack factor). Hence, we can use ASP solvers themselves to find lower bounds on the size of input spaces with arbitrarily high confidence (by either increasing t or α). To achieve a

$$1 - 2^{-7} \geq 99\%$$

confidence, the lower bounds were computed with parameters $t = 7$ and $\alpha = 1$ in our experiments.

3.2.4 Results of the Evaluation

We classified our used benchmark programs according to the hardness of the respective encoded problem. To wit, Table 3.5 summarises the results for the benchmark problems that are solvable in polynomial time and Tables 3.6 and 3.7 contain the results for the NP-hard problems. While the benchmarks in Table 3.7 contain at least one recursive predicate, the problem encodings in Table 3.6 are tight, i.e., for any input, the grounding of the encoding joined with that input does not involve positive recursion. Tight programs are in a sense easier than non-tight ones since they can be translated directly, i.e., without extending the language signature, to SAT while this is presumably not possible for non-tight programs; respective translations come with an exponential blowup with respect to the program size in the worst case [111, 110]. All experiments were carried out on a MacBook Pro with a 2.53 GHz Intel Core 2 Duo processor, 4 GB of RAM, and Mac OS X 10.6.8 installed.

For each benchmark in Tables 3.5, 3.6, and 3.7, we give the number of generated mutants and the size of the scope in terms of constant symbols in ascending order. Then, for each benchmark and each scope, we report on the size of the input space. We either give the exact number whenever exact model counting is feasible, or we give a lower bound (marked with a dagger, “†”) of the size of the input space with a confidence of at least 99% where we follow the stochastic approach sketched in the previous section. Generally, input-spaces grow exponentially with respect to scopes.

We also provide the average time in seconds that was spent to exhaustively test all mutants for some scope. It turns out that runtimes keep within reasonable bounds, even for larger scopes. Catching all mutants that are not equivalent to their reference program takes at most 930 seconds for the KNIGHTTOUR benchmark. Finally, we give the catching rates for each benchmark and scope, i.e., the ratio of mutants that were caught by some input within the considered scope and the total number of mutants.

⁴For an alternative way to implement XOR constraints, cf. the recent work of Everardo, Janhunen, Kaminski, and Schaub [61].

Table 3.6: Evaluation results for benchmark instances in NP with tight encoding.

Problem Name	# Mutants	Scope	# Inputs	Time (Sec.)	Catch. Rate
FASTFOODOPTCHECK	167	1	3	1.4	0.02
		2	17	2.7	0.28
		3	87	8.8	0.91
		4	481	43.1	0.94
		5	2663	293.7	0.94
KNIGHTTOUR	292	5	$1.8 \cdot 10^{22}^\dagger$	18.6	0.00
		6	$3.2 \cdot 10^{32}^\dagger$	917.1	0.77
		7	$7.1 \cdot 10^{44}^\dagger$	929.9	0.78
DISJUNCTSCHEDULING	285	1	2	2.4	0.07
		2	349	3.6	0.35
		3	3896155	11.8	0.76
		4	$1.3 \cdot 10^{11}^\dagger$	194.3	0.79
PACKINGPROBLEM	285	2	1	2.4	0.32
		6	56	10.4	0.64
		12	1971	410.8	0.87
MCSYSQUERYING	275	5	10644498	2.7	0.40
		6	$6.8 \cdot 10^{10}^\dagger$	2.8	0.80
		7	$5.6 \cdot 10^{14}^\dagger$	3.3	0.96
		8	$7.3 \cdot 10^{19}^\dagger$	4.7	0.98
HANOITOWER	284	6	16	9.3	0.62
		7	416	15.4	0.77
		8	832	22.9	0.82
		9	29632	41.0	0.82
		10	44448	53.0	0.87
GRAPHCOLOURING	67	2	2	0.6	0.22
		3	4	0.6	0.43
		4	20	0.6	0.95
SOLITAIRE	281	2	4	2.6	0.00
		4	512	3.4	0.04
		6	786432	16.0	0.96
		8	$2.1 \cdot 10^9^\dagger$	245.4	0.97
WEIGHTASGTREE	219	2	1	1.9	0.01
		3	4	2.1	0.01
		4	32	3.9	0.32
		5	126	5.2	0.32
		6	999	40.3	0.78
		7	4368	53.6	0.78
		8	41664	750.7	0.79

We generated up to 300 mutants for each benchmark. However, for many problems the encoding is rather simple regarding the number and structure of rules, and exhaustively applying mutation operations gives far less mutants than 300. Also, filtering out stillborn mutants usually further reduces the number of mutants by about 20%.

Table 3.7: Evaluation results for benchmark instances in NP with non-tight encoding.

Problem Name	# Mutants	Scope	# Inputs	Time (Sec.)	Catch. Rate
SOKOBANDECISION	295	2	4	3.0	0.04
		3	8	3.4	0.06
		4	256	4.4	0.60
		5	384	5.8	0.60
		6	36864	25.4	0.84
		7	49152	53.2	0.84
LABYRINTH	293	8	16777216	352.0	0.94
		6	$4.3 \cdot 10^{12}^\dagger$	3.4	0.06
		7	$8.7 \cdot 10^{12}^\dagger$	4.0	0.07
NUMBERLINK	222	8	$4.7 \cdot 10^{21}^\dagger$	23.9	0.94
		4	24	2.7	0.36
		5	56	3.2	0.36
		6	16568	20.9	0.63
MAGIC SQUARE SETS	270	7	99576	30.4	0.63
		8	92751096	517.2	0.67
		5	332	3.0	0.97
MAZE GENERATION	292	9	1048908	5.9	1.00
		3	468840	8.9	0.76
		4	42063108	16.0	0.82
		5	$2.1 \cdot 10^{12}^\dagger$	41.8	0.95

We did not exclude mutants that are equivalent to their reference program. In general, deciding whether some mutated program is equivalent to its reference program is undecidable which follows from the undecidability of query equivalence [158]. A hint to know when (almost) all non-equivalent mutants are caught is when reaching a fixed-point regarding the scope size and catching rates: if further increasing the scope does not give higher catching rates, we check manually if the remaining mutants are equivalent (at least we considered a random sample from the remaining mutants for these tests). Hence, the difference between 1 and the respective catching rates gives the rate of mutants that are equivalent to their reference program. Thus, it gives an evaluation of our mutation model for each benchmark as a by-product. Depending on the benchmark, the ratio of equivalent mutants can be as high as 0.33 for NUMBERLINK.

The main reason for equivalent mutants is redundancy in the benchmark encodings; they often contain redundant literals in rules or even redundant rules. Clearly, mutating redundant parts of a program usually leads to equivalent mutants, this is comparable to mutating dead code in imperative languages. Candidates for redundant rules typically are sets of constraints where one constraint is backing up for another constraint due to symmetries.

Recall that scope is measured in terms of constants in inputs. Regarding the progression of scope sizes, it is to say that we only indirectly determined the scopes by setting

parameters of respective input generators like s and t in the `GRAPHCOLOURING` example from the previous section. For others, parameters control size of grids, sets, etc. Although we mostly incremented these parameters stepwise, the actual scope sometimes increases in a quadratic manner like for the `PACKINGPROBLEM`. The relation between parameters and scopes depends on the individual encodings. We report only on scopes such that the set of resulting admissible inputs is not empty. Also, sometimes encodings mention already constant symbols like in `GRBSDINFEXTRACT`, hence the active domain of considered encodings is not always empty. In such cases, we let the active domain (or at least subsets identified by equivalence partitioning) contribute to the scope of generated inputs right from the beginning. Hence, the scope for, e.g., `GRBSDINFEXTRACT` starts already with 9.

An interesting observation is that the problem complexity, viz. P or NP, does not seem to have a big influence on the size of the scope needed to catch all mutants. It turned out that the considered encodings require a scope greater than the size of their active domains plus the maximal number of distinct variables occurring in any rule. While this number serves as a lower bound for estimating suitable scope sizes, no encoding requires more than a few additional constants. Hence, the size of the scopes required to obtain mutation-adequate test suites in fact allows for exhaustive testing using existing equivalence checking tools that operate on the propositional level. If the number of test inputs that have to be considered is prohibitively large for explicit testing or if no reference program is available, one may use other test input selection strategies like random testing or structure-based testing that will be developed for ASP in the sequel.

In general, the size of the active domain plus the maximal number of distinct variables occurring in any rule of a program is a good starting point for determining the size of a suitable scope. Our results also show that testing methods devised for propositional programs can be quite effective for non-ground programs as well. Since the scope needed for testing is small, the increase of size when grounding a program seems manageable for respective tools. Results of this work can be directly transferred to practice, e.g., for exhaustively testing programs submitted to an ASP solver competition over some small domain that can be learned by mutation analysis using respective reference encodings.

Mutation testing for itself can also be used as a tool to detect redundancies in ASP encodings. Sometimes, redundant rules or literals in an ASP program originate from redundant conditions in a problem statement already. More often, however, redundancies are a hint for program errors. To check for redundant rules or literals in a program, we systematically apply rule or literal deletion mutations on the program under test, respectively. Then, we fix a small scope and check whether we can catch all mutants. If we cannot catch a mutant, the small-scope hypothesis implies that the mutant is most likely equivalent and the deleted rule or literal was redundant.

Although our mutation model has been introduced for the `gringo` language, it should provide as well inspiration for respective models for other solver dialects like that of `DLV`. On another hand, we may reduce the set of mutation operations that is considered by identifying a subset of operations such that mutation adequate test suites are not less

effective for testing than test suites generated using the complete set of operations. Such an approach is known as *selective mutation* [145].

3.3 Random Testing

We now investigate the topic of *random testing*, where test inputs are drawn randomly from the test-input space. In particular, we present *Harvey*, a tool for random testing of ASP programs that relies on XOR streamlining [81] to achieve a near-uniform distribution of test inputs.⁵

Random testing is a simple black-box testing approach that, as the name suggests, requires that test inputs are randomly selected from the test-input space. The input selection should ideally reflect the expected usage pattern for the software in its operational environment. If such a pattern is not known, as it is usually the case, the best one can do is to draw samples from a uniform distribution over the test inputs. Although this approach might seem not very sophisticated at first glance, its conceptual simplicity can be regarded as an advantage, and random testing also showed to be quite effective for conventional testing [48]. Regarding more systematic testing methods, random testing is useful as a baseline for comparison when a rigorous theoretical analysis of fault detection capabilities is otherwise practicably infeasible [86].

In our subsequent discussion, we even go beyond simple random testing by allowing to incorporate constraints in the process to guide the input generation towards especially interesting areas of the test-input space. On the one hand, *Harvey* is a simple tool that can readily be used to support an ASP engineer to test ASP encodings. On the other hand, we will use random testing to experimentally evaluate structure-based testing methods in a subsequent section of this thesis.

As discussed in Section 3.1, we are concerned with testing *uniform problem encodings*, i.e., we distinguish between a *fixed program* that specifies solutions to a problem at hand and *problem instances* that are represented as sets of facts and serve as input. We need to overcome basically two challenges to realise random testing for ASP:

1. generating random test inputs, and
2. determining test verdicts automatically.

Harvey realises random testing for ASP such that test-input generation and determining test verdicts is done by employing ASP itself. The tool is comprised of two scripts, implemented in Python: the first one generates random inputs by exploiting ASP rules and achieves uniformity of the test-input selection by using XOR streamlining, while the second one takes care for the actual testing procedure, where answer sets of the

⁵The name of the tool refers to Harvey Dent, one of the infamous adversaries of Batman, who, under his pseudonym Two-Face, makes all important decisions by flipping a coin.

tested program joined with the generated input are compared with those of a test oracle under the respective inputs. For the answer-set generation, the solver `clasp` [154] is used. We discuss our solution approach to the above mentioned challenges as well as its implementation in the following sections in more detail.

3.3.1 Generating Random Inputs

Regarding the first issue, a straightforward selection strategy would be to pick subsets over a program's input signature such that the selection is uniformly distributed. Unfortunately, simple combinatorial considerations make it clear that it is often very unlikely that even a vast number of such random test inputs contains a single input that is admissible. Recall that an input is admissible if it is consistent with the preconditions of an encoding. For illustration, assume we are dealing with the problem of maze generation which requires that there is precisely one fact that specifies the position of the entrance to a maze on a grid. If we deal with 7×7 grids, then there are 49 admissible positions for an entrance vis-a-vis to a total number of 2^{49} subsets of entrance atoms from the program's input signature. The probability that a test input satisfies all preconditions is quite low.

The selection of random inputs has therefore to be based on a program's input generator which makes its preconditions explicit (cf. Section 3.1). As any answer set of a program's input generator constitutes an admissible input, generating admissible inputs that are uniformly distributed reduces to the problem of computing uniformly-distributed answer sets of an ASP input generator. A conceptually simple way would be to enumerate all answer sets and pick the required number of answer sets at random. Unfortunately, a complete enumeration is seldom an option as the number of answer sets will often be prohibitively large. However, to achieve a near-uniform distribution, an approach based on XOR streamlining [81] can be utilised. It has been used for SAT but is applicable to general combinatorial spaces. Recall from Section 3.2.3 that an XOR constraint is basically a parity constraint on a set S of atoms that expresses that an odd number of atoms from S has to be true. We can add such constraints to a program, each one cuts the solution space in half. We keep adding constraints until a complete enumeration is possible. Basically, the script `xorsample.py` re-implements the tool `xorro` [154] taking current ASP language features into account as well as utilising certain optimisations that are relevant for test-input generation.

The process of computing random answer sets is as follows: First, we add a number of XOR constraints over the ground atoms to an encoding. For each atom and each XOR constraint, it is decided with a fixed probability p (the default is 0.5) whether it should be in that constraint or not. Also, it is decided with probability 0.5 whether this constraint has to be satisfied or not. The probability p can be changed by the user; smaller values yield shorter constraints and help to boost performance in practice without changing the distribution too much from an ideal uniform one [79].

Computing a given number of random test inputs proceeds basically in three steps:

1. The input generator is grounded and s XOR constraints are added to the program. By default, $s = \log(n)$, where n is the number of atoms.
2. We try to enumerate all answer sets, if no answer set is found, the number of XOR constraints is decreased. If the enumeration cannot be completed within a given time limit, the number is increased.
3. If the number of resulting answer sets is greater than the number of inputs that we still need, we select n answer sets randomly and discard the rest. Otherwise, we keep all of them and proceed with Step 2, where we use a fresh set of XOR constraints.

3.3.2 Testing With Random Inputs

After test inputs have been generated, we next use them for actual testing. Hence, we need to address the second challenge indicated at the beginning of the section, i.e., automatically determining test verdicts.

As discussed in Section 3.1, if we want to test programs automatically, then verifying that the actual output matches the intended output requires a test oracle, i.e., some computational means to determine the correct outcome. For our random testing approach, we assume that such an oracle is given by an answer-set program itself. This can be a simple first design that is subsequently refined to address, e.g., scalability issues. When developing programs incrementally, it can be useful to test the emerging program versions against the original one.

Based on randomly selected answer sets from an input generator, there are different options to implement random testing of the program under test. One possibility is to first compute one answer set of the input generator and then to verify that *all* answer sets of the input joined with the test oracle and the program under test are matching, i.e., coincide on the output predicates. Sometimes, this can be realised using equivalence checkers for ASP [95, 142], but they come with their own limitations like scalability or certain syntactical restrictions on the programs. A more pragmatic approach is to compute only one answer set of the program under test joined with the input. If there is no answer set, we check whether the test oracle yields no answer set as well on the test input. Otherwise, we check whether this resulting answer set, projected to the output predicates, is an answer-set of the test oracle as well. This is the approach that is used in Harvey. In testing, practicality is key!

For each test input, we proceed as follows: We ground the program under test joined with the test input. We then use `clasp` to compute one answer set. We distinguish between two cases: First, if no answer set is found, we check whether we get no answer set with the test oracle as well. If so, the test case passed, otherwise it failed. For the second case, assume that $A = \{a_1, \dots, a_m\}$ is an answer set of the program under test. We need to verify that it is an answer set of the test oracle on the test input as well. To

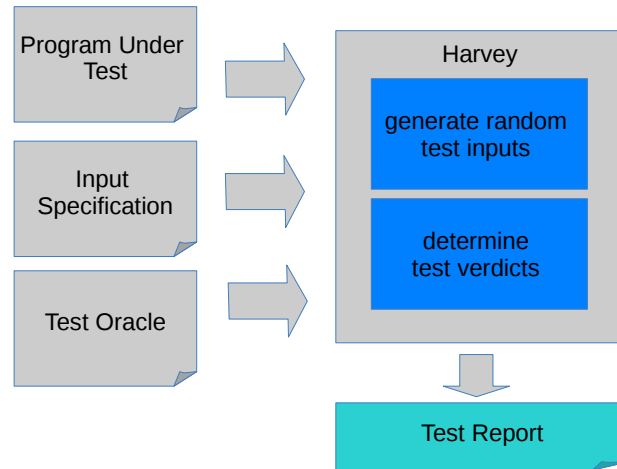


Figure 3.3: System architecture of Harvey.

this end, we temporally add the following rules to the test oracle:

$$passed \leftarrow a_1, \dots, a_m, \text{not } a_{m+1}, \dots, \text{not } a_n. \quad \perp \leftarrow \text{not } passed.$$

Here, $\{a_{m+1}, \dots, a_n\}$ is the set of atoms of the grounding of the program under test minus $\{a_1, \dots, a_m\}$. This guarantees that the solution of the program under test is included in the solutions of the test oracle for a particular input instance. Accordingly, if the test oracle joined with the input has no answer sets, the test failed, otherwise it passed. Note that we could replace, at least in principle, this part of Harvey with an external call to some other computational method to verify the correctness of a solution, therefore this testing framework can also be used when an ASP test oracle is not available.

3.3.3 The Tool Harvey

Harvey has been implemented in Python 3.4 and consists of two separate scripts, `xorsample.py` and `harvey.py`, where `xorsample.py` generates random inputs and `harvey.py` runs the tests using the inputs and the test oracle. An overview of the system architecture of Harvey is depicted in Figure 3.3. The system takes a program under test, a test oracle, and an input generator as input. Then it generates random inputs and determines test verdicts which are summarised as a short test report. Harvey can be downloaded from

<http://www.kr.tuwien.ac.at/research/systems/harvey/>.

The tool is executed from the command line as follows: Assume that the file `oracle.lp` contains the encoding of, let us say, the reviewer-assignment problem from the introduc-

tion, `inp-gen.lp` contains the input generator as discussed above, and `program.lp` is some ASP encoding of the reviewer-assignment problem that we want to test. To simply check whether solutions produced using `program.lp` are correct for 100 random inputs, we could run **Harvey** as follows:

```
python3.4 harvey.py --n=100 --g="-c n=5" --cf=inp-gen.lp
  --rf=oracle.lp --tf=program.lp
```

Option `--n` specifies the number of test inputs and option `--g` can be used to pass command-line arguments to **gringo**. In this example, we use it to set the constant `n` that is used in the input-generator to restrict the size of input instances. Likewise, `-c` can be used to pass command-line arguments to **clasp**. Options `--rf` and `--tf` are used to define the test oracle and the program under test, respectively.

If the run-time performance is poor, we could make use of the option `--q`, e.g., by invoking `--q=0.1`. This option defines the probability an atom is contained in an XOR constraint. It is better to have a less uniform distribution than to have no test inputs at all. Another option is to set a time limit in seconds for **clasp** with option `--t` (the default is no time limit). Also, sometimes the initial choice of the number of XOR constraints that are added is not optimal. If the number of solutions can be estimated to be 2^s , then `s` would be a good number of constraints to start with and can be set with option `-s`.

The method of testing a program against some reference implementation closely resembles random testing for conventional programming paradigms. Since our ASP test oracle can be used not only to check whether a given solution is correct but also to compute solutions itself, we can inverse the roles of the program under test and the test oracle: we would first compute an answer set of the test oracle and check whether it is also an answer set of the program under test. This means to check whether expected solutions are indeed produced by the program under test. While the conventional method relates to the correctness of a program, this method is rather concerned with its completeness. In **Harvey**, this can be done by simply swapping the test oracle and the program under test when running the tool.

For some problems, the distribution of test inputs that have solutions is very sparse. This means that for some programs, there are no answer sets for most input instances. Random testing would thus become uninteresting since the chances to get an input that yields some answer set can become virtually zero and any trivial inconsistent encoding would probably pass all tests. A simple remedy is to guide the generation of test inputs to areas of the test-input space where we have solutions. This can be done by simply adding the program under test (or the test oracle) entirely to the input generator. Generalising this idea, we can of course add arbitrary further constraints to the input generator if we want to focus on inputs with special properties. For example, in the reviewer assignment problem from the introduction, constraints could be added to get test inputs with many papers and few reviewers if we are expecting that such scenarios are particularly relevant.

To sum up, an ASP engineer has more options with *Harvey* than pure random testing by guiding the search for test inputs through constraints. The tool provides support when a program is modified and one needs a simple method to check whether emerging versions are still correct. Also, it can serve as a baseline for comparisons with other testing approaches.

3.4 Structure-Based Testing

An obvious desideratum in testing is to obtain small collections of test cases with a high potential to detect errors. In this section, we focus on test strategies that deduce test cases from the internal program structure. Hence, we follow principles of *structural testing*, also known as *white-box testing* [90, 118, 124].

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

Structural testing for logic programming has been considered already for Prolog [91, 14]. In particular, Belli and Jack [14] have developed coverage notions based on the computational model of Prolog: A program is executed by posing a query against it, and answers are computed using SLD resolution. Consequently, goals are understood as input while the output corresponds to specific information that is extracted from a program via resolution and unification. In contrast, output in ASP corresponds to the answer sets of a program and input corresponds to a set of facts. Hence, the setting in these works is quite incompatible with that in ASP and cannot be used directly.

Our goal is to develop a general approach to systematic structural testing of answer-set programs. In particular, we focus on the analysis of different coverage notions and propose methods for test automation in ASP. We evaluate the structure-based approach against simple random testing to evaluate the effectiveness of this more systematic method. In particular, we evaluate structure-based and random testing on the benchmarks chosen from ASP competitions [44, 27] and that have also been used in Section 3.2. Again, as in Section 3.2, the evaluation uses ideas from mutation analysis [42], where chosen instances are rendered incorrect by injecting faults that simulate minor programmer mistakes. Then, the two approaches are compared on the basis of how well they can catch such simple faults.

This section covers the following objectives:

- We define basic test coverage notions corresponding to path and branch coverage.
- We study results on the relations between the different test coverage notions and analyse their computational complexity.
- We lay down basic techniques for test automation using ASP itself, viz. for deciding test verdicts, determining coverage, and generating test cases with increasing coverage.
- We provide a quantitative comparison of structure-based and random testing using benchmarks from a previous ASP competition.

3.4.1 Coverage Metrics

Based on the basic terminology and notation for testing as presented in Section 3.1, we proceed with our central concepts to realise structural testing in ASP. We will develop our theory on structure-based testing for the fundamental fragment of normal logic programs without variables. Later, we will discuss how to apply those testing concepts for language extensions and more realistic solver dialects.

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

Coverage Functions

Traditionally, coverage is a relation between the input of test cases and particular syntactic entities (branching statements, execution paths, etc.) of a program. Coverage can thus be quantified by the sum of covered elements relative to the total number of (coverable) elements. The following central notion adopts conditions following Jack [91].

Definition 19. *A function γ from programs and collections of test inputs to the interval $[0, 1]$ is a coverage function if, for each program P and each collection $\mathcal{I} \subseteq 2^{\mathbb{I}P}$ of inputs of P ,*

- (i) $\gamma(\mathcal{I}, P) = 1$ if $\mathcal{I} = 2^{\mathbb{I}P}$, and
- (ii) $\gamma(\mathcal{I}', P) \leq \gamma(\mathcal{I}, P)$, for each collection $\mathcal{I}' \subseteq \mathcal{I}$ of inputs of P .

We say that a collection $\mathcal{I} \subseteq 2^{\mathbb{I}P}$ of inputs of a program P yields *total coverage* for P with respect to γ if $\gamma(\mathcal{I}, P) = 1$. Likewise, a test suite \mathcal{S} yields total coverage for P

with respect to γ if $\text{inp}(\mathcal{S})$ yields total coverage for P with respect to γ . Note that the exhaustive input collection $2^{\mathbb{I}P}$ trivially yields total coverage for any P with respect to γ . Thus, we call a coverage function γ *trivial* if, for any program P , only the input collection $2^{\mathbb{I}P}$ yields total coverage for γ .

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

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

Theorem 2. *No coverage function is both non-trivial and clairvoyant.*

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

Note that Theorem 2 is a very general result, not depending on the computational model of ASP, that carries over to structural testing in other programming paradigms when coverage functions of the kind as defined above are considered.

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

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

Based on this schema, we introduce the notion of *program coverage* in the next subsection that can be seen as a declarative analogue of path coverage for answer-set programs. In Section 3.4.1, we provide the notions of *rule*, *loop*, *definition*, and *component coverage* that amount to branch-coverage counterparts of program coverage. For each branch-like

coverage notion X , positive and negative X coverage are defined such that total X coverage holds exactly if both total positive and total negative X coverage do.

Path-like Test Coverage

We first define the notion of program coverage as an analogue of path coverage in conventional testing. Recall that, given a program P and an interpretation I , the set of *supporting rules of P with respect to I* , $\text{SuppR}(P, I)$, consists of all rules $r \in P$ whose body is true in I .

Definition 20. *Let P be a program, I an input of P , and $P' \subseteq P$. Then, I covers P' if $P' = \text{SuppR}(P, X)$, for some $X \in \text{AS}(P \cup I)$. Furthermore, a test case T for P covers P' if $\text{inp}(T)$ covers P' .*

Given a collection \mathcal{I} of inputs of P , by $\text{covered}_P(\mathcal{I}, P)$ we understand the number of subsets of P that are covered by some input $I \in \mathcal{I}$. It is easy to check that instantiating the basic coverage function schema (3.1) with $\text{covered}_P(\cdot, \cdot)$ (i.e., setting $X = P$) yields a coverage function. Accordingly, we call $\mathcal{C}_P(\mathcal{I}, P)$ the *program coverage of \mathcal{I} for P* .

Note that program coverage is a non-trivial coverage function. Thus, by Theorem 2, total program coverage for a program P and a test suite \mathcal{S} for P does not necessarily imply that P is correct whenever P passes \mathcal{S} .

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

Theorem 3. *Let P be a program and \mathcal{S} a test suite for P . Then, $\mathcal{C}_P(\text{inp}(\mathcal{S}), P) = 1$ implies that, for each input I of P and for each $X \in \text{AS}(P \cup I)$, \mathcal{S} contains a test case T such that for some $Y \in \text{AS}(P \cup \text{inp}(T))$, X and Y agree on the atoms that are defined by their supporting rules in P , respectively.*

Proof. Assume $\mathcal{C}_P(\text{inp}(\mathcal{S}), P) = 1$, let I be an arbitrary input of P , and let X be an arbitrary answer set of $P \cup I$ (the result holds trivially if such an answer set does not exist). Define $P' = \text{SuppR}(P, X)$. By assumption, there exists some test case $T \in \mathcal{S}$ and some $Y \in \text{AS}(P \cup \text{inp}(T))$ such that $P' = \text{SuppR}(P, Y)$. But $\text{SuppR}(P, X) = \text{SuppR}(P, Y)$ implies that X and Y have to agree on the atoms defined by their supporting rules in P and the result follows. \square

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

cases concerning the possible paths through a component, program coverage is exhaustive regarding the possible answer sets of a program (see Theorem 3).

Branch-like Test Coverage Notions

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

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

Definition 21. *Given a program P and an input I of P , a rule $r \in P$ is positively covered by I if $X \models B(r)$ for some answer set $X \in \text{AS}(P \cup I)$, and r is negatively covered by I if $X \not\models B(r)$ for some answer set $X \in \text{AS}(P \cup I)$. Furthermore, r is positively (resp., negatively) covered by a test case T for P if it is positively (resp., negatively) covered by $\text{inp}(T)$.*

Given a collection \mathcal{I} of inputs of P , by $\text{covered}_{R^+}(\mathcal{I}, P)$ (resp., $\text{covered}_{R^-}(\mathcal{I}, P)$) we understand the number of rules in P that are positively (resp., negatively) covered by some input $I \in \mathcal{I}$. Moreover,

$$\text{covered}_R(\mathcal{I}, P) = \text{covered}_{R^+}(\mathcal{I}, P) + \text{covered}_{R^-}(\mathcal{I}, P).$$

We define, *mutatis mutandis*, the values $\mathcal{C}_{R^+}(\mathcal{I}, P)$, $\mathcal{C}_{R^-}(\mathcal{I}, P)$, and $\mathcal{C}_R(\mathcal{I}, P)$ as instances of schema (3.1) as before (again giving rise to coverage functions) and refer to them as *positive rule coverage*, *negative rule coverage*, and *rule coverage* of \mathcal{I} for P , respectively.

Example 3. *Recall program P from Example 1:*

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

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

Note that it is not always possible to positively or negatively cover a rule by an input. Re-examining the rules from Example 3 reveals that no test case is able to positively cover the last rule. On the other hand, facts are rules which cannot be negatively covered. Furthermore, for the program P and an input collection \mathcal{I} consisting of I_1 and I_2 from

Example 3, we get a coverage of $\mathcal{C}_R(\mathcal{I}, P) = 1$. Thus, we obtain total rule coverage using two test cases only.

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

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

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

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

Loop Coverage While rule coverage is related to classical models of a program, the following coverage notion is concerned with loops which capture positive recursion between rules. The relevance of loops for ASP is well acknowledged in the literature [111, 107].

Definition 22. *Let P be a program and I an input of P . A loop L of P is*

- (i) *positively covered by I if there is an answer set $X \in \text{AS}(P \cup I)$ such that for every atom $a \in L$ there is a rule $r \in \text{SuppR}(P, X)$ that defines a ;*
- (ii) *negatively covered by I if for some $X \in \text{AS}(P \cup I)$ there is an $a \in L$ such that $\text{Def}_P(a) \neq \emptyset$ and no rule $r \in \text{SuppR}(P, X)$ defines a .*

As well, L is positively (resp., negatively) covered for a test case T if $\text{inp}(T)$ positively (resp., negatively) covers L .

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

Example 4. *Consider the program P consisting of rules*

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

with $\mathbb{I}_P = \{a, b, c, d\}$. Besides the singleton loops $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, and $\{e\}$, P has the loop $L = \{a, b\}$. It is easily verified that rule coverage can be achieved with inputs $\{b, c\}$ and $\{a, d\}$. Notably, neither of these test cases accounts for the mutual dependency of the first and the second rule due to the loop L . Loop coverage, however, would require an input, e.g., $\{a, c, d\}$, that explicitly covers L .

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

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

Proof. Assume total program coverage for P and \mathcal{I} , and let L be an arbitrary loop of P that can be positively (resp., negatively) covered by some input I of P . That is, for some $X \in \text{AS}(P \cup I)$, Condition (i) (resp., Condition (ii)) of Definition 22 holds. Total program coverage implies that for some input $I' \in \mathcal{I}$ and some answer set $X' \in \text{AS}(P \cup I')$, $\text{SuppR}(P, X) = \text{SuppR}(P, X')$. Hence, I' covers L positively (resp., negatively). Total loop coverage of P with respect to \mathcal{I} follows. \square

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

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

Definition 23. *Let P be a program and I an input of P . An atom a in P is*

- (i) positively covered by I if there is some $r \in \text{SuppR}(P, X)$ that defines a for an answer set $X \in \text{AS}(P \cup I)$;
- (ii) negatively covered by I if $\text{Def}_P(a) \neq \emptyset$ and no $r \in \text{SuppR}(P, X)$ defines a for an $X \in \text{AS}(P \cup I)$.

Moreover, a is positively (resp., negatively) covered by a test case T for P if it is positively (resp., negatively) covered by $\text{inp}(T)$.

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

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

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

Corollary 1. For each program P and each collection \mathcal{I} of inputs of P , total loop coverage implies total definition coverage for \mathcal{I} and P .

Rule coverage and definition coverage are related in the following way:

Theorem 6. For each program P and each collection \mathcal{I} of inputs of P , total positive rule coverage implies total positive definition coverage for \mathcal{I} and P .

Proof. Assume total positive rule coverage for \mathcal{I} and P . Let a be an arbitrary atom in P such that for some input I of P and some $X \in \text{AS}(P \cup I)$, there is a rule r in $\text{SuppR}(P, X)$ that defines a . By assumption, for some $I' \in \mathcal{I}$ and some $X' \in \text{AS}(P \cup I')$, $r \in \text{SuppR}(P, X')$. Thus I' covers the atom a as it is the head of r . Total positive definition coverage follows. \square

Note that the above theorem fails for negative rule coverage in general. A counterexample can be easily constructed.

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

Definition 24. Let P be a program and I an input of P . An SCC C of P is

- (i) positively covered by I if there is an answer set $X \in \text{AS}(P \cup I)$ such that for every node a in C , there is some $r \in \text{SuppR}(P, X)$ that defines a , and
- (ii) negatively covered by I if there is some $X \in \text{AS}(P \cup I)$ such that for every node a in C , $\text{Def}_P(a) \neq \emptyset$ and there is no $r \in \text{SuppR}(P, X)$ that defines a .

As for rules, loops, and definitions, we define the (*positive*, resp., *negative*) *component coverage* values $\mathcal{C}_{C^+}(\mathcal{I}, P)$, $\mathcal{C}_{C^-}(\mathcal{I}, P)$, and $\mathcal{C}_C(\mathcal{I}, P)$ using the general schema (3.1).

Note that every SCC corresponds to a maximal loop. We obtain the following corollary to Theorem 5:

Corollary 2. *For each program P and each set \mathcal{I} of inputs of P , total positive loop coverage implies total positive component coverage for \mathcal{I} and P .*

Negative component coverage requires that no node in the component is supported whereas only one unsupported atom is sufficient for negative loop coverage. Hence, total negative loop coverage does not guarantee total negative component coverage. Total program coverage, however, still implies total component coverage.

Theorem 7. *For each program P and each set \mathcal{I} of inputs of P , total program coverage implies total component coverage for \mathcal{I} and P .*

Proof. Assume total program coverage for P and \mathcal{I} . Total positive component coverage follows from Theorem 5 and Corollary 2. Let C be an arbitrary SSC of P . Assume that some input I of P negatively covers C , i.e., for some $X \in \text{AS}(P \cup I)$, it holds that for every node a in C , $\text{Def}_P(a) \neq \emptyset$ and there is no $r \in \text{SuppR}(P, X)$ that defines a . By assumption, for some input $I' \in \mathcal{I}$ and some $X' \in \text{AS}(P \cup I')$, $\text{SuppR}(P, X') = \text{SuppR}(P, X)$. But then, C is negatively covered by I' . Total component coverage for P and \mathcal{I} follows. \square

Recall that component coverage is designed as an approximation of loop coverage. This is formalised by the next result that follows immediately from the definitions of loop and component coverage.

Corollary 3. *Given a program P , an input I of P , and an SCC C of P with set L of nodes, positive component coverage for C by I implies positive loop coverage for each loop $L' \subseteq L$ of P by I . Likewise, negative component coverage for C implies negative loop coverage for each $L' \subseteq L$.*

Figure 3.4 summarises the central relations between the coverage metrics as discussed above. An unlabelled edge from notion X to notion Y means that for a program P and a collection \mathcal{I} of inputs of P , total X coverage for \mathcal{I} and P implies total Y coverage for \mathcal{I} and P . Moreover, an edge labelled with “+” means that total positive X coverage for \mathcal{I} and P implies total positive Y coverage for \mathcal{I} and P .

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

Example 6. *Assume that we want to encode the condition*

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

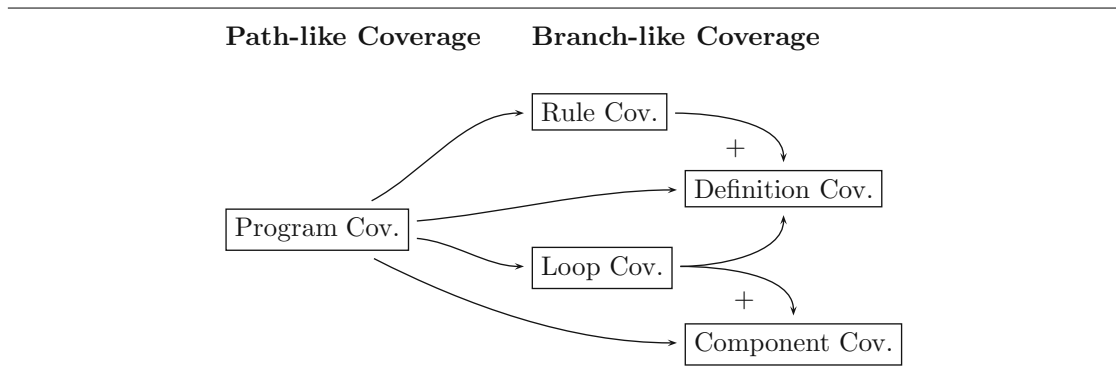


Figure 3.4: Relations between total coverage of different notions.

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

Consider the following incorrect realisation of the program P_n :

$$\begin{aligned} c &\leftarrow p_1, \dots, p_{n-1}, \\ f &\leftarrow c, p_n, \\ t &\leftarrow \text{not } f. \end{aligned}$$

The first rule states that the conjunction in φ_n is true iff all its operands are true. The second rule is a failed attempt to express that the implication in φ_n is false iff the antecedent is true and the conclusion is false: a negation preceding p_n was accidentally omitted by the programmer. The third rule states that $I \models \varphi_n$ iff $I \not\models \varphi_n$ does not hold.

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

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

3.4.2 Complexity Aspects

We now turn to complexity issues related to coverage problems and analyse the inherent complexity of relevant decision problems. Recall that D^P is the class of decision problems that can be characterised by a conjunction of an NP and an independent co-NP problem. First, we note that, in the general case, determining the test verdict for a test case is a challenging computational task:

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

Proof. Given a program P and a test case T , determining whether P is compliant with T , that is, whether $P[\text{inp}(T)] \subseteq \text{out}(T)$ holds, is co-NP-complete. This is shown by reducing the coNP-complete problem of deciding whether a program P yields no answer set to a compliance check. Define $T = \langle \emptyset, \emptyset \rangle$. Clearly, P has no answer sets iff $P[\text{inp}(T)] \subseteq \text{out}(T)$.

Analogously, the problem of deciding whether $\text{out}(T) \subseteq P[\text{inp}(T)]$ holds is NP-complete. We reduce the problem of deciding whether a program P yields an answer set, which is NP-complete, to the problem of deciding whether $\text{out}(T) \subseteq P[\text{inp}(T)]$ holds. To this end, define $T = \langle \emptyset, \{\emptyset\} \rangle$, and define the output signature of P as the empty set. Program P has an answer sets iff $\text{out}(T) \subseteq P[\text{inp}(T)]$ holds.

Program P passes T iff the two inclusion tests from the above hold. It follows that deciding whether P passes T is D^P -complete as any problem in D^P can be reduced to checking a test case by reducing the independent NP-complete and coNP-complete problems to the respective inclusion tests and assembling them into a single test case. \square

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

Proof. The following test decides whether I covers P' : Define X as the smallest set such that (i) $I \subseteq X$ and (ii) for each $r \in P'$, $H(r) \cup B^+(r) \subseteq X$. Set X can be constructed in polynomial time. Then, I covers P' iff $P' = \text{SuppR}(P, X)$ and $X \in \text{AS}(P \cup I)$. Both tests can be realised in polynomial time. \square

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

Proof. For membership, guess an interpretation X over $\text{HB}_P \cup I$ and check in polynomial time whether $X \in \text{AS}(P \cup I)$ and

- (i) $X \models B(r)$, for positive rule coverage,
- (ii) $\text{Def}_P(a) \cap \text{SuppR}(P, X) \neq \emptyset$, for positive definition coverage,
- (iii) for each $m \in L$, $\text{Def}_P(m) \cap \text{SuppR}(P, X) \neq \emptyset$, for positive loop coverage, and
- (iv) for each $n \in C$, $\text{Def}_P(n) \cap \text{SuppR}(P, X) \neq \emptyset$ for positive component coverage.

Membership for notions of negative coverage can be shown analogously: Guess an interpretation X over $\text{HB}_P \cup I$ and check in polynomial time whether $X \in \text{AS}(P \cup I)$ and

- (i) $X \not\models B(r)$, for negative rule coverage,
- (ii) $\text{Def}_P(a) \neq \emptyset$ and $\text{Def}_P(a) \cap \text{SuppR}(P, X) = \emptyset$, for negative definition coverage,
- (iii) for some $m \in L$, $\text{Def}_P(m) \neq \emptyset$ and $\text{Def}_P(m) \cap \text{SuppR}(P, X) = \emptyset$, for negative loop coverage,
- (iv) for each $n \in C$, $\text{Def}_P(n) \neq \emptyset$ and $\text{Def}_P(n) \cap \text{SuppR}(P, X) = \emptyset$ for negative component coverage.

Hardness follows by suitable reductions from the NP-complete problem of deciding whether a program P has some answer set. For positive coverage, define $P' = P \cup \{a \leftarrow\}$ for some globally fresh atom a , and let $I = \emptyset$. Note that a is, besides being an atom of P , also a loop and an SCC of P . If P has no answer set, no coverage notion can be satisfied. If P has an answer set X , $a \leftarrow \in \text{SuppR}(P', X)$ follows, and thus input I covers the rule $a \leftarrow$, the atom a , and the singleton loop and SCC $\{a\}$ under positive rule, definition, loop, and component coverage, respectively. Likewise, for negative coverage, define $P' = P \cup \{a \leftarrow a\}$ for some globally fresh atom a , and let $I = \emptyset$. As for positive coverage, if P has no answer set, no coverage notion can be satisfied. If P has an answer set X , $a \leftarrow a \notin \text{SuppR}(P', X)$. Hence, input I covers the rule $a \leftarrow a$, the atom a , and the singleton loop and SCC $\{a\}$ under negative rule, definition, loop, and component coverage, respectively. \square

Theorem 11. *Given a program P and a collection of its inputs \mathcal{I} , deciding whether*

- (i) \mathcal{I} yields total program coverage for P is co-NP-complete,
- (ii) \mathcal{I} yields total rule, definition, or component coverage for P is in Δ_2^P , and
- (iii) \mathcal{I} yields total loop coverage for P is in Π_2^P .

Proof. We give a schema for showing membership that applies to cases (i)–(iii). To decide the complementary problem of total coverage, for some ξ in P , where ξ is a set of rules, a rule, a definition, a loop, etc., and for some input $I \in 2^{\mathbb{I}_P \cap \text{HB}_P}$, check if both (a) I covers ξ and (b) no test input in \mathcal{I} covers ξ . For Item (i), we can non-deterministically guess an input of P as well as a set of rules from P and decide (a) and (b) in polynomial time. This implies co-NP-membership for deciding total program coverage. For Item (ii), $|P|$ is an upper bound for the number of rules and SCCs in a program, as well as of the number of atoms in a definition. Thus, the elements ξ of P can be enumerated in polynomial time. Problem (a) can be decided in NP and (b) can be decided in co-NP which implies Δ_2^P membership for total rule, definition, or component coverage. For Item (iii), an element ξ in P and an input for P can be non-deterministically guessed. Again, Problem (a) can be decided in NP and (b) can be decided in co-NP which implies Π_2^P membership for total loop coverage.

Hardness for item (i) can be shown by a reduction from program inconsistency to total program coverage. Let P be a program. Define program $P' = P \cup \{\leftarrow \text{not } q\}$ for some globally fresh atom q . The input alphabet of P only contains q , and $\mathcal{I} = \{\emptyset\}$. If P has no answer set, then neither P' nor $P' \cup \{q\}$ can have an answer set, and therefore no set of rules in P' can be covered by any input. Thus, \mathcal{I} yields total program coverage by definition. Observe that P and $P' \cup \{q\}$ have the same answer sets if we project away q in the comparison. It follows that if P has an answer set X , then the rules $\text{SuppR}(P, X)$ in P' are covered by the input $\{q\}$ of P' . However, due to the constraint $\leftarrow \text{not } q$ in P' , the only input \emptyset from \mathcal{I} cannot cover any set of rules as P' itself yields no answer set. Hence, \mathcal{I} does not achieve total program coverage. \square

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

3.4.3 Test Automation

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

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

$$ok \leftarrow a_1, \dots, a_n, \text{not } b_1, \dots, \text{not } b_m,$$

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

Checking Coverage Given a test input, evaluating coverage for the branch-like notions is NP-hard as shown in Theorem 10. However, we can use ASP techniques to check coverage. E.g., given a program P and an input I of P , we can determine whether a rule

$r \in P$ is covered positively (resp., negatively) by I by checking whether the atom sat (resp., the literal $\text{not } sat$) is a brave consequence of

$$P \cup I \cup \{sat \leftarrow B(r)\}.$$

Other coverage notions can be handled in a similar way.

Generating Covering Test Cases In testing, there are various strategies to obtain coverage. In randomised testing, the idea is to generate random test inputs and observe how different coverage metrics evolve when more tests are run. This approach can be used for testing ASP programs, too. When using randomly generated test inputs, it is typically possible to increase coverage only up to a limited degree. For more intelligent goal directed testing, techniques for generating test inputs guaranteed to increase coverage (up to total coverage) are needed. ASP techniques can also be used for this task, for example, by adding to a program P the set of rules

$$C(\mathbb{I}_P) = \{a \leftarrow \text{not } a' \mid a \in \mathbb{I}_P\} \cup \{a' \leftarrow \text{not } a \mid a \in \mathbb{I}_P\}.$$

Then, test inputs covering a given element (a rule, a loop, etc.) can be obtained using brave reasoning techniques as described above provided that witnessing input for a brave consequence is returned by the reasoning engine. For example, a test input covering a rule r positively can be obtained (if it exists) by checking whether the atom sat is a brave consequence of

$$P \cup C(\mathbb{I}_P) \cup \{sat \leftarrow B(r)\},$$

and extracting a test input from a witnessing answer set (provided that the solver answering this query is able to do this and such an answer set exists). A simple way of implementing such a brave reasoning engine is to use an ASP solver to look for an answer set of the program

$$P \cup C(\mathbb{I}_P) \cup \{sat \leftarrow B(r)\} \cup \{\leftarrow \text{not } sat\}.$$

3.4.4 Experimental Evaluation

We next address the question how the structure-based approach to test input generation compares to random test input generation. To this end, we evaluate structure-based and random testing using benchmark problems that have been used in ASP competitions [44, 27]. The evaluation is based on mutation analysis as discussed in Section 3.2, i.e., program instances are rendered incorrect by injecting faults according to a mutation model. Then, the two approaches are compared on the basis of how well they can catch such faults.

The class of normal logic programs is simple enough to provide a solid basis for a theoretical study on the subject of structure-based testing. For a practical evaluation however, we have to consider some language extensions. In particular, we deal with a class of logic programs that corresponds to the ground fragment of the **gringo** input language [73, 70], which extends normal programs by aggregate atoms (cf. Chapter 2).

We thus slightly modify our definition of the set of defining rule of an atom to take choice rules into account: For an ordinary atom a , $\text{Def}_P(a)$ is the set of all rules in P where a occurs *positively* in the rule head, possibly within an aggregate atom. Also, the practicably relevant rule type of a constraint needs special treatment: A constraint $c \in P$ is *covered* by I if the body of c is true in some answer set $X \in \text{AS}((P \setminus \{c\}) \cup I)$, i.e., c is temporarily removed from P . The intuition behind this idea is that, to test a constraint, we would like to use inputs that satisfy the body of that constraint. Note that under standard rule coverage, constraints cannot be positively covered because no answer set can satisfy the constraint's body.

Experimental Setup

Our experimental setup involves six steps:

1. selecting some benchmark programs;
2. grounding them to obtain reference programs;
3. injecting faults into the reference programs to generate mutants;
4. generating random test suites for mutants;
5. generating structure-based test suites for mutants; and
6. comparing the resulting test suites.

We describe these steps in more detail in what follows.

Step 1: Selection of benchmark instances. We consider programs from the benchmark collection of the second ASP competition [44] for our experiments. In particular, we selected the problems MAZEGENERATION, SOLITAIRE, GRAPHPARTITIONING, and REACHABILITY to represent typical program structures in ASP (cf. Appendix A for details on the programs):

- MAZEGENERATION is about generating a maze on a two-dimensional grid. The problem encoding is characteristic for the guess-and-check paradigm in ASP and involves inductive definitions to express the reachability conditions. A distinguishing feature of this problem is that solutions are partially fixed in the input and completed in the output.
- SOLITAIRE is a simple planning game that is played on a grid with 33 cells. The structure of the encoding is slightly simpler than for MAZEGENERATION since no positive recursion is involved.
- GRAPHPARTITIONING is a graph-theoretical problem about partitioning the nodes of a weighted graph subject to certain conditions. It is representative for hard problems on linked data structures that involve integer calculations.

- Finally, the objective of the REACHABILITY problem is to exhibit a path between two dedicated nodes in a directed graph. REACHABILITY is a representative for problems solvable in polynomial time. The central aspect is to efficiently compute the transitive closure of the edge relation of the input graph.

Step 2: Grounding. Since our testing approach is defined for propositional programs, we need to ground the programs from the first step. Uniform ASP encodings usually involve a natural parameter that allows one to scale the size of ground instances:

- For MAZEGENERATION, we used a bound on the size of the grid of 7×7 to obtain a finite grounding.
- Since SOLITAIRE is a planning problem, we parameterized it by setting the upper bound on the lengths of plans to 4.
- For GRAPHPARTITIONING, since it takes graphs as input, one natural scaling parameter is a bound on the number of nodes which we fixed to 7. Furthermore, we fixed a bound of 3 on the maximal number of partitions and a bound of 20 on maximal weights assigned to edges.
- For REACHABILITY, we fixed an upper bound of 6 on the number of nodes as the scaling parameter.

We refer to the ground program obtained in this step as *reference program* since it later serves as a *test oracle*, i.e., as a method to determine the correct output given some input.

Step 3: Fault injection. Based on grounded encodings from the ASP competition, we follow the approach outlined in Section 3.2 that is inspired by mutation analysis. In particular, we generate different classes of mutants, one class for each mutation operation. To this end, we consider

- (i) deleting a single body literal,
- (ii) deleting a single rule of a program,
- (iii) swapping the polarity of literals,
- (iv) increasing or decreasing the bound of an aggregate by one, and
- (v) replacing an atom by another atom appearing in the program.

For each grounded benchmark encoding, we generate one class of mutants for each mutation operation. The bound modification was not applied to MAZEGENERATION and REACHABILITY instances which do not involve aggregate atoms nor bounds. Each mutant was generated by applying a mutation operation precisely once. Each class consists of 100 mutants which are publicly available.⁶ We used dedicated equivalence tests [95] to eliminate equivalent mutants that cannot be distinguished from the reference program by any test input.

⁶<http://www.kr.tuwien.ac.at/research/projects/mmdasp/mutant.tgz>.

Step 4: Random test suite. For each benchmark problem in our experiments, we formalised the preconditions as a simple ASP program following the idea of defining input generators from Section 3.1 (cf. also Appendix A). Any answer set of such a program constitutes an admissible input, i.e., one which is consistent with the preconditions of the encoding. Hence, the task of generating admissible random inputs can be reduced to computing uniformly-distributed answer sets of logic programs (random testing is discussed in depth in Section 3.3).

Step 5: Structure-based test suite. We next review our approach to generate test inputs that yield total coverage. First, we recapitulate how we can use ASP itself to generate covering inputs.

Let P be a program with input signature \mathbb{I}_P . We define the *input generator* for P , denoted $\text{IG}(P)$, as the program

$$\{a' \leftarrow \text{not } a''; a'' \leftarrow \text{not } a'; a \leftarrow a' \mid a \in \mathbb{I}_P\},$$

where all primed and double primed atoms do not occur anywhere else.

A program P is labeled for rule coverage as the program

$$\text{LR}(P) = \{H(r) \leftarrow r'; r' \leftarrow B(r) \mid r \in P\},$$

where r' is a globally new label for each $r \in P$. To cover individual rules either positively or negatively, we use programs

$$P_r^+ = \text{LR}(P) \cup \{\leftarrow \text{not } r'\} \quad \text{and} \quad P_r^- = \text{LR}(P) \cup \{\leftarrow r'\},$$

respectively. Then, the inputs for P that cover r positively and negatively are in a one-to-one correspondence with the respective sets

$$\{X \cap \mathbb{I}_P \mid X \in \text{AS}(\text{IG}(P) \cup P_r^+)\} \quad \text{and} \quad \{X \cap \mathbb{I}_P \mid X \in \text{AS}(\text{IG}(P) \cup P_r^-)\}$$

of inputs. Similar reductions are devised for the other coverage notions. The preconditions of each benchmark program P form a set C of constraints which can be incorporated into P_r^+ and P_r^- , thus restricting their answer sets to admissible test inputs of P .

Given a mutated program P , we generate test suites that yield total rule coverage for P using the reduction P_r^+ . We first generate a test suite \mathcal{S} that obtains total positive rule coverage for P . To this end, we pick a rule $r \in P$ not yet positively covered and check whether there is an input and a corresponding answer set X that positively covers r . If such an input exists, r is marked as positively covered, if not, r is marked as positively failed, meaning that no inputs exist that cover r positively. Then, some simple bookkeeping takes place: any rule that is positively, resp., negatively, covered by the answer set X is marked accordingly. The procedure iterates until all rules are marked as positively covered or failed. A similar procedure

is then applied to extend \mathcal{S} to obtain total negative rule coverage. The method for the other notions of coverage is analogous.

Note that for a program P , there is a test suite that obtains total rule, definition, or constraint coverage whose size is bounded by the number of rules in P . Although our method does not guarantee minimality of a test suite, it always produces one within this bound.

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

Results and Discussion

Our test results in terms of catching rates for the MAZEGENERATION and REACHABILITY benchmarks are summarised in Table 3.8. The mutation operations studied were atom replacement (AR), literal deletion (LD), rule deletion (RD), and literal polarity swapping (PS). Besides definition, rule, and constraint coverage, we considered the combinations of definition and rule coverage with constraint coverage. The column “size of test suite” in the tables refers to the size of the generated test suites for each mutant class. For the coverage-based suites, the numbers are averages over all test suites in a class.

For MAZEGENERATION, random testing only obtains very low catching rates, while systematic testing yields total rates of up to 0.9 using a significantly smaller test suite. One explanation for the poor performance of random testing is that the probability that a random test input yields any answer set for a mutant is quite low. Inputs that yield no or few outputs for a mutant seem to have a low potential for fault detection. The situation is different for the REACHABILITY benchmark: random testing performs slightly better than systematic testing. This is mainly explained by the higher number of test inputs used for random testing. The conclusion is that the considered structural information seems to provide no additional advantage compared to randomly picking inputs for this benchmark. As regards structural testing, the best detection rate is obtained by combining rule and constraint coverage. However, this comes at a cost of larger test suites compared to the other coverage notions.

We conducted similar experiments using the GRAPHPARTITIONING and SOLITAIRE benchmarks. Interestingly, it turned out that testing seems to be trivial in these cases. Since both random and structural testing achieved a constant catching rate of 1.00 for all mutant classes, we omit the respective tables. In fact, almost any test input catches all mutants for these programs. This nicely illustrates how the non-determinism involved in answer-set semantics affects testing. While in conventional testing only a single program execution is associated with a particular test input, the situation in ASP is different. Given a test input, the output of a uniform encoding can consist of a vast number of

Table 3.8: Catching rates for the MAZEGENERATION and REACHABILITY benchmark.

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

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

answer sets, each of which potentially revealing a fault. For example, typical random inputs for SOLITAIRE yield tens of thousands of answer sets which gives a large amount of information for testing. In other words, there is quite a chance that some effect of a fault shows up in at least one answer set.

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

An Annotation Language for ASP

This chapter deals with the interrelation of testing and support for systematic program development in ASP.

In the first section, we present LANA, an annotation language for ASP. This language can be used to structure a program into blocks and to declare language elements like predicates with type information, input and output signatures, pre- and postconditions, test cases, etc. Annotations do not interfere with the languages of answer-set solvers as they have the form of program comments. They can be interpreted by tools to support the development process, to automatically test and verify programs, and to increase maintainability by enhancing program documentation.

In Section 4.2, we present ASPDoc, a tool that interprets LANA annotations and generates a corresponding HTML documentation, similar to JavaDoc, while in Section 4.3, we describe ASPUnit, a tool that executes test cases formulated in LANA.

4.1 Basic Elements of the Annotation Language LANA

In this section, we propose to augment programs with additional meta-information to facilitate the ASP development process. To this end, we devised a dedicated annotation language, LANA, standing for “Language for ANnotating Answer-set programs”, that specifies specially formatted program comments. This meta-information is invisible to an ASP solver—therefore not altering the semantics of the program—but different tools may interpret and use the annotated information to various ends like testing, documentation, verification, code completion, or other development support.

4.1.1 Overview

One particular and quite central feature of LANA is grouping rules that are related in meaning into coherent blocks. Although different notions of modularity have already been proposed for ASP in the literature [24, 53, 74, 6, 96], a strict concept of a program module can sometimes be a tight corset. In particular, notions of modularity in ASP often come with their own semantics and different kinds of constraints need to be satisfied. For example, *DLP-functions* [96] require that their input and output signatures are disjoint and two DLP-functions need to satisfy certain syntactic constraints in order to be composable. On the other hand, *lp-functions* are a modular approach to build a logic program from its specification [74]. That is, an *lp-function* is used to realise some functional specification that relates input and output relations for some domain by means of a logic program. The kind of grouping that we are proposing has, however, no semantical ramifications other than documenting that some rules belong together. Nevertheless, the benefit is that we add some extra structure to a program, improving the clarity and coherence of the program parts, which in turn can be used by other tools for, e.g., unit testing [12].

While unit testing is an integral element in software development using common languages like Java or C, this approach has been addressed in ASP only by a few researchers [64, 4]. We provide means to formulate unit tests for single blocks using LANA, allowing for easy regression testing. After rules are grouped into blocks, we may use further annotations to declare respective input and output signatures, which are also useful for testing and verification. Furthermore, we can declare the names and arities of predicates that are used within a block. This information can be exploited by, e.g., an integrated development environment (IDE) for syntax checking and code completion features while a user is writing a program. Besides names, description, and arities of predicates, one can also specify the domains of term arguments of a predicate using respective language features for declaring types. This information can be used for automated detection of type violations. These declarations have the potential to eliminate the source for quite common programmer errors with only little extra cost.

For verification purposes, our annotation language can be used to specify assertions like pre- and postconditions for blocks. Preconditions are expected to hold for any input of a block, while postconditions have to hold for any output. Together, they can be used to verify correctness of an ASP encoding with respect to such assertions.

In the following sections, we are going to introduce LANA and explain its basic features using a running example. Furthermore, we describe two tools that exploit program annotations to aid in development support. The first tool is ASPDoc, a JavaDoc [97] inspired tool, which takes an answer-set program, interprets the meta-comments, and automatically generates an HTML file documenting the program. The other tool is ASPUnit, an implementation of a unit-testing framework in the spirit of JUnit [98] based on the structural annotations found in a program. This framework allows to formulate unit tests for individual program blocks, to execute them, and to monitor test runs.

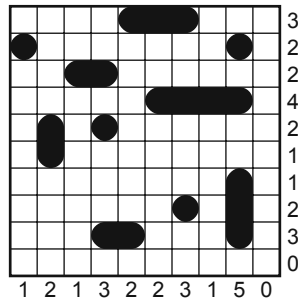


Figure 4.1: Solved instance of a Battleship puzzle.

4.1.2 Annotating Answer-Set Programs in LANA

In this section, we describe our annotation language LANA. An overview of most language elements of LANA is depicted in Table 4.1; Table 4.2 gives a summary of the remaining language elements related to testing in LANA. We make use of a simple answer-set program to illustrate all the language features in a step-by-step fashion. In particular, we use an encoding of the *Battleship* puzzle. A solved instance of Battleship appears in Fig. 4.1. In Battleship, a group of ships is hidden on a grid and one has to find the positions of each. The fleet consists of one four-squares long battleship, two three-squares long cruisers, three two-squares long destroyers, and four one-square long submarines. Each ship is placed horizontally or vertically on the grid such that no ship is touching any other ship (not even diagonally). To provide hints, some squares may show parts of a ship or water. Moreover, a number besides each row and each column indicates how many squares in that row or column are occupied by ship parts. A solution of a puzzle is a configuration of all the ships that is consistent with the initially given hints.

Assume that a puzzle instance is defined in terms of facts `water/2` and `ship/2` for specifying which squares contain water or parts of a ship, respectively. The facts `rowHint/2` and `colHint/2` determine the numbers associated with each row and each column. Problem solutions are represented by facts `ship(W, X, Y, Z)` expressing that a ship is occupying the squares from position (W, X) to (Y, Z) .

Blocks

In general, all keywords of our annotation language start with the symbol `@`. A central feature of LANA is to group rules together. This is done using the `@block` keyword. To specify that, we are going to define some rules that encode solutions to the Battleship puzzle, we declare a block with the name `Battleship` as follows:

```

%** @block Battleship { *%

% encoding of the Battleship puzzle

%** } *%

```

Table 4.1: Overview of LANA based on BNF.

Element	Definition	Informal Description
<i>block</i> <i>element</i>	<i>block</i> <i>atom</i> <i>term</i> <i>input</i> <i>signature</i> <i>output signature</i> <i>precondition</i> <i>postcondition</i>	LANA elements related to blocks.
<i>block</i>	“@block” <i>name</i> “{” [<i>description</i>] { <i>block element</i> } [<i>ASP code</i>] “}”	Groups ASP rules into coherent parts.
<i>atom</i>	“@atom” <i>name</i> (“ <i>termList</i> “)” [<i>description</i>]	Defines a predicate; <i>termList</i> are the predicate’s arguments.
<i>term</i>	“@term” <i>name</i> [<i>description</i>] [<i>type</i>]	Declares a term from some <i>atom</i> term list, its meaning, and type information.
<i>type</i>	“@from” <i>groundTerms</i> “@with” <i>ruleBody</i> “@samerangeas” <i>term</i>	Type of a term is defined by a list of ground terms, the terms satisfying <i>ruleBody</i> , or as the type of another term.
<i>input</i> <i>signature</i>	“@input” <i>inputPredicates</i> “@requires” <i>inputPredicates</i>	Declares input predicates of a block as a list of name/arity pairs.
<i>output</i> <i>signature</i>	“@output” <i>outputPredicates</i> “@defines” <i>outputPredicates</i>	Declares output predicates of a block as a list of name/arity pairs.
<i>assertion</i>	“@assert” <i>name</i> “{” [<i>description</i>] <i>assertspec</i> “}”	A logical condition for answer sets.
<i>pre-</i> <i>condition</i>	“@precon” <i>name</i> “{” [<i>description</i>] <i>assertSpec</i> “}”	A logical condition for the inputs of a block.
<i>post-</i> <i>condition</i>	“@postcon” <i>name</i> “{” [<i>description</i>] <i>assertspec</i> “}”	A logical condition for the answer sets of a block.
<i>assertspec</i>	(“@always” “@never”) <i>atmList</i> [<i>embASPcode</i>]	The testmode for assertions, preconditions and postconditions; <i>embASPcode</i> is code within the LANA comment environment.

The annotation @block is followed by the name of the block and the opening bracket “{”. Everything between “{” and the closing bracket “}” now belongs to the block Battleship. Blocks can be nested but they must not overlap.

Note that every annotation has the form of an ASP comment. ASP block-comments can be used instead of starting every single line with “%” when an ASP solver, in particular its grounding component, supports this feature. To distinguish annotations from ordinary (block-)comments, an additional star, “*”, is always placed after “%*” at the beginning.

Predicate Signatures

LANA allows to declare the names and arities of used predicates. Often, predicates play different roles in an encoding, and it can make sense to explicitly distinguish between these roles. In particular, it can make sense to document which are the relations that are defined by the rules of a block (those will usually appear in the heads of some rules) and which are the relations that are expected to be already defined by some other rules (the required predicates will usually appear in some rule bodies). This distinction is

Table 4.2: Test case specification in LANA.

Element	Definition	Informal Description
<i>test case</i>	“@testcase” <i>name</i> [<i>description</i>] “@scope” <i>blocks</i> <i>testcond</i> [<i>ASP code</i>]	A test case for the blocks from <i>blocks</i> passes if the blocks joined with <i>ASP code</i> satisfy all specified test conditions.
<i>testcond</i>	“@testhasanswerset” “@testnoanswerset” “@testatoms” <i>atmList mode</i>	A test condition holds if one, resp., no, answer set is found, or all ground atoms in <i>atmList</i> are entailed according to <i>mode</i> .
<i>mode</i>	“@trueinall” “@trueinatleast” <i>n</i> “@trueinatmost” <i>n</i> “@falseinall” “@falseinatleast” <i>n</i> “@falseinatmost” <i>n</i>	Mode of entailment of a test condition, i.e., if test atoms are true, resp., false, in all, at most <i>n</i> , or at least <i>n</i> answer-sets.

closely related to intensional and extensional predicates in a database setting. If a block is regarded as a declarative problem-solving module, the predicates that are required will usually encode problem instances, and respective predicates form the *input signature* of a block. Accordingly, the relations that are defined form the *output signature* of a block. We note, however, that input and output signatures might overlap in a declarative setting.

We proceed by declaring the predicate names as well as the input and output signatures of our encoding as described above. Within block `Battleship`, we add the following:

```

%**
@atom water(X,Y)
there is no ship at position (X,Y)

@atom ship(X,Y)
a ship is occupying position (X,Y)

@atom rowHint(X,H)
in row X, H squares are occupied

@atom colHint(Y,H)
in column Y, H squares are occupied

@atom ship(X1,Y1,X2,Y2)
a ship is occupying the squares from (X1,Y1) to (X2,Y2)

@input water/2, ship/2, rowHint/2, colHint/2

@output ship/4
*%

```

We use “@atom” to introduce the name of a predicate along with its arity and some information describing its intended use, and we use “@input” and “@output” to determine which predicate symbols are used to represent the input for the block and which ones correspond to the output. For input and output signatures, we also have to explicitly give the arity of the involved predicates. This is needed to disambiguate between predicates having the same name but a different number of arguments, as ship in our running example. Note that annotations are optional, declarations are not enforced. We stress that declaring input and output signatures should be done only if this is beneficial for the development process and is consistent with the declarative reading of a program. Input and output are terms that are quite commonly used in declarative programming—e.g., all problem specifications for the benchmark problems used for the ASP competitions explicitly define input and output signatures. However, if rules are seen as more general definitions of relations of some problem domain, it might be more appropriate to use @defines to declare the relations that are defined by a block of rules and @requires to declare the input signature of a block. Again, such annotations are not mandatory. The more information is made explicit, the more it can be used by tools that interpret such information to the benefit of the developer.

4.1.3 Types

Regarding the declaration of predicates, we can provide information about the types of its term arguments. Assume that row and column positions are specified by ascending integers starting from 1. To make this assumption explicit, we add the following lines to the block:

```
%**
@term X, Y
@from 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
X (Y) is a row (column) index ranging from 1 to 10
*%
```

Here, we use @term to declare variable names and @from to specify the type of a variable by means of a non-empty comma-separated list of admissible ground terms. As an alternative to @from, we can use @with followed by a rule body:

```
%** @with integer(#V), #V>1, #V<10 *%
integer(1..1000).
```

The type of X and Y is now determined by the ground substitutions for the reserved symbol #V that satisfy the rule body given after @with. Here, “integer(1..1000).” is regular ASP code for defining facts that encode the integer range that we are considering. Furthermore, to express that variables are of the same type as ones already known, we can use @samerangeas, as illustrated next:

```

%**
@term X1, Y1, X2, Y2
  further row and column indices
@samerangeas X
*%

```

Thus, any of $X1$, $Y1$, $X2$, and $Y2$ is of the same type as X .

As mentioned previously, blocks can be nested. To proceed with our Battleship encoding, we add a block of rules within block `Battleship` for guessing solution candidates according to the usual guess-and-check paradigm:

```

%**
@block Guess {
  guess a configuration of battleships on the grid
*%
r(1..10). c(1..10).
{ship(X1,Y1,X2,Y2) : r(X1) : c(Y1) : r(X2) : c(Y2) : X2>=X1 : Y2>=Y1}.
:- ship(X1,Y1,X2,Y2), X1 != X2, Y1 != Y2.
%** } *%

```

Note that in block `Guess`, all declarations for predicates and terms are inherited from the enclosing block `Battleship`. One can proceed in a similar way to define blocks for constraints (along with auxiliary definitions) to prune away unwanted solution candidates to complete the encoding.

Assertions

Towards testing and the verification of programs, LANA allows the formulation of general assertions, as well as pre- and postconditions for blocks. A precondition is a logical condition that is assumed to hold for any input while a postcondition has to hold on any output of a block. Together, pre- and postconditions can be regarded as a *contract*: it is the responsibility of any input provider that a block's preconditions are satisfied, and it is the responsibility of the rules in the respective block that its postconditions are satisfied. Both pre- and postconditions are formulated in ASP itself; they are placed within the block they belong to. As an illustration, we formulate as a precondition of our Battleship encoding that no square shows both water and parts of a ship jointly as follows:

```

%**
@precon Excl {
  no square shows both water and a part of a ship
@never clash
clash :- water(X,Y), ship(X,Y).
}
*%

```

In general, a precondition is introduced with the keyword `@precon` followed by a name for the condition. The actual content is then written enclosed between “{” and “}”, similar to the definition of blocks. An optional description follows. Then, we have to specify a non-empty list of ground atoms after the keyword `@never` or `@always`. After that, we add some ASP rules that define the before-mentioned ground atoms. The intended semantics is as follows: Some input, i.e., a set of facts over a block’s input signature, passes a precondition if that input combined with the rules of the precondition cautiously entails all the ground atoms after `@always` and the negation of the atoms after `@never`.

Postconditions are expressed analogously to preconditions. To say that battleships must not be longer than four squares, we add the following code to our Battleship block:

```
%**
@postcon Overlength {
  battleships are never longer than four squares
  @never ov
  ov :- ship(X1,Y1,X2,Y2),L=X2-X1,L>4.
  ov :- ship(X1,Y1,X2,Y2),L=Y2-Y1,L>4.
}
*%
```

A block and an input for a block satisfy a postcondition if the block joined with the input and the rules of the postcondition cautiously entails all the ground atoms after `@always` and the negation of the atoms after `@never`.

Having pre- and postconditions explicitly formalised offers various ways to support the development process. For one thing, they can be used to automatically generate input instances for testing purposes. This can be automated for a systematic testing of a block, including random testing and structure-based testing as discussed in Sections 3.3 and 3.4. Towards program verification, one can check whether a block passes its postconditions for any admissible input, at least from some fixed small scope, i.e., involving only a bounded number of constant symbols. The effectiveness of exhaustive testing with respect to a small scope has been demonstrated in Section 3.2. Though they are formulated in ASP itself and thus tend to duplicate some code from within a block, pre- and postconditions are especially of value if the rules in a block are changed, e.g., to optimise an encoding, and one wants to be sure that the changes did not render the program incorrect. This can be done, e.g., by searching for inputs within some small scope that violate some postcondition of the optimised program, provided respective tool support is available.

We note that pre- and postconditions make only sense in a setting where we can also distinguish between input and output relations. If this is not the case and one wants to formulate general assertions on the answer sets of a program, LANA allows to define them using `@assert`. Semantically, an assert statement is a postcondition of the whole program.

Unit Tests

Though pre- and postconditions allow to partially verify program correctness, LANA also supports a light-weight form of program verification that is inspired by unit testing. A unit test in procedural languages is commonly a test for an individual function or procedure. While in a related approach for unit testing answer-set programs [64], the scope of a test is defined in terms of sets of rules, unit tests are formulated for blocks or sets of blocks in our setting. This related work was extended recently by introducing a specification language for unit testing that also took inspiration from LANA as it is annotation-based and allows the development of test cases within ASP code [4].

To check whether the guessing part of our running example generates solution candidates where one ship occupies precisely the first four horizontal squares of the field, we could formulate a unit test as follows:

```
%**
@testcase ShipTopLeftCorner
  a ship is horizontally placed at the top-left corner
@scope Guess
@testatoms goalShip @trueinatleast 1
*%
goalShip :- ship(1,1,1,4).
```

A unit test starts with the reserved word `@testcase` followed by a name. Then, a short description of the purpose of the unit test may be given as a comment. After `@scope`, a list of block names is expected. In the above example, we used `@testatoms` to declare that `goalShip` is an atom that has to be true in at least one answer set (`@trueinatleast 1`) of block `Guess` joined with the subsequent rule that defines `goalShip`. Instead of or additional to `@trueinatleast n` , a tester might use `@trueinatmost m` , `trueinall`, `falseinatleast p` , `falseinatmost q` , and `falseinall`, where m , n , p , and q are positive integers. Also, instead of `@testatoms`, one may use `@testhasanswerset` or `@testnoanswerset` to express that at least one or no answer set is expected, respectively.

The semantics of a unit test is as follows. A test case passes iff the answer sets of the rules of the test case combined with all the blocks specified after `@scope` satisfy the testing conditions expressed using any of the expressions `@testatoms`, `@testhasanswerset`, and `@testnoanswerset`. For example, to additionally test that a ship is never placed diagonally on the field, one could formulate a further test case:

```
%**
@testcase NoDiagonalShips
  ships are never placed diagonally on the field
@scope Guess
@testatoms forbiddenShip @falseinall
*%
```

```
forbiddenShip :- ship(1,1,3,3).
```

Of course, this test case can only guarantee that one particular ship is not placed diagonally at some particular position. However, this distinguishes test cases from more general assertions like postconditions. To generalise the above test case to arbitrary ships, we would rather use a postcondition. Typically, test cases represent concrete situations by means of facts that can be easily verified by a user and document individual situations that are allowed or forbidden. They cannot, however, guarantee correctness of an encoding but only increase our confidence regarding its functionality.

Unit testing is a convenient way to test properties of individual blocks of an ASP encoding. Furthermore, they can be used to iteratively develop programs in a test-driven fashion. In test-driven development, unit tests are formulated before the code is written. First, a unit test for a single property of the block that we want to develop is specified. Then, it is checked whether the test case fails for the program under development. If this is the case, the block is extended by the necessary rules to make the failed test case pass. After the code is refactored towards efficiency, readability, etc., and after it is verified that all test cases still pass, the next property is addressed by formulating a respective unit test. This continues until the program is completely implemented. For illustration, the unit tests `ShipTopLeftCorner` and `NoDiagonalShips` will pass for the current state of the Battleship program. However, if we want to implement the property that ships must not touch each other, we could specify the following test case (which will currently fail):

```
***
@testcase TouchingShips
two ships must not touch each other
@scope Guess,Touch
@testatoms forbiddenShip @falseinall
*%
forbiddenShip :- ship(1,1,1,2), ship(1,2,1,4).
```

Then, we would proceed to implement a block `Touch` with a constraint that forbids answer sets with ships that are touching each other and check whether the new test case and the old ones pass.

4.2 ASPDoc

ASPDoc is a command-line tool that interprets meta-information given in an answer-set program and generates a corresponding HTML documentation file similar to, e.g., JavaDoc for Java programs. Information regarding block structure, input and output signatures, used predicate symbols, etc. is clearly arranged so that the answer-set program can easily be understood, used, or extended by other developers. Such documentation features are especially useful to make ASP problem-solving libraries, i.e., collections of

Table 4.3: Command-line options for ASPDoc.

Option	Description
<code>-o=<i>path</i></code>	set output directory to <i>path</i>
<code>-HA</code>	show hidden atoms
<code>-ha</code>	do not show hidden atoms
<code>-S</code>	include ASP code in the HTML document
<code>-s</code>	do not include ASP code in the HTML document
<code>-A</code>	include LANA code in generated ASP code
<code>-a</code>	do not include LANA code in generated ASP code
<code>-potassco, -p</code>	input language is that of gringo
<code>-dlv, -d</code>	input language is that of DLV
<code>-help, -h</code>	print usage information

ASP encodings that can be used as building blocks for larger programs, accessible to developers.

The tool is developed in Java; an executable JAR file is available at

`students.sabanciuniv.edu/dgkisa/aspdoc-aspunit.`

Assume that the source code of our running example is stored in the file `battleship.lp`. A corresponding HTML documentation can be created as follows:

```
java -jar aspdoc.jar -p battleship.lp.
```

Different HTML documents are created with `index.html` as the usual entry point. Here, option `-p`, or `-potassco`, is used to tell ASPDoc that the answer-set program is written using **gringo** syntax. For **DLV**, option `-d` or `-dlv` can be used instead. Furthermore, an output directory *d* can be specified with option `-o=d`, and help on available options can be obtained with the option `-h`. A summary of ASPDoc options is given in Table 4.3.

A screenshot of the documentation for the Battleship example presented above is given in Figure 4.2. The documentation of the complete encoding can be found at

`www.kr.tuwien.ac.at/research/projects/mmdasp/battleship.`

The document contains descriptions of all the blocks of the answer-set program, where sub blocks are indented relative to their parent blocks. To provide an overview, a summary of the block structure of the entire answer-set program is presented at the beginning of the documentation. We note that programmers are not forced to declare blocks at all. If no block is specified in a file, all rules in that file belong to a dedicated default block. For each block, descriptions of the used predicates and types of involved terms, as well as pre-

Block Battleship (Check the source code of block Battleship)

encoding of the Battleship puzzle
[@Block Battleship](#)
 | [@block Guess](#)

Term Descriptions

X: X (Y) is a row (column) index ranging from 1 to 10
 In interval 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Y: X (Y) is a row (column) index ranging from 1 to 10
 In interval 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

X1: further row and column indices
 With same range as [X](#)

Y1: further row and column indices
 With same range as [X](#)

X2: further row and column indices
 With same range as [X](#)

Y2: further row and column indices
 With same range as [X](#)

Input Atoms

water(X, Y)
 there is no ship at position (X,Y)
[X](#): X (Y) is a row (column) index ranging from 1 to 10
[Y](#): X (Y) is a row (column) index ranging from 1 to 10

ship(X, Y)
 a ship is occupying position (X,Y)
[X](#): X (Y) is a row (column) index ranging from 1 to 10
[Y](#): X (Y) is a row (column) index ranging from 1 to 10

rowHint(X, H)
 in row X, H squares are occupied
[X](#): X (Y) is a row (column) index ranging from 1 to 10

colHint(Y, H)
 in column Y, H squares are occupied
[Y](#): X (Y) is a row (column) index ranging from 1 to 10

Output Atoms

ship(X1, Y1, X2, Y2)
 a ship is occupying the squares from (X1,Y1) to (X2,Y2).
[X1](#): further row and column indices
[Y1](#): further row and column indices
[X2](#): further row and column indices
[Y2](#): further row and column indices

Preconditions

Excl: no square shows both water and a part of a ship

Postconditions

Overlength: battleships are never longer than four squares

Subblocks

Block Guess (Check the source code of block Guess)

guess a configuration of battleships on the grid
[@Block Battleship](#)
 | [@block Guess](#)

Figure 4.2: HTML documentation of the Battleship program.

and postconditions are given. By default, hidden atoms, i.e., atoms mentioned neither in a block’s input nor in its output signature, are displayed as well in a dedicated section entitled “Hidden Atoms”. To hide them, option `-ha` can be used. The documentation also includes HTML versions of the programs’ source code, which can be particularly useful for sharing ASP code online. There are links from the documentation to the source code and vice versa. In particular, the document contains a link to the actual rules inside a block, unless this is suppressed using option `-s`. These rules are, by default, displayed together with the meta-comments of LANA. If option `-a` is used, such comments are not shown. Likewise, the rules for defining pre- and postconditions can be inspected by using the respective links. To enhance navigability between different parts of the document, predicates used in the source code or in signature declarations are, if available, linked to their respective descriptions. For instance, to find out the range of a variable in the output atoms section, say `X1`, the user can simply follow the link and thereby navigate to the description of `X1` (and `X`) in the term description section.

4.3 ASPUnit

ASPUnit is a tool to execute test cases that are formulated in LANA. Like ASPDoc, it is a command-line tool. An executable JAR file can be downloaded from the same web page as ASPDoc. For ASPUnit, each unit test has to be stored in a separate file. Although test cases are required to have a name, we allow that a user may omit an explicit name, in which case the file name is used by default. The tool takes as input a test-suite specification file, i.e., a file that contains the relevant information regarding locations of

```

Test Case ShipTopLeftCorner: Successful

Test Case NoDiagonalShips  : Successful

Test Case TouchingShips    : Failed
    two ships must not touch each other

Failed Test : @falseinall forbiddenShip
Counterexample:
Answer set:
  c(1).c(10).c(2).c(3).c(4).c(5).
  c(6).c(7).c(8).c(9).forbiddenShip.
  r(1).r(10).r(2).r(3).r(4).
  r(5).r(6).r(7).r(8).r(9).
  ship(1, 1, 1, 2).ship(1, 2, 1, 4).

```

Figure 4.3: A test report for the Battleship program.

the answer-set program, the files containing individual test cases, and the ASP solver that is needed to execute them. The syntax of a test-suite specification file is closely related to our annotation language itself. In particular, the specification of a test-suite has the following form:

```

@testsuite name
  description
@program      ASP files
@programdir   path to ASP files
@test         test case 1
@test         test case 2
  ...
@testdir      path to test cases
@solvertype   ASP solver
@solver       solver binary
@grunder      grunder binary

```

Hence, a test-suite specification starts with `@testsuite` followed by a name. Then, a short description may be given. A list of file names that together contain the answer-set program under test is expected after `@program`. These file names are relative to a path specified after `@programdir`. For each test case that we want to execute, we have to provide the file name that contains that test case specified by `@test`. The path to these files appears after `@testdir`. Then, information regarding the ASP solver has to be given. For this, `@solvertype` is used; the solver type is one of `DLV`, `clasp`, or `clingo`.

Table 4.4: Command-line options for ASPUnit.

Option	Description
-CE	show counterexample if a test case fails
-ce	do not show counterexample if a test case fails
-D	show description of a test case if it fails
-d	do not show description of a test case if it fails
-help, -h	print usage information

After `@solver`, an absolute file name of the ASP solver is expected. This file name may include additional parameters for that solver. If a separate grounder is needed, like for `clasp`, an absolute file name including command-line parameters have to be specified after `@grounder`.

Now, to run a number of test cases specified within a test-suite file, say `testsuite`, ASPUnit is invoked as follows:

```
java -jar aspunit.jar testsuite.
```

The tool will run all the unit tests on the answer-set program using the solver settings according to specifications in `testsuite`. A test report is printed to standard-output. This report contains information regarding success or failure for each test case. If a test case fails, a counterexample may be included, depending whether option `-CE` is set when ASPUnit is executed. Furthermore, if option `-D` is used, the test report will contain a short description of each test case that fails, obtained from the specification of the test cases themselves. For illustration, assume we run the test cases presented in Section 4.1.2 on the partial encoding of the battleship example. Recall that the first and second test case pass while the third one fails. The resulting test report, including a description of each test case and counterexamples for the failing test case, is given in Figure 4.3. A summary of ASPUnit command-line options is given in Table 4.4.

4.4 Related Work

As mentioned earlier, there are many notions of modularity for ASP in existence [24, 53, 74, 6, 96]. The advantage of the light-weight approach of LANA for grouping rules together by means of declaring blocks is that we do not change the semantics of programs and they can be directly parsed by ASP solvers. However, there are also disadvantages compared to other approaches to modularise logic programs. For example, related to our approach for annotating type information is RSIG [6], which is a language extension for specifying simple type information for programs and modules and thus requires its own parser. However, this type information simplifies the program rules as information about the type of variables is not required to be provided. On the other hand, DLP-functions [96] allow to compute the semantics of a logic program based on the semantics

of its separate DLP-functions which opens up new paths for potentially more efficient answer-set computation. To support the incremental development of logic programs, *lp*-functions can be used to structure a program and to develop it along with its specification by using specification constructors and their realisation theorems [74]. Blocks in LANA are not designed to serve one particular purpose, different interpretations are conceivable and are eventually determined by respective tool support like ASPUnit for unit testing blocks of rules.

To support large application developments, traditional languages offer programming tools that automatically generate searchable documentation, like e.g., JavaDoc. Methodologies like test-driven development provide a mechanism to incrementally unit test code and to support regression testing; JUnit is an example of this for Java. LANA provides the support for both, incorporating the annotation of tests directly into the documentation of the program. The use of assertions in LANA is inspired by the Java Modelling Language [106] and annotations as used in Prolog [105].

Similar to our unit testing approach, Prolog offers unit-testing functionality called PLUnit [151]. As in our approach, where tests are expressed using ASP itself and only a Java wrapper is used to call all tests within a given test-suite, tests in PLUnit are formulated as Prolog clauses.

While many interesting features of LANA for development support can be realised by stand-alone tools like ASPDoc and ASPUnit, things become more interesting when the respective functionalities are available within an IDE for ASP. In particular, Febbraro, Leone, Reale, and Ricca [64] provide a mechanism for unit testing in ASP which is incorporated in their IDE *Aspide*. They base unit tests on clusters on the dependency graphs or rule labelling, while we allow the user to decide which rules belong to a test by defining blocks. Recently, Amendola, Berei, and Ricca [4] introduced a unit-testing framework that unifies the strengths of LANA and the earlier approach for unit testing in *Aspide*: Like LANA, it is annotation based and test cases can be defined within ASP code, but it keeps the style of expressing test case conditions from *Aspide*. Other IDEs which have been developed for ASP include *SeaLion* [135], *APE* [162], and *iGROM*¹.

In traditional software engineering, coding standards including documentation are imposed, especially in case of developing large software projects. In this chapter, we proposed LANA as part of coding standards for ASP. Future work will look into other best-practices for writing and maintaining ASP programs. The growing number of applications of ASP can provide a wealth of information for this.

¹<https://sourceforge.net/projects/igrom/>.

ASP for Event-Sequence Testing

The previous chapters covered various aspects of testing and systematic programming support for ASP programs. Many of the methods that have been introduced rely on ASP itself. In this chapter, we continue to use ASP for testing, but the object of interest becomes a different one: instead of ASP, we focus on an interesting combinatorial problem outside of ASP that is relevant for testing event-driven software, viz. on so-called *sequence-covering arrays* (SCAs), due to Kuhn, Higdon, Lawrence, Kacker, and Lei [103], and we show how to exploit ASP to compute SCAs.

This chapter is organised as follows: In Section 5.1, we review SCAs. In Section 5.2, we analyse the intrinsic problem complexity of SCA computation which indeed shows that ASP is a suitable computational means. Then, in Section 5.3, we show how SCAs can be generated using ASP. We present improved, sometimes optimal, upper bounds regarding the size of many SCAs. We furthermore present a greedy algorithm, based on ASP, for computing larger SCAs. In Section 5.4, we turn towards a real-world example as described by Kuhn et al. [103]. We discuss how the basic ASP encoding from Section 5.3 can be refined, step-by-step, to take different constraints and problem variations into account. The resulting array is significantly smaller than the one of Kuhn et al. that was created by modifying a precomputed SCA. In fact, we show that our solution is optimal with respect to the specified coverage criteria.

5.1 Sequence-Covering Arrays

5.1.1 Overview

In many applications, faults only show up if events occur in a certain order. An example are atomicity violations in multi-threaded applications where a pair of shared memory accesses of one thread is interleaved with an unfortunate access of another thread. Testing such applications thus requires exercising *event sequences*. Since the number of event

sequences is factorial in the number of events, exhaustive testing is infeasible in general. If we assume that bugs are triggered by the interaction of only a low number of events, testing costs can be reduced drastically without sacrificing much fault-detection potential by using suitable combinatorial designs [84, 119]. To this end, Kuhn, Higdon, Lawrence, Kacker, and Lei [103] introduced *sequence-covering arrays* (SCAs) for combinatorial event sequence testing. An SCA of strength t is an array of permutations of events such that every ordering of any t events appears as a subsequence of at least one row. For illustration, the following matrix is an SCA for four events $\{1,2,3,4\}$ with $t = 3$:

$$\begin{pmatrix} 3 & 1 & 2 & 4 \\ 1 & 3 & 4 & 2 \\ 2 & 3 & 4 & 1 \\ 4 & 1 & 2 & 3 \\ 2 & 1 & 4 & 3 \\ 4 & 3 & 2 & 1 \end{pmatrix}.$$

Any ordering of three events can be found as subsequence of at least one row. If three particular events occur as subsequence of a row, we say that a row covers the three events. For example $(1, 2, 3)$ is a subsequence of the fourth row, $(1, 3, 2)$ is a subsequence of the second row, $(2, 3, 4)$ is covered by the third row, and so on.

SCAs are relevant for testing applications where the order of events is decisive. Examples of respective event sequences in such applications are user actions for user-interface testing [170], visited web pages in dynamic web applications [166], and shared variable accesses in multi-threaded programs [87, 113] as we already mentioned. If an SCA of strength t is used as basis for a test plan for such applications, i.e., each row of the SCA is turned into the specification of a test run that imposes a particular order on relevant event, not all permutations of events will be tested in general, but at least we have the guarantee that the potential interaction of any t events will be tested at least once.

In practice, a direct application of SCAs for testing is often impaired by additional constraints on the order of events. It can be necessary to exclude, for example, that a “paste” event happens before a “copy” event when testing a user interface. Also, the conditions that identify the sequences that should be covered can vary and often involve quite complex definitions. For example, to test thread interleavings, one could require to test all sequences such that one variable is written by one thread and subsequently read by another thread but there is no write operation between them [87, 113]. Hence, quite expressive constraints and variations from standard SCAs have to be taken into account. Furthermore, sometimes certain orderings are regarded as redundant and should be avoided to reduce testing costs. For example, the order in which devices are connected to a computer is not relevant if the computer is not booted.

One approach to address such considerations is to accordingly modify precomputed SCAs as exemplified by Kuhn et al. [103]. This means that any test sequence which, e.g., violates some ordering constraints has to be removed from the SCA. To maintain coverage, removed sequences have to be replaced by permutations thereof that comply to

the problem-specific requirements. This is not always possible in a straightforward way and can result in a considerable and in principle avoidable overhead regarding the size of arrays. On the other hand, developing and maintaining dedicated algorithms to compute variations of SCAs usually comes with high costs and is not preferable if requirements change over time—which is a daily aspect in real-world system development—or one wants to experiment with different designs.

We propose to use ASP for computing SCAs and variations thereof. As an expressive high-level specification language, it allows to state complex coverage criteria, involving constraints and complex, possibly recursive, definitions in a concise and *elaboration-tolerant* way, i.e., small variations in a problem specification require only small modifications of the program representation [117]. On the other hand, SCAs can be efficiently computed through highly optimised ASP solvers [25]. Since it requires only little effort to state quite complex coverage conditions in ASP, a tester is able to rapidly specify and to experiment with different versions of SCAs.

5.1.2 Definition of Sequence-Covering Arrays

Let us now review the formal definition of SCAs as introduced by Kuhn et al. [103]. SCAs are combinatorial designs related to covering arrays. While covering arrays require that each t -way combination of parameters occurs at least once in a test case for some fixed t , SCAs take the order of events into account and require that each t -sequence of events is tested in at least one test sequence in that order, where a t -sequence over a set S of symbols is a sequence of t pairwise distinct elements of S . Following Kuhn et al. [103], SCAs are defined as follows:

Definition 25. A sequence-covering array (SCA) with parameters n , S , and t , or an (n, S, t) -SCA for short, is an $n \times |S|$ matrix M of symbols from a finite set S of symbols such that

- (i) each row of M is a permutation of S , and
- (ii) for each t -sequence $\sigma = (s_1, s_2, \dots, s_t)$ over S , there is one row $\rho = (a_{i1}, \dots, a_{i|S|})$ in M such that σ is a subsequence of ρ .

We say that an (n, S, t) -SCA is of strength t and of size n .

Definition 26. The sequence-covering array number for S and t , $\text{SCAN}(S, t)$, is the smallest n such that an (n, S, t) -SCA exists.

Definition 27. An (n, S, t) -SCA is optimal if $\text{SCAN}(S, t) = n$.

As usual, l is a lower bound for $\text{SCAN}(S, t)$ if $l \leq \text{SCAN}(S, t)$, and u is an upper bound for $\text{SCAN}(S, t)$ if $\text{SCAN}(S, t) \leq u$. We will also denote an $(n, \{1, \dots, s\}, t)$ -SCA as an (n, s, t) -SCA with $\text{SCAN}(s, t)$ for brevity.

For illustration, the following matrix M constitutes an optimal $(7, 5, 3)$ -SCA:

$$M = \begin{pmatrix} 5 & 2 & 3 & 1 & 4 \\ 3 & 2 & 5 & 4 & 1 \\ 1 & 5 & 4 & 3 & 2 \\ 3 & 4 & 5 & 1 & 2 \\ 4 & 2 & 5 & 1 & 3 \\ 2 & 4 & 3 & 1 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}.$$

Each of the seven rows is a permutation of the set $S = \{1, \dots, 5\}$ and each 3-sequence over S is covered by at least one row. For instance, the 3-sequence $(5, 3, 4)$ is covered by the first row of M , and $(3, 4, 5)$ is covered by the fourth row of M (as well, $(3, 4, 5)$ is covered by the last row of M). Note that there are $5 \cdot 4 \cdot 3 = 40$ such 3-sequences.

A collection of precomputed SCAs of strength 3 and 4 is available online.¹ These SCAs were computed using a simple greedy algorithm introduced by Kuhn et al. [103]. To compute a t -strength SCA for a set S of events, this algorithm iteratively computes single rows of the SCA: In each iteration, it computes a fixed number of permutations of S . Then, it selects the permutation π that obtains maximal coverage of previously uncovered t -sequences as the next row of the SCA. After that, π in reverse order, π' , is added. Adding π' is justified because π' always covers the same number of previously uncovered t -sequences as π [103]. This procedure is iterated until all t -sequences are covered. If event x must not occur before event y in any test case, it is possible to specify the pair (x, y) as additional input of the algorithm. Subsequently, only rows that do not contain (x, y) as subsequence are added. Also, if a constraint is specified, the heuristic to add rows in reverse order is disabled.

Though the greedy algorithm can take simple constraints into account, one downside is that more complex constraints or other variations from plain SCAs arising from the requirements of different test scenarios are hard to incorporate. To overcome this shortcoming, we use ASP in what follows as a declarative tool to compute SCAs and demonstrate that quite complex constraints can be incorporated into a solution in a concise and elaboration-tolerant way, and with ease.

5.2 Prelude: Complexity of SCA Generation

Deciding whether a normal logic program has an answer-set is NP-complete, thus computing answer-sets can be quite expensive. Indeed, the runtime of ASP solvers is exponential with respect to the size of the ground program in the worst case. In this section, we analyse the computational worst-case complexity of generating SCAs.

¹csrc.nist.gov/projects/automated-combinatorial-testing-for-software/combinatorial-methods-in-testing/event-sequence-testing.

For our complexity analysis, we actually study a slight generalisation of the problem of generating SCAs. On the one hand, usually not all permutations of events are allowed for testing, some could be excluded for various reasons. On the other hand, usually not all t -sequences need to be covered, some may be forbidden or regarded as redundant. We next formalise this natural generalisation as a decision problem and study its complexity.

Definition 28. *An instance of the generalised event sequence testing problem GEST is given by a tuple (S, P, T, k) , where P is a set of permutations of a set S of symbols, T is a set of t -sequences over S with $t \geq 2$, and k is a positive integer. A tuple (S, P, T, k) is a yes-instance of GEST iff there exists a matrix M with at most k rows such that*

- (i) *each row of M is an element from P , and*
- (ii) *for each t -sequence $\sigma = (s_1, s_2, \dots, s_t)$ in T , there is a row $\rho = (a_{i_1}, \dots, a_{i_t})$ in M such that σ is a subsequence of ρ .*

Theorem 12. *GEST is NP-complete.*

Proof. We first show that GEST is in NP. Any instance (S, P, T, k) of GEST can be decided by non-deterministically “guessing” a $k \times |S|$ matrix M of symbols from S and checking conditions (i) and (ii) from Definition 28 in polynomial time.

To show NP-hardness, we reduce the NP-hard problem of checking set coverage to GEST. Formally, an instance of *set cover* (SC) is a tuple (V, F, k) , where V is a set of elements, F is a collection of subsets of V , and k is a positive integer. A tuple (V, F, k) is a yes-instance of SC iff there is a subcollection $F' \subseteq F$ of size at most k whose union contains each element of V . It is well known that SC is NP-complete [99].

Let (V, F, k) be an instance of SC. Assume that “ \square ” is a separation symbol not contained in V . Define

$$S = V \cup \{\square\}.$$

For each $f \in F$, construct a permutation π_f of S by arbitrarily arranging the symbols from f before \square and the symbols in $V \setminus f$ after \square . Define

$$P = \{\pi_f \mid f \in F\} \quad \text{and} \quad T = \{(v, \square) \mid v \in V\}.$$

Note that $|P| = |F|$. We show that (V, F, k) is a yes-instance of SC iff (S, P, T, k) is a yes-instance of GEST.

Assume that (V, F, k) is a yes-instance of SC. Hence, there exists a set $F' \subseteq F$ of size at most k whose union contains each element of V . Construct a matrix M such that $\pi_f \in P$ is a row of M iff $f \in F'$. Clearly, M is a matrix from symbols from S that satisfies Condition (i) of Definition 28. We show that M satisfies Condition (ii) as well. Towards a contradiction, assume that there is a 2-sequence (v, \square) in T and there is no row in M such that v occurs before \square . Since (V, F, k) is a yes-instance of SC, F' contains

at least one set f with $v \in f$. Since π_f is a row of M , and v occurs before \square in π_f by construction, we arrive at a contradiction. Hence, (S, P, T, k) is a yes-instance of GEST.

For the other direction, assume that (S, P, T, k) is a yes-instance of GEST. Hence, there exists a matrix M that satisfies conditions (i) and (ii) from Definition 28. We show that then (V, F, k) is a yes-instance of SC. Define set F' as a subset of F that contains an element $f \in F$ iff

(*) there is a row π of M such that all elements of f occur before \square in π .

Clearly, F' is of size at most k since M consists of at most k rows, and, for any row of M , precisely one $f \in F$ satisfies (*). It remains to show that for each $v \in V$, v is contained in some set in F' . Towards a contradiction, assume that for some $v \in V$ there is no set in F' that contains v . Since (S, P, T, k) is a yes-instance of GEST, and $(v, \square) \in T$, it follows that in one row π_f of M , v occurs before \square . By construction of F' , F' contains a set $f \in F$ consisting of all symbols of π_f that occur before \square . Hence, $v \in f$ which contradicts the assumption that no such set in F' exists. It follows that (V, F, k) is a yes-instance of SC. \square

Hence, any approach that is capable of deciding problems in GEST cannot avoid worst-case exponential runtime behaviour unless $P = NP$. Note that the SCA generation problems studied in this paper are instances of GEST. Moreover, for any problem in NP, there exists a uniform ASP encoding [156, 38]. Hence, NP-completeness of GEST further justifies ASP as a tool to formalise and compute such problems.

Although GEST is NP-complete, one could further ask whether GEST is *fixed-parameter tractable* for a suitable problem parameter. A natural choice for such a parameter for a problem instance (S, P, T, k) would be the size k of the SCA because it can be assumed to be small in practice. We denote the parameterised version of GEST with k as parameter as the *standard parameterisation of GEST*. The following result implies that GEST is not in FPT unless the W-hierarchy collapses up to the second level.

Theorem 13. *The standard parameterisation of GEST is $W[2]$ -complete.*

Proof. Consider an instance (V, F, k) of SC. If we take k as parameter, i.e., the size of the subcollection $F' \subseteq F$ whose union contains each element of V , SC is $W[2]$ -complete [47].

Membership. To show membership in $W[2]$, we use an fpt-reduction from GEST to SC. Let (S, P, T, k) be an instance of GEST. For any $\pi \in P$, construct a set $f_\pi \subseteq T$ that contains any t -sequence $\tau \in T$ iff the elements of τ occur in π in the same order as in τ . Define

$$F = \{f_\pi \mid \pi \in P\}.$$

We show that (S, P, T, k) is a yes-instance of GEST iff (T, F, k) is a yes-instance of SC.

Assume that (S, P, T, k) is a yes-instance of GEST. Hence, there exists a matrix M satisfying Conditions (i) and (ii) from Definition 28. Define F' as the subset of F that

contains f_π iff π is a row of M . Clearly, the size of F' is at most k . It remains to show that the union of the elements of F' contain each $\tau \in T$. Towards a contradiction, assume that there is an element $\tau \in T$ such that no set in F' contains τ . Since (S, P, T, k) is a yes-instance of GEST, M contains a row $\pi \in P$ such that the symbols in τ occur in π in the same order as in τ . Thus $\tau \in f_\pi$. Since $f_\pi \in F'$, we arrive at a contradiction. Therefore, (T, F, k) is a yes-instance of SC.

Assume now that (T, F, k) is a yes-instance of SC. Hence, there is a subset $F' \subseteq F$ of size at most k whose union contains each element of T . Construct a matrix M according to Definition 28 such that M contains, for each $f \in F'$, one row $\pi \in P$ that satisfies $f_\pi = f$. Clearly, M consists of at most k rows from P . It remains to show that M satisfies Condition (ii) of Definition 28. Towards a contradiction, assume that there is a t -sequence $\tau \in T$ such that no row contains the symbols in τ in the same order as in τ . Since (T, F, k) is a yes-instance of SC, there is a set $f \in F'$ that contains τ . Since M contains a row π with $f_\pi = f$, it follows from the construction of f_π that π contains the symbols in τ in the same order as in τ . We thus arrive at a contradiction, and so (S, P, T, k) is yes-instance of GEST.

Hardness. To show that GEST is W[2]-hard, we define an fpt-reduction from SC to GEST. In fact, the reduction used in the hardness proof of Theorem 12 is such an fpt-reduction since the problem parameter k is preserved, and W[2]-hardness of GEST follows. \square

Hence, even if we are interested only in relatively small test plans, it is presumably not possible to avoid worst-case exponential runtime.

5.3 SCA Computation

We now discuss how ASP can be used to generate SCAs. Our goal is not only to present approaches to compute generic SCAs, i.e., SCAs created without additional constraints or requirements, rather we want to demonstrate that ASP can be used as an efficient and effective declarative tool to compute SCAs tailored to specific test scenarios. We in particular demonstrate that (i) the declarative nature of ASP encodings can help to state complex coverage criteria, involving constraints and possibly recursive definitions with ease in a concise and elaboration-tolerant way, and (ii) when the declarative nature of ASP encodings is coupled with ever-improving efficiency of ASP solvers, even simple encodings that closely reflect the problem statement in natural language can provide better SCAs (e.g., smaller SCAs) compared to those obtained from a dedicated algorithm.

Prior to our subsequent discussion in Section 5.4 addressing how different problem elaborations can be incorporated into a single answer-set program, we introduce in what follows an answer-set program for computing generic SCAs. This program will serve as basis for further problem elaboration discussed in the sequel. We also introduce a new greedy approach that combines a simple variation of the basic ASP encoding with an iterative greedy procedure.

5.3.1 Basic Encoding

To begin with, we present an ASP program for computing (n, s, t) -SCAs with $t = 3$. We assume throughout that $s \geq 3$. Note that this program can be changed in a straightforward way to obtain encodings for any fixed $t > 3$. We introduce our encoding step-by-step.

Encoding

We start by expressing that the symbols of the array are integers between 1 and s , and row indices of the SCA correspond to integers 1 to n . Note that s and n function as parameters of the program:

```
sym(1..s) .
row(1..n) .
```

For the representation of the SCA, we use the predicate $hb(N, X, Y)$, expressing that in row N event X happens before event Y . The basic idea is that we will define this happens-before relation in a way that it is, for each row, a strict total order on the event symbols.

The first rule states that for any two distinct symbols X and Y in each row, either X happens before Y or Y happens before X :

```
1 {hb(N, X, Y), hb(N, Y, X)} 1 :- row(N), sym(X; Y), X != Y.
```

We need further rules to guarantee that the happens-before relation is indeed a strict total order. In particular, we need rules that guarantee that the happens-before relation is transitive and irreflexive. Now, transitivity can be expressed in a straightforward way:

```
hb(N, X, Z) :- hb(N, X, Y), hb(N, Y, Z) .
```

Directly expressing inductive definitions as above is a particular strength of ASP and distinguishes it from related declarative approaches that are more based on the semantics of classical first-order logic.

To state that the happens-before relation is irreflexive, a simple additional constraint is required.

```
:- hb(N, X, X) .
```

Hence, it is forbidden that a symbol occurs before itself.

```

% ASP encoding for (n, s, 3)-SCAs
sym(1..s). row(1..n).

% guess happens-before relation
1 { hb(N, X, Y), hb(N, Y, X) } 1 :- row(N), sym(X; Y), X != Y.

% happens-before is transitive
hb(N, X, Z) :- hb(N, X, Y), hb(N, Y, Z).

% happens-before is irreflexive
:- hb(N, X, X).

% check if each 3-sequence is covered
covered(X, Y, Z) :- hb(N, X, Y), hb(N, Y, Z).
:- not covered(X, Y, Z), sym(X; Y; Z), X != Y, Y != Z, X != Z.

```

Figure 5.1: ASP encoding $\Pi^3(n, s)$.

This finally implies that the happens-before relation is a strict total order on the event symbols $\{1, \dots, s\}$, which further implies that each row is a permutation of the event symbols when we order them according to the happens-before relation.

It only remains to make sure that each 3-sequence of symbols is covered by some row. Observe that a 3-sequence is a triple of pairwise distinct symbols. A 3-sequence (X, Y, Z) is covered if X happens before Y and Y happens before Z in some row N . We finally define covered 3-sequences and forbid that a 3-sequence is not covered:

```

covered(X, Y, Z) :- hb(N, X, Y), hb(N, Y, Z).
:- not covered(X, Y, Z), sym(X; Y; Z), X != Y, Y != Z, X != Z.

```

The entire ASP program, $\Pi^3(n, s)$, with parameters n and s for generating $(n, s, 3)$ -SCAs is given in Figure 5.1.

Intuitively, each answer set of program $\Pi^3(n, s)$ represents an $(n, s, 3)$ -SCA. In fact, the answer sets of $\Pi^3(n, s)$ and the $(n, s, 3)$ -SCAs are in a one-to-one correspondence. This relation can be formalised as follows:

Definition 29. *An answer set X of $\Pi^3(n, s)$, for $s \geq 3$, represents an $n \times s$ matrix M if, for any i, j , $1 \leq i < j \leq s$, and any r , $1 \leq r \leq n$, it holds that both $M_{r,i} = s_1$ and $M_{r,j} = s_2$ precisely in case X contains the atom $\text{hb}(r, s_1, s_2)$.*

Proposition 2. *Each answer set of $\Pi^3(n, s)$ represents a single $(n, s, 3)$ -SCA, and each $(n, s, 3)$ -SCA is represented by a single answer set of $\Pi^3(n, s)$.*

For illustration, to compute a $(7, 5, 3)$ -SCA, `gringo` and `clasp` can be invoked as follows:

```
gringo sca-3.gr -c n=7,s=5 | clasp.
```

File `sca-3.gr` contains program $\Pi^3(n, s)$. The `gringo` option `-c n=7,s=5` instantiates the program parameters n and s to 7 and 5, respectively. Any resulting answer set corresponds to a $(7, 5, 3)$ -SCA. For instance, in some answer set, the first row of the SCA M given in Section 5.1 is encoded by the atoms

```
hb(1, 1, 4), hb(1, 3, 1), hb(1, 2, 3), hb(1, 5, 2), hb(1, 5, 3),  
hb(1, 2, 1), hb(1, 3, 4), hb(1, 2, 4), hb(1, 5, 4), hb(1, 5, 1).
```

To compute more than one $(7, 5, 3)$ -SCA, an upper bound on the number of answer sets that `clasp` should compute can be specified as an integer option (0 means that all answer sets are computed).

Program $\Pi^3(n, s)$ nicely illustrates how challenging search problems can be concisely encoded using ASP: The program consists of only seven rules that closely reflect the problem statement in natural language. We note that only little training time is needed to enable a tester to use ASP for test authoring. This is mainly because of the genuine declarative nature of ASP, which does not require specialised knowledge on data structures or algorithms.

Also, by using our ASP encoding $\Pi^3(n, s)$ and the ASP solver `clasp`, we could improve known upper bounds for many SCAs significantly. A comparison of the SCAs generated using ASP and the greedy algorithm of Kuhn et al. [103] is given in Table 5.1. Computation times for the reported upper bounds range from fractions of a second to 180 minutes. We have considered strength 3 SCAs for five to 80 events. The known upper bounds reported by Kuhn et al. [103] could be improved throughout. The more events are considered, the more drastic are the improvements: for some arrays, we almost only need half the number of test sequences. Such savings are especially significant in settings where running single test sequences are costly.

For small SCAs—viz. for 5 to 10 events—the new upper bounds are actually optimal bounds. Optimality of upper bounds was established using ASP itself. To show that an (n, s, t) -SCA is optimal, we try to compute an $(n - 1, s, t)$ -SCA. If this fails, i.e., if the ASP solver terminates without returning an answer set, the (n, s, t) -SCA is indeed optimal. Since $\text{SCAN}(10, 3) = 9$, 9 is a trivial lower bound for any $\text{SCAN}(s, 3)$ with $s > 10$. Note that greedy algorithms—or any approaches based on incomplete search—are unable to prove optimal bounds or to establish lower bounds at all.

A limitation of using the ASP encoding $\Pi^3(n, s)$ concerns scalability. The greedy approach of Kuhn et al. [103] seems to scale quite well; the authors report on SCAs for up to 80 events not only for strength 3 arrays, but they also consider arrays of strength 4 where our ASP approach quickly reaches its limits.

Table 5.1: Upper bounds n for $\text{SCAN}(s, 3)$ obtained by Kuhn et al. [103] and our ASP encoding. A star indicates an optimal bound.

s	n (Kuhn et al. [103])	n (ASP)
5	8	7*
6	10	8*
7	12	8*
8	12	8*
9	14	9*
10	14	9*
11	14	10
12	16	10
13	16	10
14	16	10
15	18	10
16	18	10
17	20	11
18	20	12
19	22	12
20	22	12
21	22	12
22	22	12
23	24	13
24	24	13
25	24	14
26	24	14
27	26	14
28	26	14
29	26	14
30	26	15
40	32	17
50	34	18
60	38	20
70	40	22
80	42	23

5.3.2 Greedy Algorithm

In the remainder of this section, we introduce and discuss an ASP-based greedy algorithm, inspired by that of Kuhn et al. [103], for computing larger SCAs. The motivation to study such an algorithm is to combine the modelling capabilities of ASP, especially in the light of constraints and problem elaborations (as detailed in the next section), with the scalability of a greedy approach.

Require: s is the number of symbols.
Ensure: N represents an $(n, s, 3)$ -SCA.

- 1: $N \Leftarrow \emptyset$
- 2: $n \Leftarrow 0$
- 3: **repeat**
- 4: $n \Leftarrow n + 1$
- 5: $X \Leftarrow$ answer set of $\Pi_{grdy}^3(s, n) \cup N$
- 6: $N \Leftarrow N \cup X|_{hb/3}$
- 7: **until** N represents an $(n, s, 3)$ -SCA

Figure 5.2: Greedy algorithm for computing an $(n, s, 3)$ -SCA.

In this context, we also mention that the greedy algorithm of Kuhn et al. has a certain weakness, which is related to the heuristic that for any newly computed sequence the reverse sequence is added as well (cf. Section 5.1). As we will show next, this makes the algorithm inherently unable to compute optimal SCAs in general. Actually, the inability to find optimal SCAs follows immediately from the observation that some optimal SCAs, e.g., $(7, 5, 3)$ -SCAs, are of odd size. However, ASP can be used to show that even optimal SCAs of even size cannot be found by that greedy approach in general. The idea is to augment program $\Pi^3(n, s)$ by a rule that states that every second row is the inversion of the previous one. This is simply expressed by the following rule:

$$hb(N, X, Y) :- row(N), hb(N-1, Y, X), N \#mod\ 2 == 0.$$

Here, predicate $\#mod$ is the usual modulo operation. Hence, the intuitive reading of this rule is that for any row with even index N , the happens-before relation is the inverse of the happens-before relation of the preceding row $N-1$. We know already from Table 5.1 that any $(8, 6, 3)$ -SCA is optimal. However, $\Pi^3(8, 6)$ augmented by the above rule yields no answer set, which shows that $(8, 6, 3)$ -SCAs cannot be computed by the greedy algorithm of Kuhn et al. [103]. Next, we present an ASP-based greedy algorithm inspired by that of Kuhn et al. that does not rely on adding inverted rows.

Encoding

Figure 5.2 represents our ASP-based greedy algorithm for computing SCAs. The main idea is to compute one row of a SCA at a time instead of computing the entire array. In each iteration, one further row is computed using ASP where the number of covered 3-sequences is maximised. For this purpose, we use program $\Pi_{grdy}^3(s, i)$, which is depicted in Figure 5.3. Program $\Pi_{grdy}^3(s, i)$ takes the number s of events and a row index i as parameters. Both the ASP encoding and the greedy algorithm are introduced only for SCAs of strength 3. However, versions for computing SCAs of strength greater than 3

are obtained in a straightforward way. For example to obtain a program for strength 4 SCA, only the last two rules of $\Pi_{grdy}^3(s, i)$ have to be replaced by the following two rules:

```
covered(W, X, Y, Z) :- hb(n, W, X), hb(n, X, Y), hb(n, Y, Z).
#maximize[covered(_, _, _, _)].
```

Program $\Pi_{grdy}^3(s, i)$ is quite similar to $\Pi^3(n, s)$. However, each answer set of $\Pi_{grdy}^3(s, i)$ corresponds only to a single row with index i of an SCA. The idea is to represent preceding rows with index 1 to $i - 1$ by means of facts $hb/3$. These facts are joined with $\Pi_{grdy}^3(s, i)$. Then, the answer sets of $\Pi_{grdy}^3(s, i)$ correspond to those rows that obtain maximal coverage of previously uncovered 3-sequences. The encoding follows the *guess, check, and optimise pattern*, i.e., we use guessing rules to span the search space, constraints to filter unwanted solution candidates, and rules that express a preference relation on answer sets. In particular, rule

```
#maximize[covered(_, _, _)].
```

states that we seek for answer sets with a maximal number of covered 3-sequences.

The algorithm itself is rather simple (cf. Figure 5.2): It takes parameter s as input and computes an $(n, s, 3)$ -SCA. Initially, the set N that represents a (partial) SCA by means of facts $hb/3$ equals the empty set. In each iteration, $\Pi_{grdy}^3(s, i) \cup N$ is used to compute the next row of the SCA that obtains maximal increase of previously uncovered 3-sequences. The respective $hb/3$ facts for that row are then added to N . This procedure iterates until no uncovered 3-sequences are left (the ASP solver itself will indicate that no further optimisation is possible). Since the computation of optimal answer sets can become very time consuming, we additionally impose an upper bound on the time that is spent for optimising answer sets, thus improvements in each step will not be maximal in general. However, this seems to be a reasonable compromise regarding runtime and the size of computed SCAs. We used time limits of up to 10 minutes for computing single rows, depending on the problem size.

To sum up our results so far, our analysis of the heuristic proposed by Kuhn et al. [103] using ASP has pinpointed some shortcomings of the former and has helped us to learn more about the problem at hand. Furthermore, we have proposed a new greedy algorithm making use of a slight variation of $\Pi^3(n, s)$. The ASP solver takes care entirely of the greedy optimisation of the single rows of the SCA. The algorithm thus only keeps track of the partial SCA and incrementally calls the ASP solver to compute new rows.

Table 5.2 summarises a comparison of our greedy ASP algorithm with the greedy algorithm of Kuhn et al. [103] for strength 3 and 4 SCAs involving 10 to 80 events. For strength 3 SCAs, our algorithm is competitive with that of Kuhn et al. and upper bounds could be improved throughout by some rows. For strength 4 SCAs, the greedy ASP approach is feasible for up to 40 symbols where upper bounds could be improved even more

```
% guess single row with index i of an (n,s,3)-SCA
sym(1..s).

% guess happens-before relation
1 { hb(i,X,Y), hb(i,Y,X) } 1 :- sym(X;Y), X != Y.

% happens-before is transitive
hb(i,X,Z) :- hb(i,X,Y), hb(i,Y,Z).

% happens-before is irreflexive
:- hb(i,X,X).

% maximise coverage
covered(X,Y,Z) :- hb(N,X,Y), hb(N,Y,Z).
#maximize[covered(_,_,_)].
```

Figure 5.3: ASP encoding $\Pi_{grdy}^3(s, i)$.

drastically than for strength 3 SCAs. However, we were not able to compute SCAs for 40 to 80 symbols, which shows a limitation of our ASP-based approach that is probably acceptable unless the need for larger instances with a high level of interaction is indeed motivated by some application scenario. Here, it is to mention that scalability is certainly a characteristic strength of the simple greedy algorithm of Kuhn et al., since dedicated data structures, e.g., efficient bit-vectors, can be used for representing covered sequences. However, by using ASP we get bounds for strength 3 SCAs for up to 80 symbols and can also improve bounds for strength 4 SCAs for up to 40 symbols. Again, we emphasise that our goal is not to compute generic SCAs but to allow a tester to express different requirements with little effort, by adding or changing some rules of the ASP program, which can readily be done using the greedy ASP approach. We pursue this issue in the next section.

5.4 Problem Elaborations

Next, we turn to the actual strengths of using ASP as an elaboration tolerant representation formalism for event sequence testing. We describe how ASP can be used for generating SCAs in a scenario that involves additional constraints and other problem variations that make it impossible to directly use precomputed SCAs. In particular, we use a *real-world testing problem* described by Kuhn et al. [103] for making our point. The specification of this testing problem is as follows: There are five different devices that have to be connected to a laptop. These devices can be connected before or after a boot-up phase. Further actions that have to be performed on the laptop are opening

Table 5.2: Comparison of our greedy ASP approach and that of Kuhn et al. [103]: upper bounds n for SCAN($s, 3$) and SCAN($s, 4$).

s	$t = 3$		$t = 4$	
	Kuhn et al. [103]	ASP	Kuhn et al. [103]	ASP
10	14	11	66	55
20	22	17	120	104
30	26	22	156	149
40	32	26	182	181
50	34	29	204	-
60	38	32	222	-
70	40	35	238	-
80	42	36	250	-

an application and initiating a scanning process. The peripherals can be connected to the laptop in any order; however, the order of events influences the functionality of the system. Thus, SCAs lend themselves as a basis for a suitable testing plan.

There are eight events relevant for testing: connecting devices (p_1, \dots, p_5), booting the system ($boot$), starting an application ($appl$), and running a scan ($scan$). Testing in this scenario is rather time consuming since it requires setting up the system manually. Therefore, obtaining an optimal test plan is desirable. Following Kuhn et al., only SCAs of strength 3 are considered to keep the size of the test plan reasonable.

5.4.1 Forbidden Sequences

For eight events, optimal SCAs of strength 3 comprise eight rows. However, we cannot use precomputed $(8, 8, 3)$ -SCAs since certain constraints regarding the order of events have to be taken into account. While most events can happen in any order, starting the application cannot happen before the system is booted, and running a scan requires that the application is already running.

Instead of covering all 3-sequences, we want to generate SCAs such that (i) in each row, $boot$ happens before $appl$ and $appl$ happens before $scan$, and (ii) all 3-sequences such that $boot$ happens before $appl$ and $appl$ happens before $scan$ are covered by at least one row. We only have to slightly modify program $\Pi^3(n, s)$ to account for (i) and (ii). First, instead of integers to denote events, we would like to use more descriptive constant symbols. Thus, we replace $\text{sym}(1..s)$ in $\Pi^3(n, s)$ by

$$\text{sym}(boot;p_1;p_2;p_3;p_4;p_5;appl;scan) .$$

Concerning (i), we define which orderings are excluded and add a respective constraint that forbids that event a happens before b if “ a before b ” is excluded.

```

excluded(scan,appl) .
excluded(appl,boot) .
excluded(X,Z) :- excluded(X,Y) , excluded(Y,Z) .
:- hb(_,X,Y) , excluded(X,Y) .

```

Regarding (ii), we simply define those 3-sequences that are not consistent with the excluded orderings as already covered:

```

covered(X,Y,Z) :- excluded(X,Y) , sym(X;Y;Z) .
covered(X,Y,Z) :- excluded(X,Z) , sym(X;Y;Z) .
covered(X,Y,Z) :- excluded(Y,Z) , sym(X;Y;Z) .

```

We denote the resulting program as $\Pi_1^3(n)$.

Recall that $(8, 8, 3)$ -SCAs are optimal for eight symbols. Since, $\Pi_1^3(8)$ does not yield any answer set, it follows that the stipulation on admissible orderings requires additional rows. In this case, this is because the number of 3-sequences that can be covered by a single row is reduced if certain events are required to happen in a strict order. Indeed, a solution for $\Pi_1^3(9)$ can be computed, hence 9 is an optimal bound for an SCA satisfying that each row is consistent with the specified ordering constraints. The solver `clasp` needs fractions of a second to find an SCA of size 9 and about 1 minute for checking optimality.

5.4.2 Redundant Sequences

Besides forbidden orderings, we also have to deal with redundant sequences: If devices are connected to the laptop before the boot-up phase, the order is not relevant. In fact, we only require strength 3 coverage for events p_1, \dots, p_5 , `appl`, and `scan`. Concerning the interaction of events p_1, \dots, p_5 , and `boot`, we regard strength 2 coverage as sufficient, i.e., we are only interested in whether the connection of the peripherals happens before or after the boot-up phase. Hence, we need a variable strength SCA, in which we seek to have strength 2 coverage for one set of events and strength 3 coverage for another one.

First, we add two sets of facts to declare the sets of events for which we want to obtain strength 2 and strength 3 coverage, respectively:

```

threeWay(p1;p2;p3;p4;p5;appl;scan) .
twoWay(boot;p1;p2;p3;p4;p5) .

```

Next, we have to modify some rules where appropriate. In particular, we only want to cover 3-sequences over symbols from `threeWay/1`. Hence, we rewrite rule

```

threeSeq(X,Y,Z) :- sym(X;Y;Z) , X != Y , Y != Z , X != Z .

```

Table 5.3: Test plan of size 8 for the laptop application obtained from an answer set of $\Pi_4^3(8)$.

row	event 1	event 2	event 3	event 4	event 5	event 6	event 7	event 8
1	p3(l)	p2(r)	p1(b)	p4	boot	appl	scan	p5
2	boot	p4	p1(r)	appl	p5	p3(l)	scan	p2(b)
3	boot	appl	scan	p1(r)	p2(b)	p4	p3(l)	p5
4	p1(r)	p2(b)	p5	p3(l)	boot	appl	scan	p4
5	boot	p3(b)	p5	p1(r)	appl	p4	p2(l)	scan
6	p4	boot	p2(b)	p5	appl	p1(l)	scan	p3(r)
7	boot	appl	scan	p5	p3(l)	p4	p2(b)	p1(r)
8	p5	boot	p2(l)	p4	p3(r)	appl	scan	p1(b)

into

```
threeSeq(X,Y,Z) :- threeWay(X;Y;Z), X != Y, Y != Z, X != Z.
```

To address two-way coverage of the symbols from predicate `twoWay/1`, we add two further rules:

```
covered(X,Y) :- hb(_,X,Y).
:- twoWay(X;Y), X != Y, not covered(X,Y).
```

The resulting program is denoted by $\Pi_2^3(n)$.

Program $\Pi_2^3(n)$ incorporates both forbidden configurations and redundant sequences. Respective SCAs can be obtained for $n = 8$ already. SCAs of size 8 are indeed optimal arrays, which follows from the observation that $\Pi_2^3(7)$ yields no answer set at all. It takes on average 0.1 seconds to compute the first answer set of a size 8 SCA when using `clasp` as ASP solver. Showing optimality, i.e., that no size 7 SCA exists, needs several minutes.

The solution approach of Kuhn et al. uses a precomputed (12, 7, 3)-SCA to account for the seven events `p1`, `p2`, `p3`, `p4`, `p5`, `scan`, and `appl`. In a post-processing step, rows that are not consistent with the ordering constraints (cf. Section 5.4.1) are replaced. However, this requires that further rows are added to preserve coverage. Note that this testing application was considered before the greedy algorithm was extended to directly express simple constraints [103]. In a further manual post-processing step, to account for the two-way coverage with respect to events `p1`, `p2`, `p3`, `p4`, `p5`, and `boot`, Kuhn et al. add `boot` as the first event of each row. Finally, an additional row is added, in which all events `p1`, `p2`, `p3`, `p4`, `p5` are arranged prior to `boot`, thereby obtaining strength 2 coverage between `boot` and events `p1`, `p2`, `p3`, `p4`, `p5`. The resulting array consists of 19 rows.

Note that using ASP enabled us to easily embed the additional requirements directly in the ASP program rather than employing an ad hoc and mostly manual approach. Furthermore, using ASP significantly reduced the size of the resulting SCA by eleven rows, hence cutting the size basically in half, cf. Table 5.3.

5.4.3 Adding Attributes to Events

The next problem elaboration that we consider is related to the way the peripherals are connected to the laptop. Devices `p1`, `p2`, and `p3` have to be connected to USB ports. Three ports are available: `left`, `right`, and `back`. In each test sequence, one port has to be assigned to a USB device.

For encoding this problem variant, we use predicate `port(N, X, Y)`, stating that USB device `X` is connected to port `Y` in row `N` of the array. This assignment should satisfy the following coverage criteria:

- (i) each USB device has to be connected to each port at least once, and
- (ii) connections to the ports after the boot event should be made in any possible order.

The above requirements can be formalised using few further rules.

In the following rules, we first specify the USB ports and devices. Then, it is expressed that each USB device is assigned to precisely one port in each test sequence. Finally, USB devices must not be connected to the same port in any sequence.

```
usbPort(right; left; back).
usbDevice(p1; p2; p3).
1{port(N, X, Y) : usbPort(Y)}1 :- row(N), usbDevice(X).
:- port(N, X, Y), port(N, Z, Y), X != Z.
```

Next, we state coverage criterion (i):

```
portCov(X, Y) :- port(N, X, Y).
:- usbDevice(X), usbPort(Y), not portCov(X, Y).
```

Lastly, we add rules for coverage criterion (ii):

```
portSeq(X, Y, Z) :- usbPort(X; Y; Z), X != Y, X != Z, Y != Z.

seqCov(N, X, Y, Z) :- hb(N, boot, X), hb(N, X, Y), hb(N, Y, Z).

pSeqCov(R, S, T) :- seqCov(N, X, Y, Z),
                    port(N, X, R), port(N, Y, S), port(N, Z, T).

:- portSeq(X, Y, Z), not pSeqCov(X, Y, Z).
```

```

% ASP encoding for the laptop example
sym(boot; p1; p2; p3; p4; p5; appl; scan).
row(1..n).
threeWay(p1; p2; p3; p4; p5; appl; scan).
twoWay(boot; p1; p2; p3; p4; p5).

% guess happens-before relation
1 { hb(N,X,Y), hb(N,Y,X) } 1 :- row(N), sym(X;Y), X != Y.
% happens-before is transitive
hb(N,X,Z) :- hb(N,X,Y), hb(N,Y,Z).
% happens-before is irreflexive
:- hb(N,X,X).

% check three-way and two-way coverage
covered(X,Y,Z) :- hb(N,X,Y), hb(N,Y,Z).
:- not covered(X,Y,Z), threeWay(X;Y;Z), X != Y, Y != Z, X != Z.
covered(X,Y) :- hb(_,X,Y).
:- twoWay(X;Y), X != Y, not covered(X,Y).

% excluded orderings
excluded(scan,appl).
excluded(appl,boot).
excluded(X,Z) :- excluded(X,Y), excluded(Y,Z).
:- hb(_,X,Y), excluded(X,Y).
covered(X,Y,Z) :- excluded(X,Y), sym(X;Y;Z).
covered(X,Y,Z) :- excluded(X,Z), sym(X;Y;Z).
covered(X,Y,Z) :- excluded(Y,Z), sym(X;Y;Z).

% coverage of USB ports
usbPort(right; left; back).
usbDevice(p1; p2; p3).
1{port(N,X,Y):usbPort(Y)}1 :- row(N), usbDevice(X).
:- port(N,X,Y), port(N,Z,Y), X != Z.
portCov(X,Y) :- port(N,X,Y).
:- usbDevice(X),usbPort(Y),not portCov(X,Y).
portSeq(X,Y,Z) :- usbPort(X;Y;Z), X != Y, X != Z, Y != Z.
seqCov(N,X,Y,Z) :- hb(N,boot,X),hb(N,X,Y), hb(N,Y,Z).
pSeqCov(R,S,T) :- seqCov(N,X,Y,Z),port(N,X,R),port(N,Y,S),port(N,Z,T).
:- portSeq(X,Y,Z), not pSeqCov(X,Y,Z).

% maximise covered 4-sequences
covered(W,X,Y,Z) :- hb(N,W,X),hb(N,X,Y), hb(N,Y,Z).
#maximize[covered(_,_,_,_)].

```

Figure 5.4: ASP encoding $\Pi_4^3(8)$.

Let us denote the resulting program by $\Pi_3^3(n)$.

Note that the additional conditions regarding the USB ports do not result in larger SCAs, still SCAs of size 8 can be obtained by computing the answer sets of $\Pi_3^3(8)$. Clearly, 8 is also an optimal bound. The runtime of the ASP solver is not affected by the additional requirements.

Kuhn et al. deal with the issue of USB ports by adding respective port assignments in a post-processing step once an SCA is computed. However, they do not provide details on which basis this is done, i.e., it is not clear if or in what sense they strove for systematic coverage.

5.4.4 Expressing Preferences

Any answer set of $\Pi_3^3(n)$ represents one admissible test plan for the application under test. Although each such SCA satisfies all of the requirements discussed so far, different SCAs could differ in their fault detection potential.

We next augment program $\Pi_3^3(n)$ by rules that state a preference relation among solutions, similar to program $\Pi_{grady}^3(\cdot, \cdot)$ from the previous section. In particular, although any SCA guarantees full three-way interaction coverage for some specified events, the degree of four-way coverage of events may differ from one SCA to another. We will use the number of covered 4-sequences as discrimination criterion regarding the quality of solutions and consequently prefer SCAs that cover more 4-sequences over SCAs that cover fewer.

We define program $\Pi_4^3(n)$ as $\Pi_3^3(n)$ augmented by the following rules:

```
covered(W, X, Y, Z) :- hb(N, W, X), hb(N, X, Y), hb(N, Y, Z).
#maximize[covered(_, _, _, _)].
```

The first rule defines which 4-sequences are covered, the second rule states that the number of covered 4-sequences should be maximised. The complete ASP encoding $\Pi_4^3(8)$ is given in Figure 5.4.

An SCA of size 8 corresponding to an answer set of $\Pi_4^3(8)$ is given in Table 5.3. In the computation of the SCA, `clasp` has been configured to optimise a solution until no improvements can be found for 15 minutes.

On the other hand, Kuhn et al. [103] have not handled preferences over solutions at all. The algorithm of Kuhn et al. is tailored for computing a single SCA. Thus, it may be hard to use such an algorithm to directly deal with optimisation issues.

This case study demonstrates that often generic SCAs cannot be used in a real world scenario without significant modifications. In general, such modifications lead to a considerable overhead or are not feasible at all. By using ASP, however, a test author has a tool to state different requirements relevant for individual scenarios. Often, this will need only little effort such as adding few rules.

5.5 Related Work

The ASP-based approach introduced in this chapter is the first account of an approach for directly generating SCAs in the presence of expressible constraints and problem elaborations. We note, however, that after the results of this chapter were published as a conference paper [57] and a journal article [19], our ideas were picked up soon by Banbara, Tamura, and Inoue [9]. They propose a constraint-programming encoding called the *incidence-matrix model* for generating SCAs which scales better than ours for $\text{SCAN}(s, 4)$ SCAs.

Closely related to our work are techniques for computing covering arrays (CAs), which we will review next. An overview of different approaches and tools for generating CAs is given by Grindal, Offutt, and Andler [84]. There, greedy algorithms that construct one row at a time are quite common. The most prominent representative is the AETG system [32]. Our greedy approach to compute SCAs is close in spirit to AETG-like algorithms since it also proceeds row by row. Also, meta-heuristics, like simulated annealing, taboo search, or genetic algorithms, have been applied for constructing CAs [35, 125].² Greedy algorithms usually scale well while meta-heuristics tend to produce arrays of smaller sizes [35]. However, neither greedy techniques nor meta-heuristics can guarantee optimal bounds.

As a complete method being able to establish optimality of arrays, different SAT encodings have been considered [89, 8]. Similar to our ASP encoding, SAT encodings allow to compute combinatorial designs as a whole. From a computational point of view, SAT and ASP are closely related, and ASP solvers like `clasp` employ many techniques also used by SAT solvers, like conflict-driven clause learning—in fact, `clasp` can be used as a SAT solver itself. A distinctive feature of ASP compared to SAT is the high-level modelling capabilities of ASP that allow to model problems concisely at the first-order level as demonstrated by our SCA encodings. SAT is certainly a promising approach for tackling problems described in Section 5.3, i.e., for computing SCAs and checking optimality of upper bounds. However, the problem variations discussed in Section 5.4 require a formalism that allows for elaboration-tolerant representations, which is not a characteristic feature of SAT. Regarding modelling, it is to mention that Hnich, Prestwich, Selensky, and Smith [89], as well as Banbara, Matsunaka, Tamura, and Inoue [8], initially considered constraint programming (CP) models, which are subsequently translated to SAT. Although this has not been considered, further constraints, at least forbidden tuples, could be incorporated rather easily into the CP model. A comparison of ASP and constraint (logic) programming (CLP) is given in a related article [46]. There, the authors conclude that ASP allows for more declarative and concise problem representations and is easier to learn for beginners than CLP.

The need for stating constraints and other user requirements in combinatorial interaction testing for real-world applications has been discussed by different authors [32, 88, 112,

²See also the overview articles by Grindal, Offutt, and Andler [84] and by Nie and Leung [119] for more details.

23, 36, 33, 34, 29]. The prevalent approach is to first generate a CA and then to delete and permute rows that are not consistent with certain requirements. The number of rows that need to be replaced can be vast and this approach can lead to a considerable increase of the array size [34]. This applies not only for CAs but for SCAs as well as we have illustrated in the previous section. Another common method requires remodelling of the specification [32, 112].

The tool PICT [36], also based on an AETG-like greedy algorithm, allows to directly express constraints; however, the details how this is realised are not accessible.

Cohen, Dwyer, and Shi [33, 34] introduced approaches that integrate techniques for generating covering arrays with SAT to deal with constraints. Forbidden tuples are represented as Boolean formulas and a SAT solver is used to compute models. They integrated SAT with greedy AETG-style algorithms and also with simulated annealing. Hence, their approach is closely related to our integration of ASP into a greedy procedure. Calvagna and Gargantini [29] follow a similar approach but they use an SMT solver instead of a SAT solver, which offers a richer language than plain SAT solvers. In their approach, constraints are stated as formal predicate expressions. Besides SMT, Calvagna and Gargantini also considered a model checker for verifying test predicates which would also be suitable for specifications involving temporal constraints and state transitions.

Bryce and Colbourn [23] distinguish forbidden tuples and tuples that should be avoided. They refer to the latter as soft constraints and they present an algorithm for generating CAs that avoids the violation of soft constraints. However, their algorithm cannot guarantee that certain tuples are avoided, hence it cannot deal with forbidden tuples or other hard constraints. Using ASP, soft constraints can be easily expressed by means of minimise or maximise statements. Some ASP solvers allow to express soft constraints even more directly in the form of weak constraints. We illustrated in the previous section how one can combine hard integrity constraints with soft constraints to express that uncovered 4-sequences should be avoided. For even more fine-grained modelling, ASP allows to assign different priorities to soft constraints and maximise, resp., minimise, statements. Recently, an ASP-based approach for computing covering arrays has been introduced that features multi-criterion optimisation [7].

Conclusion

The goal of this thesis is to investigate methods related to testing of ASP programs (and beyond). As such, our contribution falls within a line of research that aims at supporting the development process in ASP. Testing is well recognised as an integral part of conventional software testing and constitutes a major part of any software development project. A systematic study of this subject has been missing so far for ASP, however. Specific ASP features, like its declarative nature that allows for concise logic-based encodings, the lack of a control flow, and its non-determinism make many methods from conventional testing not directly applicable. Although testing is different for ASP, it is nonetheless important, and this is not only to ease the process of writing programs: If ASP is to be used within larger software projects, testing methods to ensure our trust in the resulting product cannot stop at the ASP modules. Respective methods how to address this issue are clearly needed for ASP as well.

Let us briefly walk through our main contributions and how they fit together. As a basis for a systematic study, we introduced formal definitions for relevant testing concepts: this includes test inputs and outputs for uniform problem encodings based on language signatures as well as test cases for ASP and when they pass and fail. Related to test inputs, we discussed the issue of generating test inputs that are admissible, i.e., comply with a program's preconditions. We then investigated how effective it is to test programs with inputs from a limited scope. We arrived at the conclusion that we can indeed find most program errors by restricting testing to relatively small inputs only—a phenomenon known as the small scope hypothesis in conventional programming. This has important implications: We can apply testing methods to programs after grounding them over a small domain instead of directly considering the non-ground versions. One such testing method is bounded-exhaustive testing based on suitable equivalence checkers. This is the method that we used to empirically evaluate the small scope hypothesis in this work. Another method is random testing: We presented a respective approach that is based on XOR constraints for generating a nearly uniform distribution of test inputs.

Random testing is not only a simple yet effective testing method by itself but is also a basis for comparison for more sophisticated testing methods. In particular, we studied structure-based testing for ASP where program elements guide the selection of inputs and showed its advantage over simple random testing for certain problems both theoretically and empirically.

Towards a more intertwined approach for testing and program development, we introduced the annotation language LANA for ASP. This language offers various ways to express meta-information for rules and other language elements which can be used to support and ease the development process, test and verify programs, and to eliminate many sources for common programmer errors. Furthermore, LANA allows to formulate unit tests for individual program blocks and to develop programs in a test-driven fashion. Features of our annotation language have been integrated into **SeaLion**, an IDE for ASP, as well as used for interpreting answer sets in natural language [63].

On the one hand, we discussed and evaluated a set of methods to test ASP programs. On the other hand, there are various concrete tools that have been developed in the course of our work on testing methods:

- a mutation generator that can be used for mutation testing, i.e., evaluating a test suite regarding its potential to distinguish a program under test from mutated versions thereof,
- the tool **Harvey** to generate and execute random test cases,
- **ASPDoc** which takes an answer-set program, interprets the meta-comments, and automatically generates an HTML file documenting the program, and
- **ASPUnit**, an implementation of a unit-testing framework based on annotations found in a program.

We note that these tools support ASP solver dialects that were state-of-the art at the time the tools were developed and published. Although language dialects are ever changing which can render tools outdated, we believe that the underlying principles of our testing methods are not affected.

To address the question of which methods and tools to actually employ for testing, we provided a broad arsenal of options, depending on the situation. Often, we can designate an answer-set program to serve a test oracle. This is the case, e.g., for testing student assignments in programming courses, testing submissions in answer-set programming challenges, or iteratively refining answer-set programs to meet performance goals. In such situations, exhaustively testing programs within a limited scope proved to be very effective. If programs are too big and even a limited scope for inputs is not workable, we can still use random testing to generate a large number of inputs and test them against the reference implementation. If we do not have a reference implementation, testing can become significantly more expensive. Assume, for example, that test outcomes

have to be verified by hand. In such cases, carefully selecting a smaller number of test inputs based on our structure-based approach seems viable. The practice of iteratively and incrementally develop programs is reflected best by continuously using unit test as provided by LANA, thus following a more test-driven development approach.

Many methods we studied here rely on ASP itself. We used ASP for test oracles, to specify preconditions of programs and respective input generators, to determine test verdicts in random and structure-based testing, to show program equivalence in bounded-exhaustive testing, and to craft unit tests and testing assertions in LANA. Indeed, we were using ASP itself for testing in ASP. But we did not stop there as methods based on ASP can be used for testing problems outside of ASP as well. For example, our method for random testing is potentially useful to generate structurally complex test inputs that are specified using ASP but used by other software components. Also, we dedicated an entire chapter to ASP for event sequence testing which is a combinatorial method to test systems where faults are triggered by events that occur in a particular order. There, we introduced a novel technique for computing sequence-covering arrays that allows to easily incorporate problem elaborations that are indispensable for testing real-world applications.

In general, ASP has a strong potential for specifying test criteria in combinatorial interaction testing as witnessed by work on ASP for generating covering arrays [7] or ASP-based combinatorial methods to address testing of concurrent programs in recent work [126]. While ASP is well established in other communities as a method to address problems from the area of artificial intelligence and knowledge representation, there is too little awareness of ASP in the software-engineering community. Hence, our work on SCAs can also be seen as a step towards promoting ASP as an approach to tackle challenging problems in the realm of combinatorial testing. Besides improving the state-of-the-art of event sequence testing, our results also show that ASP provides a viable tool that enables a tester to rapidly specify problems and to experiment with different formulations at a purely declarative level. Off-the-shelf ASP solvers can then be used for computing solutions without the need of post-processing steps or developing dedicated algorithms.

Concerning future work in this direction, versions of SCAs for different testing applications, like testing of concurrent programs where the order of shared variable accesses is crucial for triggering certain bugs that are otherwise hard to evoke [113, 114], would be a worthwhile endeavour. Also, not only the problem of statically generating suitable designs can be addressed, but also to investigate ways to do this in an *online fashion*, where an ASP solver is coupled with a scheduler to improve coverage with respect to different interleaving metrics. Such an online approach would also allow to deal with, e.g., exceptional events in a more interactive testing environment. Moreover, developing support tools for a tester regarding the modelling of a system's test space without requiring expert knowledge on ASP, by exploiting a dedicated front-end language for ASP tailored to specific testing domains, would be another interesting open issue.

Appendix A:

ASP Benchmark Problems

We describe benchmark instances, including input and output signatures, preconditions, encodings, and input generators, from the third ASP solver competition [28]. Problem descriptions are taken from the competition's web page www.mat.unical.it/aspcomp2011. The input generators are our contribution.

Problems in P

Reachability

The objective of the problem REACHABILITY is to exhibit a path between two dedicated nodes in a directed graph. The input of this problem is a graph specified via relation `edge/2` and one fact `query(a,b)` that determines the two nodes `a` and `b` for which reachability should be decided. The output is the fact `query(a,b)`, if a path exists, and no answer set otherwise. Reachability is a representative for problems solvable in polynomial time. The central aspect is to efficiently compute the transitive closure of the edge relation of the input graph.

Input signature: `edge/2, query/2`

Output signature: `query/2`

Preconditions: Precisely one fact `query(a,b)` defines the two nodes `a` and `b` for which reachability should be decided.

Encoding:

```
reaches(X,Y) :- edge(X,Y) .
reaches(X,Y) :- edge(X,Z), reaches(Z,Y) .
:- query(X,Y), not reaches(X,Y) .
```

Input generator:

```

% Reachability
% s ... max number of nodes

% nnodes/1 ... max. number of nodes of the input graph
dom(1..s).
1 { nnodes(X) : dom(X) } 1.

{ edge(X,Y) } :- dom(X;Y), X <= N, Y <= N, nnodes(N).
1 { query(X,Y) : dom(X;Y) } 1.

#hide.
#show nnodes/1.    % EVA for input generator
#show edge/2.     % input
#show query/2.    % input and output
    
```

Grammar Based Information Extraction

For the grammar-based information extraction problem GRBSDINFEXTRCT, the goal is to decide whether a given string is element of a context free language that specifies arithmetic expressions. The grammar is defined as follows:

```

<equation> -> <expression> <greater> <sign_1> <number>
<expression> -> <sign_1> <term> <term_1>
<term_1> -> <sign> <term> <term_1> | epsilon
<term> -> <number> | <op_bracket> <expression> <cl_bracket>
<sign> -> <plus> | <minus>
<sign_1> -> <sign> | epsilon
<number> -> <digit> <number_1>
<number_1> -> <digit> <number_1> | epsilon
<plus> -> "+"
<minus> -> "-"
<greater> -> ">"
<op_bracket> -> "("
<cl_bracket> -> ")"
<digit> -> "0" | "1" | "2" | "3" | "4" |
           "5" | "6" | "7" | "8" | "9"
    
```

The non-terminals, especially `<equation>`, `<expression>`, `<term>`, `<number>`, have an associated value, usually representing a numerical value, and, in the case of `<equation>`, a binary value. Given an equation (i.e., a sequence of terminals), the problem is determining whether it belongs to the language defined by this grammar (starting from `<equation>`) and if so, whether the value associated to the axiom is “true” proving that the equation is true as well.

An instance of this problem contains a fact (of arity two) for each character of the equation. The first argument is a constant from the set $\{p, m, g, o, c, 0, 1, \dots, 9\}$ representing, respectively, a terminal of the alphabet, and the second one is its position (starting from 1).

If the string described by the input does not belong to the language (i.e., is not a well-formed expression), or if it is a well-formed but false expression, then the instance should be unsatisfiable. If the string is well-formed and true, the witness should consist of

```
sol. values(S1, N1, S2, N2) .
```

Here, the value of two expressions are represented by $(S1, N1)$ and $(S2, N2)$, where S_i is a sign (p or m , standing for $+$ and $-$) and N_i a positive integer in the set $\{0, 1, 2, \dots\}$. The value zero is represented as $(p, 0)$. The number represented by $(S1, N1)$ should be larger than $(S2, N2)$.

Input signature: char/2

Output signature: values/4, sol/0

Preconditions: Predicate `char/2` specifies an input string using consecutive integers starting at 1 as its first argument; the second argument is a constant from the set $\{p, m, g, o, c, 0, 1, \dots, 9\}$.

Encoding:

```
is_digit(0..9) .

digit(X, Y, C) :- char(C, Y), is_digit(C), X = Y - 1.
dig(X) :- digit(Fv1, X, Fv2) .

%%%%%%%%%% <sign> -> <plus> | <minus>
sign(X, p) :- char(p, X) .
sign(X, m) :- char(m, X) .

greater(X) :- char(g, X) .

multiply(p, p, p) .
multiply(p, m, m) .
multiply(m, p, m) .
multiply(m, m, p) .

opposite(m, p) .
opposite(p, m) .
```

```

%%%%%%%%%%%%% <number> -> <digit>+
%%%%%%%%%%%%% <digit> -> '0' | ... | '9'
num_1(X, Y, Val) :- digit(X, Y, Val), not dig(X).
num_1(X, Z, NewVal) :- num_1(X, Y, Val_1), digit(Y, Z, Val_2),
    NewVal_1 = 10 * Val_1,
    NewVal = NewVal_1 + Val_2.
num(X, Y, Val) :- num_1(X, Y, Val), Z = 1 + Y, not dig(Z).

par_expr(X, W, S, Val) :- char(o, Y), expr(Y, Z, S, Val),
    W = Z + 1, char(c, W), X = Y - 1.

sign_term(X, Z, S3, Val) :- sign(Y, S1),
    par_expr(Y, Z, S2, Val),
    multiply(S1, S2, S3),
    X = Y - 1.
sign_term(X, Z, S, Val) :- sign(Y, S),
    num(Y, Z, Val),
    X = Y - 1.

expr(X, Y, p, Val) :- char(o, X), num(X, Y, Val).
expr(0, Y, p, Val) :- num(0, Y, Val).
expr(X, Y, S, Val) :- par_expr(X, Y, S, Val).
expr(X, Y, S, Val) :- sign_term(X, Y, S, Val).

expr(X, Z, S, Val) :- expr(X, Y, S, Val_1),
    sign_term(Y, Z, S, Val_2),
    Val = Val_1 + Val_2.

expr_1(X, Z, S1, S2, Val_1, Val_2) :- expr(X, Y, S1, Val_1),
    sign_term(Y, Z, S2, Val_2),
    opposite(S1, S2).

expr(X, Y, p, 0) :- expr_1(X, Y, Fv1, Fv2, Val, Val).
expr(X, Y, S1, Val) :- expr_1(X, Y, S1, Fv1, Val_1, Val_2),
    Val_1 > Val_2, Val = Val_1 - Val_2.
expr(X, Y, S2, Val) :- expr_1(X, Y, Fv1, S2, Val_1, Val_2),
    Val_1 < Val_2, Val = Val_2 - Val_1.

%%%%%%%%%%%%% <greater> -> '>'
%%%%%%%%%%%%% <sign_1> -> <sign> | epsilon
greater_sign(X, Y, p) :- greater(Y), digit(Y, Fv1, Fv2),
    X = Y - 1.
greater_sign(X, Z, S) :- greater(Y), Z = Y + 1, sign(Z, S),
    X = Y - 1.

%%%%%%%%%%%%% <equation> -> <expression> <greater> <sign_1> <number>
values(S_left, Val_left, S_right, Val_right) :-
  
```

```

    expr(0, Y, S_left, Val_left), greater_sign(Y, Z, S_right),
    num(Z, W, Val_right).

```

```

sol :- values(p, Val_left, p, Val_right), Val_left > Val_right.
sol :- values(m, Val_left, m, Val_right), Val_left < Val_right.
sol :- values(p, Val_left, m, Val_right).

```

```

:- not sol.

```

Input generator: Note that the range of the domain for the second argument of `char/2` has been restricted: Instead of all integer digits, only $\{0, 1, 2\}$ are considered. Thus, the size of the input space is drastically reduced without sacrificing much potential for fault detection.

```

% Grammar-Based Information Extraction
% s ... maximal word length

dom(p;m;g;o;c).
dom(0..2).      % restricted domain for digits

int(1..s).
1 { length(X) : int(X) } 1.
1 { char(C,P) : dom(C) } 1 :- int(P), P >= 1, P <= X, length(X).

#hide.
#show length/1.    % EVA for input generator
#show char/2.     % input
#show sol/0.      % output
#show values/4.   % output

```

Hydraulic Leaking

HYDRAULICLEAKING is a planning problem based on a graph that represents the simplified hydraulic system of the Space Shuttle. The nodes are tanks, jets, or junctions. The edges are valves that can be leaking. The goal is to find a sequences of valves that need to be opened to pressurise a certain jet were leaking valves should be avoided as far as possible.

Input signature: tank/1, jet/1, junction/1, valve/1, link/3, numValves/1, full/1, leaking/1, goal/1

Output signature: switchon/2

Predonctions:

- Each node of the input graph is either a tank, a jet, or a junction.

- Precisely one fact `numValves/1` specifies the total number of valves.
- Between any two nodes p, q , there is at most one link labelled (p, q) ; (p, q) is the name of the respective valve, and the total number of valves matches the value specified by `numValves/1`.
- Tanks can be full and valves might be leaking; only tanks can be full, only valves can be leaking.
- No tank can be reached from another tank in the input graph; it cannot be the case that a jet is not reachable from some tank in the input graph.
- Any jet can be set as goal but at least one has to be chosen; only jets can be set as goal.

Encoding:

```

% graph nodes
node(N) :- jet(N). node(N) :- junction(N). node(N) :- tank(N).

% Lengths of paths to goal, bounded by the number of valves,
% recording also the number of leaking valves on the path.
dist(J,0,0) :- goal(J).
dist(N,DN,LN) :- dist(K,DK,LK), link(N,K,V), DN=DK+1,
    LN=LK+1, numValves(NV), DN <= NV, leaking(V).
dist(N,DN,LK) :- dist(K,DK,LK), link(N,K,V), DN=DK+1,
    numValves(NV), DN <= NV, not leaking(V).

% Minimum leaking distance of a node to the goal node.
nodemindist(N,DN,LDN) :- node(N), minLkDist(N,LDN),
    minDist(N,DN,LDN).
minLkDist(N,LDN) :- dist(N,Fv1,LDN), not existLdnLessThan(N,LDN).
existLdnLessThan(N,LDN) :- dist(N,Fv1,LDN1), dist(N,Fv2,LDN1),
    LDN1 < LDN.
minDist(N,DN,LDN) :- dist(N,DN,LDN), not existDnLessThan(N,DN,LDN).
existDnLessThan(N,DN,LDN) :- dist(N,DN,LDN1), dist(N,DN1,LDN),
    DN1 < DN.

% Minimum leaking distance of a full tank to the goal node.
fulltankmindist(T,DT,LDT) :- tank(T), full(T),
    nodemindist(T,DT,LDT).

% Fail if no full tank can be reached from the goal.
reachablefulltankexists :- fulltankmindist(T,DT,LDT).
:- not reachablefulltankexists.

% The full tanks and their minimum leaking distances to the
% goal node, which have the minimum leaking distance over all

```

```

% full tanks.
bestfulltankldist(T,SD,SDL) :- fulltankmindist(T,SD,SDL),
                               not existFTLLessThan(SDL).
existFTLLessThan(SDL) :- fulltankmindist(Fv1,Fv2,SDL),
                          fulltankmindist(Fv3,Fv4,SDL1),
                          SDL1<SDL.

% Among the full tanks with minimum leaking distance choose
% those with minimum distance from the goal.
bestfulltankdist(T,SD,SDL) :- bestfulltankldist(T,SD,SDL),
                              not existFTLessThan(SD,SDL).
existFTLessThan(SD,SDL) :- fulltankmindist(Fv1,SD,SDL),
                            fulltankmindist(Fv2,SD1,SDL),
                            SD1<SD.
goodtank(T) :- bestfulltankdist(T,Fv1,Fv2).

% "Choose" the lexicographically smallest good tank, as any will do.
nbesttank(T) :- goodtank(T), goodtank(T1), T1 < T.
besttank(T) :- goodtank(T), not nbesttank(T).

% Now go back to the goal, starting from the chosen tank.
reached(T,SD,SDL) :- bestfulltankdist(T,SD,SDL).

% In any step, choose the smallest valve among those linking the
% reached node of distance D to other nodes of distance D-1, tracking
% also the leaking distance (stays equal for non-leaking valves,
% decreases by one for leaking valves).
cand(V1,D1,LD1) :- reached(N,D,LD), LD1=LD-1, D1=D-1, link(N,N1,V1),
                  nodemindist(N1,D1,LD1), leaking(V1).
cand(V1,D1,LD)  :- reached(N,D,LD), D1=D-1, link(N,N1,V1),
                  nodemindist(N1,D1,LD), not leaking(V1).
nsmallestvalve(V2,D1,LD2) :- cand(V1,D1,LD1), cand(V2,D1,LD2),
                             V1 < V2.
smallestvalve(V1,D1,LD1) :- cand(V1,D1,LD1),
                             not nsmallestvalve(V1,D1,LD1).

% Reverse order for switching on (moving from tank to jet, starting
% from 0).
switchon(V,DN) :- smallestvalve(V,D,Fv1),
                 bestfulltankdist(Fv2,BD,Fv3), DX=BD-D, DN=DX-1.

% Now choose the smallest node linked by the chosen
% valve as being reached.
nsmallestnodeforvalve(N1,D1,LD1) :- reached(N,D,LD),
                                     smallestvalve(V,D1,LD1),
                                     nodemindist(N1,D1,LD1),
                                     nodemindist(N2,D1,LD1),
                                     N1 > N2, link(N,N1,V),
                                     link(N,N2,V).
  
```

```

smallestnodeforvalve(N1,D1,LD1) :- smallestvalve(V,D1,LD1),
                                   reached(N,D,LD), D1=D-1, link(N,N1,V),
                                   not nsmallestnodeforvalve(N1,D1,LD1).

reached(N,D,LD) :- smallestnodeforvalve(N,D,LD).

```

Input generator: To avoid isomorphic test inputs, some symmetry breaking was incorporated. In particular, we enforce that identifiers for tanks are never smaller than identifiers for jets, and identifiers for jets are never smaller than identifiers for junctions.

```

% Hydraulic Leaking
% s ... max number of nodes
% t ... max number of valves

dom(1..s).
{ tank(X), jet(X), junction(X) } 1 :- dom(X).

n(X) :- tank(X).
n(X) :- jet(X).
n(X) :- junction(X).

:- tank(X), jet(Y), X < Y.
:- jet(X), junction(Y), X < Y. % break symmetries.

{ link(X,Y,v(X,Y)) } :- n(X), n(Y).
valve(V) :- link(X,Y,V).
:- t+1 { valve(V) }.

domV(1..t).
numValves(X) :- X { valve(V) } X, domV(X).

{ full(X) } :- tank(X).
{ leaking(X) } :- valve(X).

linkC1(X,Y) :- link(X,Y,V).
linkC1(X,Z) :- linkC1(X,Y), linkC1(Y,Z).
:- tank(N1), tank(N2), linkC1(N1,N2).
:- jet(X), not supp(X).
supp(X) :- linkC1(T,X), tank(T).

{ goal(X) } :- jet(X).
:- not 1 { goal(X) }.

#hide.
#show tank/1.      %input
#show jet/1.      %input
#show junction/1. %input

```

```

#show valve/1.      %input
#show link/3.      %input
#show numValves/1. %input
#show full/1.      %input
#show leaking/1.   %input
#show goal/1.      %input
#show switchon/2.  %output
  
```

Hydraulic Planning

HYDRAULICPLANNING is a planning problem based on a graph that represents the simplified hydraulic system of the Space Shuttle. The nodes are tanks, jets, or junctions. The edges are valves. Valves can be stuck in the closed position. The goal is to find a sequences of valves that need to be opened to pressurise a certain jet were stuck valves cannot be opened.

Input signature: tank/1, jet/1, junction/1, valve/1, link/3, numValves/1, full/1, stuck/1, goal/1

Output signature: switchon/2

Preconditions:

- Each node of the input graph is either a tank, a jet, or a junction.
- Precisely one fact numValves/1 specifies the total number of valves.
- Between any two nodes p, q , there is at most one link labelled (p, q) ; (p, q) is the name of the respective valve, and the total number of valves matches the value specified by numValves/1.
- Tanks can be full and valves might be leaking; only tanks can be full, only valves can be leaking.
- No tank can be reached from another tank in the input graph; it cannot be the case that a jet is not reachable from some tank in the input graph.
- Any jet can be set as goal but at least one has to be chosen; only jets can be set as goal.

Encoding:

```

% graph nodes
node(N) :- jet(N). node(N) :- junction(N). node(N) :- tank(N).

% Useable links are those without stuck valves.
  
```

```

useablelink(N1,N2,V) :- link(N1,N2,V), not stuck(V).

% Lengths of useable paths (ie w/o stuck valves) to goal, bounded by
% the number of valves.
dist(J,0) :- goal(J).
dist(N,DN) :- dist(K,DK), useablelink(N,K,Fv1), DN=DK+1,
              numValves(NV), DN <= NV.

% Mminimum distance of a node to the goal node.
nodemindist(N,DN) :- node(N), dist(N,DN),
                    not existDnLessThan(N,DN).
existDnLessThan(N,DN) :- dist(N,DN),
                        dist(N,DN1), DN1<DN.

% Minimum distance of a full tank to the goal node.
fulltankmindist(T,DT) :- tank(T), full(T), nodemindist(T,DT).

% Fail if no full tank can be reached from the goal.
reachablefulltankexists :- fulltankmindist(T,D).
:- not reachablefulltankexists.

% The full tanks and their minimum distances to the goal node,
% which have the minimum distance over all full tanks.
bestfulltankdist(T,SD) :- fulltankmindist(T,SD),
                        not existFTLessThan(SD).
existFTLessThan(SD) :- fulltankmindist(Fv1,SD),
                      fulltankmindist(Fv2,SD1), SD1<SD.
goodtank(T) :- bestfulltankdist(T,Fv1).
bestdist(D) :- bestfulltankdist(Fv1,D).

% "Choose" the lexicographically smallest good tank, as any
% will do.
nbesttank(T) :- goodtank(T), goodtank(T1), T1 < T.
besttank(T) :- goodtank(T), not nbesttank(T).

% Now go back to the goal, starting from the chosen tank.
reached(T,SD) :- bestfulltankdist(T,SD), besttank(T).

% In any step, choose the smallest non-stuck valve
% among those linking the reached node of distance D
% to other nodes of distance D-1.
nsmallestvalve(V2,D1) :- reached(N,D), D1=D-1,
                        useablelink(N,N1,V1),
                        useablelink(N,N2,V2), V1 < V2,
                        nodemindist(N1,D1),
                        nodemindist(N2,D1).

smallestvalve(V,D1) :- reached(N,D), D1=D-1,
                      useablelink(N,N1,V), nodemindist(N1,D1),

```

```

    not nsmallestvalve(V,D1).

% Reverse order for switching on (moving from tank to jet,
% starting from 0).
switchon(V,DN) :- smallestvalve(V,D), bestdist(BD),
                 DX=BD-D, DN=DX-1.

% Now choose the smallest node linked by the chosen
% valve as being reached.
nsmallestnodeforvalve(N1,D1) :- smallestvalve(V,D1), reached(N,D),
                                D1=D-1, useablelink(N,N1,V),
                                useablelink(N,N2,V), N1 > N2.

smallestnodeforvalve(N1,D1) :- smallestvalve(V,D1), reached(N,D),
                                D1=D-1, useablelink(N,N1,V),
                                not nsmallestnodeforvalve(N1,D1).

reached(N,D) :- smallestnodeforvalve(N,D).

```

Input generator: To avoid isomorphic test inputs, some symmetry breaking was incorporated. In particular, we enforce that identifiers for tanks are never smaller than identifiers for jets, and identifiers for jets are never smaller than identifiers for junctions.

```

% Hydraulic Planning
% s ... max number of nodes
% t ... max number of valves

dom(1..s).
{ tank(X), jet(X), junction(X) } 1 :- dom(X).

n(X) :- tank(X).
n(X) :- jet(X).
n(X) :- junction(X).

:- tank(X), jet(Y), X < Y.
:- jet(X), junction(Y), X < Y. % break symmetries.

{ link(X,Y,v(X,Y)) } :- n(X), n(Y).
valve(V) :- link(X,Y,V).
:- t+1 { valve(V) }.

domV(1..t).
numValves(X) :- X { valve(V) } X, domV(X).

{ full(X) } :- tank(X).
{ stuck(X) } :- valve(X).

```

```
linkCl(X,Y) :- link(X,Y,V).
linkCl(X,Z) :- linkCl(X,Y), linkCl(Y,Z).
:- tank(N1), tank(N2), linkCl(N1,N2).
:- jet(X), not supp(X).
supp(X) :- linkCl(T,X), tank(T).

{ goal(X) } :- jet(X).
:- not 1 { goal(X) } 1.

#hide.
#show tank/1.           % input
#show jet/1.           % input
#show junction/1.      % input
#show valve.           % input
#show link/3.          % input
#show numValves/1.     % input
#show full/1.          % input
#show stuck/1.         % input
#show goal/1.          % input
#show switchon/2.      % output
```

Stable Marriage

Given n men and n women, each person has ranked all members of the opposite sex with a unique number between 1 and n in order of preference. The objective of STABLEMARRIAGE is to find a matching between the n men and the n women such that for no pair (m, w)

- (i) man m is matched with a woman w' different from w and such that m prefers w over w' ;
- (ii) woman w is matched with a man m' different from m and such that w prefers m over m' , or m and m' are equally preferred by w .

This variant is known to be polynomially solvable.

Input signature: womanAssignsScore/3, manAssignsScore/3

Output signature: match/2

Preconditions: There are n men and n women; each man, resp., woman, assigns one number in the range $\{0, \dots, n\}$ to each woman, resp., man.

Encoding: Note that although the problem is known to be solvable in polynomial time, the encoding involves non-deterministic choices and is thus suitable not only to decide if a matching exists but also to reveal all admissible matchings.

```

% guess matching
match(M,W) :- manAssignsScore(M,Fv1,Fv2),
              womanAssignsScore(W,Fv3,Fv4), not nonMatch(M,W).
nonMatch(M,W) :- manAssignsScore(M,Fv1,Fv2),
                 womanAssignsScore(W,Fv3,Fv4), not match(M,W).

% no polygamy
:- match(M1,W), match(M,W), M != M1.
:- match(M,W), match(M,W1), W != W1.

% no singles
jailed(M) :- match(M,Fv1).
:- manAssignsScore(M,Fv1,Fv2), not jailed(M).

% strong stability condition
:- match(M,W1), manAssignsScore(M,W,Smw), W1 != W,
   manAssignsScore(M,W1,Smw1), Smw > Smw1,
   match(M1,W), womanAssignsScore(W,M,Swm),
   womanAssignsScore(W,M1,Swm1), Swm >= Swm1.

```

Input generator:

```

% Stable Marriage
% s ... max number of persons

dom(1..s).
1 { maxS(X) : dom(X) } 1.

1 { womanAssignsScore(W,M,S) : dom(S) } 1 :- dom(W;M), W <= V,
                                              M <= V, maxS(V).
:- womanAssignsScore(W,M,S), maxS(X), S > X.

1 { manAssignsScore(W,M,S) : dom(S) } 1 :- dom(W;M), W <= V,
                                              M <= V, maxS(V).
:- manAssignsScore(W,M,S), maxS(X), S > X.

#hide.
#show maxS/1.           % EVA for input generator
#show womanAssignsScore/3. % input
#show manAssignsScore/3. % input
#show match/2.         % output

```

Partner Units Polynomial

A given people counting system includes three types of components, namely door sensors, zones, and communication units. The problem requirements of PARTNERUNITSPOLY are:

1. Each zone as well as each door sensor must be connected to exactly one unit.
2. Each unit can control at most two door sensors and at most two zones. If a unit controls a door sensor that contributes to a zone controlled by another unit, then the two units must be connected directly, i.e., one unit becomes a partner unit of the other and vice versa.
3. Each unit can have at most two partner units.

The solution of PARTNERUNITSPOLY is defined as follows: Given a consistent configuration of door sensors and zones (encoded in the binary predicate `zone2sensor/2`) and a set of available units (`comUnit/1`), a maximum number of allowed partnerunits (`maxPU/1`), equal to two in this case, find a valid assignment of units that satisfies all requirements.

Input signature: `comUnit/1, maxPU/1, zone2sensor/2`

Output signature: `unit2zone/2, unit2sensor/2, partnerunits/2`

Preconditions: `comUnit/1` specifies the set of available units as integers, `maxPU(2)` fixes the maximum number of allowed partnerunits to two, and `zone2sensor/2` is a mapping from sensors to zones modelled as integers.

Encoding:

```
zone(Z) :- zone2sensor(Z,D).
doorSensor(D) :- zone2sensor(Z,D).

{ unit2zone(U,Z) } :- zone(Z), comUnit(U).
:- unit2zone(U,Z1), unit2zone(U,Z2), unit2zone(U,Z3),
   Z1 != Z2, Z2 != Z3, Z1 != Z3.
:- unit2zone(U1,Z), unit2zone(U2,Z), U1 != U2.
atLeastOneUnit(Z) :- unit2zone(Fv1,Z).
:- zone(Z), not atLeastOneUnit(Z).

{ unit2sensor(U,D) } :- doorSensor(D), comUnit(U).

:- unit2sensor(U,D1), unit2sensor(U,D2), unit2sensor(U,D3),
   D1 != D2, D2 != D3, D1 != D3.
:- unit2sensor(U1,D), unit2sensor(U2,D), U1 != U2.
atLeastOneSensor(D) :- unit2sensor(Fv1,D).
```

```

:- doorSensor(D), not atLeastOneSensor(D).

partnerunits(U,P) :- unit2zone(U,Z), unit2sensor(P,D),
                    zone2sensor(Z,D), U != P.
partnerunits(U,P) :- partnerunits(P,U).

:- partnerunits(U,P1), partnerunits(U,P2), partnerunits(U,P3),
   P1 != P2, P2 != P3, P1 != P3.

```

Input generator:

```

% Partner Units Polynomial
% s ... max number of units

domU(1..s).
domX(1..2*s).

1 { maxU(X) : domU(X) } 1.
comUnit(X) :- domU(X), maxU(U), X <= U.

maxPU(2).
{ zone2sensor(Z,S) } :- domX(Z;S), maxU(M), Z <= 2*M, S <= 2*M.

#hide.
#show maxU(X).          % EVA for input generator
#show comUnit/1.       % input
#show maxPU/1.         % input
#show zone2sensor/2.   % input
#show unit2zone/2.     % output
#show unit2sensor/2.   % output
#show partnerunits/2.  % output

```

Problems in NP with a Tight Encoding

Fast Food Optimality Check

The objective of `FASTFOODOPTCHECK` is to verify that a given solution of the fast food optimisation problem is optimal or to find a better solution, otherwise. Restaurants are located on a highway, where `restaurant(N,K)` specifies that restaurant with name `N` is located at kilometer `K` of the highway. Furthermore, depots are built at some restaurants, specified via `debot(N,K)`. Each restaurant is supplied by the nearest depot, in case of equidistant depots, precisely one supplies the restaurant. Any solution to `FASTFOODOPTCHECK` should encode an alternative configuration of the depots, specified via `altdepot/2` such that the average distance between depots and supplied restaurants is smaller than in the current configuration.

Input signature: `restaurant/2, depot/2`

Output signature: altdepot/2

Preconditions:

- At each kilometer, there are at most two restaurants; any restaurant is found at most once at the highway.
- At each restaurant, there is at most one depot.

Encoding:

```

% Auxiliary predicate: All distances between restaurant locations.
distance(X,L,Y):- restaurant(R,X), restaurant(S,L), X>L, Y = X - L.
distance(X,L,Y):- restaurant(R,X), restaurant(S,L), X<=L, Y = L - X.

% The supply distance for each restaurant for the candidate solution
% in the input.
serves(Rname,Dist) :- restaurant(Rname,RK), depot(Dname,DK),
                        distance(RK,DK,Dist),
                        D = #min [ distanceH1(DK1,RK,Y) = Y ],
                        D == Dist.

distanceH1(DK1,RK,Y) :- distance(DK1,RK,Y), depot(Dname1,DK1).

% Each restaurant may be an alternative depot or not.
{ altdepot(R,K) } :- restaurant(R,K).

% The number of alternative depots must be equal to the number of
% depots in the input.
:- N1 = #count{depot(D,K)}, N2 = #count{altdepot(D1,K1)}, N1 != N2.

% The supply distance for each restaurant for the alternative
% solution.
altserves(Rname,Dist) :- restaurant(Rname,RK), altdepot(Dname,DK),
                        distance(RK,DK,Dist),
                        D = #min [ distanceH2(DK1,RK,Y) = Y ],
                        D == Dist.

distanceH2(DK1,RK,Y) :- distance(DK1,RK,Y), altdepot(Dname1,DK1).

% Accept an alternative solution only if its supply costs are less
% than the supply costs for the input candidate.
:- Cost1 = #sum [ serves(R,Dist) = Dist ],
   Cost2 = #sum [ altserves(R1,Dist1) = Dist1 ], Cost1 <= Cost2.

```

Input generator: To shrink the test-input search space, some symmetry breaking is applied. In particular, we use consecutive integers $1, \dots, q$ to denote q restaurants, and we enforce that restaurant names are given in ascending order when we follow the highway.

```

% Fastfood Optimality Check
% s ... kilometer

dom(1..s).
{ restaurant(X,Y) : dom(X) } 2 :- dom(Y).
:- restaurant(RN,RK1), restaurant(RN,RK2), RK1 < RK2.

% breaking symmetries
:- N = #count{restaurant(X,Y)}, restaurant(X1,Y1), X1 > N.
:- restaurant(RN1,RK1), restaurant(RN2,RK2), RK1 < RK2, RN1 > RN2.

{ depot(X,Y) } 1 :- restaurant(X,Y).

#hide.
#show restaurant/2. % input
#show depot/2. % input
#show altdepot/2. % output

```

Knight Tour

In KNIGHTTOUR, we need to find a tour for the knight piece that starts at any square, travels all squares, and comes back to the origin, following the rules of chess. Input predicate `size/1` specifies the size of the chess board and `givenmove/4` represents moves that the knight tour has to contain. The output is a set of moves of the knight represented via `move/4`.

Input signature: `size/1, givenmove/4`

Output signature: `move/4`

Predonctions: Precisely one fact `size/1` represents the size of the chessboard. All moves specified via `givenmove/4` have to be valid moves according to the rules of chess.

Encoding:

```

% Knight Tour

% Input:
% size(N): chessboard is NxN
% givenmove(X1,Y1,X2,Y2): knight must move from X1,Y1 to X2,Y2.

% Output:
% move(X1,Y1,X2,Y2): knight moves from X1,Y1 to X2,Y2.

number(X) :- size(X).
number(X) :- number(Y), X=Y-1, X>0.

```

```

% There is no tour for a NxN chessboard where N is odd.
even :- size(N), number(X), N = X+X.
:- not even.

% There is no tour for a NxN chessboard where N is lesser than 6.
:- size(N), N < 6.

% Compute the cells of the chessboard.
row_col(X) :- number(X), X >= 1, X <= N, size(N), even.
cell(X,Y) :- row_col(X), row_col(Y).

% Given moves must be done.
move(X1,Y1,X2,Y2) :- givenmove(X1,Y1,X2,Y2).

% Guess the other moves.
{ move(X1,Y1,X2,Y2) } :- valid(X1,Y1,X2,Y2).

% Compute the valid moves from each cell.
valid(X1,Y1,X2,Y2) :- cell(X1,Y1), cell(X2,Y2), X1 = X2+2, Y1 = Y2+1.
valid(X1,Y1,X2,Y2) :- cell(X1,Y1), cell(X2,Y2), X1 = X2+2, Y2 = Y1+1.
valid(X1,Y1,X2,Y2) :- cell(X1,Y1), cell(X2,Y2), X2 = X1+2, Y1 = Y2+1.
valid(X1,Y1,X2,Y2) :- cell(X1,Y1), cell(X2,Y2), X2 = X1+2, Y2 = Y1+1.
valid(X1,Y1,X2,Y2) :- cell(X1,Y1), cell(X2,Y2), X1 = X2+1, Y1 = Y2+2.
valid(X1,Y1,X2,Y2) :- cell(X1,Y1), cell(X2,Y2), X1 = X2+1, Y2 = Y1+2.
valid(X1,Y1,X2,Y2) :- cell(X1,Y1), cell(X2,Y2), X2 = X1+1, Y1 = Y2+2.
valid(X1,Y1,X2,Y2) :- cell(X1,Y1), cell(X2,Y2), X2 = X1+1, Y2 = Y1+2.

% Exactly one move entering to each cell.
:- cell(X,Y), not exactlyOneMoveEntering(X,Y).
exactlyOneMoveEntering(X,Y) :- move(X,Y,X1,Y1),
                             not atLeastTwoMovesEntering(X,Y).
atLeastTwoMovesEntering(X,Y) :- move(X,Y,X1,Y1),
                               move(X,Y,X2,Y2), X1 != X2.
atLeastTwoMovesEntering(X,Y) :- move(X,Y,X1,Y1),
                               move(X,Y,X2,Y2), Y1 != Y2.

% Exactly one move leaving each cell.
:- cell(X,Y), not exactlyOneMoveLeaving(X,Y).
exactlyOneMoveLeaving(X,Y) :- move(X1,Y1,X,Y),
                              not atLeastTwoMovesLeaving(X,Y).
atLeastTwoMovesLeaving(X,Y) :- move(X1,Y1,X,Y),
                               move(X2,Y2,X,Y), X1 != X2.
atLeastTwoMovesLeaving(X,Y) :- move(X1,Y1,X,Y),
                               move(X2,Y2,X,Y), Y1 != Y2.

% Each cell must be reached by the knight.
reached(X,Y) :- move(1,1,X,Y).
reached(X2,Y2) :- reached(X1,Y1), move(X1,Y1,X2,Y2).

```

```
:- cell(X,Y), not reached(X,Y).
```

Inuput generator:

```
% Knight Tour
% s ... maximal size of the board

dom(1..s).
1 { size(X) : dom(X) } 1.
{ givenmove(X1,Y1,X2,Y2) } :- valmv(X1,Y1,X2,Y2).

valmv(X1,Y1,X2,Y2) :- dom(X1;X2;Y1;Y2), X1 = X2+2, Y1 = Y2+1,
    size(N), X1 <= N, X2 <= N, Y1 <= N, Y2 <= N.
valmv(X1,Y1,X2,Y2) :- dom(X1;X2;Y1;Y2), X1 = X2+2, Y2 = Y1+1,
    size(N), X1 <= N, X2 <= N, Y1 <= N, Y2 <= N.
valmv(X1,Y1,X2,Y2) :- dom(X1;X2;Y1;Y2), X2 = X1+2, Y1 = Y2+1,
    size(N), X1 <= N, X2 <= N, Y1 <= N, Y2 <= N.
valmv(X1,Y1,X2,Y2) :- dom(X1;X2;Y1;Y2), X2 = X1+2, Y2 = Y1+1,
    size(N), X1 <= N, X2 <= N, Y1 <= N, Y2 <= N.
valmv(X1,Y1,X2,Y2) :- dom(X1;X2;Y1;Y2), X1 = X2+1, Y1 = Y2+2,
    size(N), X1 <= N, X2 <= N, Y1 <= N, Y2 <= N.
valmv(X1,Y1,X2,Y2) :- dom(X1;X2;Y1;Y2), X1 = X2+1, Y2 = Y1+2,
    size(N), X1 <= N, X2 <= N, Y1 <= N, Y2 <= N.
valmv(X1,Y1,X2,Y2) :- dom(X1;X2;Y1;Y2), X2 = X1+1, Y1 = Y2+2,
    size(N), X1 <= N, X2 <= N, Y1 <= N, Y2 <= N.
valmv(X1,Y1,X2,Y2) :- dom(X1;X2;Y1;Y2), X2 = X1+1, Y2 = Y1+2,
    size(N), X1 <= N, X2 <= N, Y1 <= N, Y2 <= N.

#hide.
#show size/1.      % input
#show givenmove/4. % input
#show move/4.     % output
```

Disjunctive Scheduling

In `DISJUNCTSCHEDULING`, we are given a set of task intervals with fixed durations represented via `task(T,D)`, where each task `T` has a fixed duration `D`. Also, each task `T` has an earliest start time `est/2` and a latest end time `let/2`. Precedence constraints, `prec/2`, state which tasks have to be completed before other tasks, and disjunctive constraints, `disj/2`, state which task intervals cannot overlap. The goal is to assign a start time to each task using predicate `time/2` such that the given constraints are satisfied.

Input signature: `task/2, est/2, let/2, disj/2, prec/2`

Output signature: `time/2`

Preconditions:

- The number of tasks cannot exceed the number of distinct points in time. Each task has precisely one associated duration, earliest starting time, and latest end time, respectively.
- For each task, its latest end time is never before its earliest starting time.
- The transitive closure of the precedence relation is irreflexive.

Encoding:

```

mintime(T) :- est(Fv1,T), not timeLessThan(T).
timeLessThan(T) :- est(Fv1,T), est(Fv2,T1), T1<T.

maxtime(T) :- let(Fv1,T), not timeGreaterThan(T).
timeGreaterThan(T) :- let(Fv1,T), let(Fv2,T1), T1>T.

times(T) :- mintime(T).
times(T) :- times(T1), T=T1+1, T<=MAX, maxtime(MAX).

{ time(I,T) } :- task(I,D), times(T),
                est(I,S), S<=T, End=T+D,
                let(I,E), End<=E.

:- task(I,Fv1), not exactlyOneScheduling(I).
exactlyOneScheduling(I) :- time(I,T), not atLeastTwoScheduling(I).
atLeastTwoScheduling(I) :- time(I,T1), time(I,T2), T1 != T2.

:- prec(I1,I2), task(I1,D), time(I1,T1), time(I2,T2),
   End1 = T1+D, T2 < End1.

:- disj(I1,I2), task(I1,D1), task(I2,D2), time(I1,T1), time(I2,T2),
   End1 = T1+D1, End2=T2+D2, T1<=T2, T2 < End1.

:- disj(I1,I2), task(I1,D1), task(I2,D2), time(I1,T1), time(I2,T2),
   End1 = T1+D1, End2=T2+D2, T2<=T1, T1 < End2.

```

Input generator: We break symmetries by enforcing that earlier tasks have lower IDs.

```

% Disjunctive Scheduling
% s ... max number of distinct time points

dom(1..s).
{ taskID(T) : dom(T) }.
1 { task(T,D) : dom(D) } 1 :- taskID(T).

```

```

1 { est(T,S) : dom(S) } 1 :- taskID(T).
1 { let(T,S) : dom(S) } 1 :- taskID(T).
:- est(T,S1), let(T,S2), S1 > S2.

:- est(T1,S1), est(T2,S2), S1 < S2, T2 < T1. % break symmetries

0 { disj(X,Y) } 1 :- taskID(X), taskID(Y), X != Y.
0 { prec(X,Y) } 1 :- taskID(X), taskID(Y), X != Y.

precCl(X,Y) :- prec(X,Y).
precCl(X,Z) :- precCl(X,Y), precCl(Y,Z).
:- precCl(X,X).

#hide.
#show taskID/1. % EVA for input generator
#show task/2. % input
#show est/2. % input
#show let/2. % input
#show disj/2. % input
#show prec/2. % input
#show time/2. % output

```

Packing Problem

We want to pack N squares on a rectangular area of size $W \times H$ such that no two squares overlap. The number of squares is given by $\text{max_square_num}(N)$, the size of the area by $\text{area}(W, H)$, and the size S of each square I by $\text{square}(I, S)$. Solutions of the **PACKINGPROBLEM** are encoded using $\text{pos}(I, X, Y)$ stating that the top-left corner of square I is at position (X, Y) of the area.

Input signature: $\text{area}/2, \text{max_square_num}/1, \text{square}/2$

Output signature: $\text{pos}/3$

Preconditions:

- Precisely one fact $\text{area}/2$ represents the size of the rectangular area.
- The number of squares is between one and the surface of the rectangular area.
- The size of each square is as least one; each square has to fit on the area.

Encoding:

```

int(X) :- area(X, Fv1).
int(X) :- area(Fv1, X).

```

```

int(X) :- int(Y), X = Y-1, X >= 0.

{ pos(I,X,Y) } :- square(I,D), area(W,H), int(X), int(Y),
                 X >= 0, Y >= 0, X1 = X + D, Y1 = Y + D,
                 W >= X1, H >= Y1.

:- pos(I,X,Y), pos(I,X1,Y1), X1 != X.
:- pos(I,X,Y), pos(I,X1,Y1), Y1 != Y.

pos_square(I) :- pos(I,X,Y).
:- square(I,D), not pos_square(I).

overl(I1,I2) :- pos(I1,X1,Y1), square(I1,D1),
                pos(I2,X2,Y2), square(I2,D2), I1 != I2, W1 = X1+D1,
                H1 = Y1+D1, X2 >= X1, X2 < W1, Y2 >= Y1, Y2 < H1.
overl(I1,I2) :- pos(I1,X1,Y1), square(I1,D1),
                pos(I2,X2,Y2), square(I2,D2), I1 != I2, W1 = X1+D1,
                H1 = Y1+D1, H2 = Y2+D2, X2 >= X1,
                X2 < W1, H2 > Y1, H2 <= H1.

:- overl(I1,I2).

```

Input generator: We break symmetries as we demand that for any two squares with IDs I_1 and I_2 , $I_1 < I_2$ whenever I_1 is smaller than I_2 .

```

% Packing
% s ... max width, resp, height, of the area

dom(1..s).
domSq(1..s*s).
1 { area(X,Y) : dom(X) : dom(Y) } 1.
1 { max_square_num(X) : domSq(X) } 1.

1 { square(S,D) : dom(D) } 1 :- max_square_num(X),
                               1 <= S, S <= X, domSq(S).
:- square(S1,D1), square(S2,D2), S1<S2, D2>D1. % break symmetries

#hide.
#show area/2.           % input
#show max_square_num/1. % input
#show square/2.        % input
#show pos/3.           % output

```

Multi-Context System Querying

Multi-context systems (MCSs) [20] deal with interlinking distributed knowledge bases which we call *contexts*. We consider systems where each context C_i is a normal propositional logic program P_i under the stable-model semantics. We use `ctx` (ContextID) to

introduce identifiers of contexts, `ctxrule(ContextID, CRuleID)` to introduce identifiers of rules in contexts, and `ctxrulehead(CRuleID, Atom)` to describe the head of a rule within a context. Furthermore, `ctxrulebody(CRuleID, Atom, PosNeg)` describes one body atom of a rule within a context, where `PosNeg` is either `pos` or `neg`, `brule(ContextID, BRuleID)` introduces a bridge rule identifier at a certain context, `brulehead(BRuleID, Atom)` describes the head of a bridge rule (there must be exactly one), `brulebody(BRuleID, ContextID, Atom, PosNeg)` describes one body atom of a bridge rule, where `ContextID` is the context where the atom is checked, `Atom` is the atom, and `PosNeg` is again one of `pos` or `neg`.

A query to `in(ContextID, Atom)` returns true iff in all equilibria of the encoded MCS, `Atom` is contained in the stable model of `ContextID`.

Input signature:

`ctx/1, ctxrule/2, ctxrulehead/2, ctxrulebody/3, brule/2, brulehead/2, brulebody/4, query/2`

Output signature: `in/2`

Input Preconditions:

- Identifiers for context modules are consecutive integers starting at 1.
- Each context rule has at most one head.
- Each bridge rule has precisely one head.
- Precisely one fact `query/2` specifies the input query.

Encoding:

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Context Meta Interpreter %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Basic context rule meta interpreter.
ctxruleblock(CRuleID) :- ctxrule(Context, CRuleID),
                        ctxrulebody(CRuleID, Atom, pos),
                        not in(Context, Atom).

ctxruleblock(CRuleID) :- ctxrule(Context, CRuleID),
                        ctxrulebody(CRuleID, Atom, neg),
                        in(Context, Atom).

% Enforce constraints.
ctxrulehashead(CRuleID) :- ctxrulehead(CRuleID, Fv1).
:- ctxrule(Fv1, CRuleID), not ctxruleblock(CRuleID),
   not ctxrulehashead(CRuleID).
  
```

```

% Execute regular rules.
in(Context,Atom) :- ctxrule(Context,CRuleID),
                  ctxrulehead(CRuleID,Atom),
                  not ctxruleblock(CRuleID).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Equilibrium Meta Interpreter %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Guess output projected equilibrium.
1 { eqin(Context,Atom), eqout(Context,Atom) } 1 :-
  brulebody(BRuleID,Context,Atom,Fv1).
% Evaluate bridge rules.
bruleblock(BRuleID) :- brule(Fv1,BRuleID),
                      brulebody(BRuleID,ContextID,Atom,pos),
                      not eqin(ContextID,Atom).
bruleblock(BRuleID) :- brule(Fv1,BRuleID),
                      brulebody(BRuleID,ContextID,Atom,neg),
                      eqin(ContextID,Atom).
in(Context,Atom) :- brule(Context,BRuleID),
                   brulehead(BRuleID,Atom),
                   not bruleblock(BRuleID).

% Enforce that output of context is equal to guess.
:- eqout(Context,Atom), in(Context,Atom).
:- eqin(Context,Atom), not in(Context,Atom).

:- query(C,A), not in(C,A).
  
```

Input generator: Note that to obtain EVA property, it was necessary to declare predicates `maxCtx/1` and `eqin/2` as visible.

```

% Multi-Context System Querying
% r ... max number of context modules
% s ... max number of rules
% t ... max number of atoms

cDom(1..r).
1 { maxCtx(X) : cDom(X) } 1.
ctx(X) :- cDom(X), X <= S, maxCtx(S).

rDom(1..s).
{ ctxrule(C,R) : rDom(R) } :- ctx(C).
cR(R) :- ctxrule(C,R).

aDom(1..t).
{ ctxrulehead(R,A) : aDom(A) } 1 :- cR(R).

pol(pos;neg).
{ ctxrulebody(R,A,P) } :- cR(R), aDom(A), pol(P).
  
```

```

{ brule(C,R) : rDom(R) } :- ctx(C).
bR(R) :- brule(C,R).

1 { brulehead(R,A) : aDom(A) } 1 :- bR(R).

{ brulebody(R,C,A,P) } :- cR(R), ctx(C), aDom(A), pol(P).

{ query(C,A) : aDom(A) } 1 :- ctx(C).
:- not 1 { query(C,A) : cDom(C) : aDom(A) } 1.

#hide.
#show maxCtx/1.          % EVA for input generator
#show eqin/2.           % EVA for encoding
#show eqout/2.          % EVA for encoding
#show ctx/1.            % input
#show ctxrule/2.        % input
#show ctxrulehead/2.   % input
#show ctxrulebody/3.   % input
#show brule/2.          % input
#show brulehead/2.     % input
#show brulebody/4.     % input
#show query/2.         % input
#show in/2.            % output

```

Towers of Hanoi

We consider a variant of the Towers of Hanoi problem with four pegs and n disks. Initially, all n disks are on the left-most peg. The goal of HANOITOWER is to move all n disks to the right-most peg with the help of the two middle pegs. The rules are:

- move one disk at a time,
- only the top disk on a peg can be moved,
- a larger disk cannot be placed on top of a smaller one.

Predicate `steps/1` represents the number of steps in which the goal state is to be reached.

Moreover, $k + 1$ occurrences of predicate `time/1` define the steps of the problem; time starts at 0 and ends at k , and $n + 4$ atoms `disk/1` define the four pegs and n disks: the first four `disk(1)`, `disk(2)`, `disk(3)`, `disk(4)` are the four pegs. Disks $5, 6, \dots, n+4$ are the n disks so that disk i is larger than disk j if $i < j$. The initial placement of disks on the pegs is represented using `on0/2`; `on0(x,y)` specifies that disk x is placed on top of disk y in the initial configuration. The final placement of disks on the pegs is represented using `ongoal/2`; `ongoal(x,y)` specifies that disk x is placed on top of

disk y in the goal configuration. The solution must be encoded by predicate `put/3`, where `put(T, I, J)` stands for “at step T , put disk J on top of disk I ”.

Input signature: `steps/1, time/1, disk/1, on0/2, ongoal/2`

Output signature: `put/3`

Preconditions:

- Precisely one atom `steps/1` represents the number of steps.
- Time points are given as consecutive integers starting at 0.
- Four atoms `disk(1)`, `disk(2)`, `disk(3)`, `disk(4)` are the four pegs; disks `5, 6, ..., n + 4` are the n disks.
- The transitive closure of the `on0/2`, resp., `ongoyal/2`, relation is irreflexive.
- Pegs cannot be placed on disks; each disk has to be above one peg; at most one disk can be placed on top of another disk; a disk can be placed on top of at most one other disk.

Encoding:

```
% Read in data.
on(0,N1,N) :- on0(N,N1).
onG(K,N1,N) :- ongoal(N,N1), steps(K).

% Specify valid arrangements of disks.
% Basic condition. Smaller disks are on larger ones.
:- time(T), on(T,N1,N), N1 >= N.

% Specify a valid move (only for T<t).
% Pick a disk to move.
{ move(T,N) } :- disk(N), time(T), steps(K), T<K.
:- move(T,N1), move(T,N2), N1 != N2.
:- time(T), steps(K), T<K, not diskMoved(T).
diskMoved(T) :- move(T,Fv1).

% Pick a disk onto which to move.
{ where(T,N) } :- disk(N), time(T), steps(K), T<K.
:- where(T,N1), where(T,N2), N1 != N2.
:- time(T), steps(K), T<K, not diskWhere(T).
diskWhere(T) :- where(T,Fv1).

% Pegs cannot be moved.
:- move(T,N), N<5.
```

```

% Only top disk can be moved.
:- on(T,N,N1), move(T,N).

% A disk can be placed on top only.
:- on(T,N,N1), where(T,N).

% No disk is moved in two consecutive moves.
:- move(T,N), move(TM1,N), TM1=T-1.

% Specify effects of a move.
on(TP1,N1,N) :- move(T,N), where(T,N1), TP1=T+1.

on(TP1,N,N1) :- time(T), steps(K), T<K,
                 on(T,N,N1), not move(T,N1), TP1=T+1.

% Goal description.
:- not on(K,N,N1), onG(K,N,N1), steps(K).
:- on(K,N,N1), not onG(K,N,N1), steps(K).

% Solution.
put(T,M,N) :- move(T,N), where(T,M), steps(K), T<K.
  
```

Input generator: Note that to obtain the EVA property for the encoding, it was necessary to declare predicates `where/2` and `move/2` as visible.

```

% Tower of Hanoi
% s ... max number of steps
% t ... max number of disks

domS(1..s).
1 { maxS(X) : domS(X) } 1.
steps(X) :- maxS(X).

time(0).
time(X) :- domS(X), maxS(S), X <= S.

domD(1..t+4).
1 { maxD(X) : domD(X) : X <= t } 1.

disk(X) :- domD(X), maxD(S), X <= S+4.

{ on0(X,Y) } :- disk(X;Y), X != Y.
:- on0(X,Y), base(X).
onCl(X,Y) :- on0(X,Y).
onCl(X,Z) :- onCl(X,Y), onCl(Y,Z).
:- onCl(X,X).
base(1..4).
  
```

```
onBase(X) :- disk(X), onCl(X,Y), base(Y).
:- disk(X), not base(X), not onBase(X).
:- on0(X,Y), on0(X,Z), Y != Z.
:- on0(X,Y), on0(Z,Y), X != Z.

{ ongoal(X,Y) } :- disk(X;Y), X != Y.
:- ongoal(X,Y), base(X).
onGlCl(X,Y) :- ongoal(X,Y).
onGlCl(X,Z) :- onGlCl(X,Y), onGlCl(Y,Z).
:- onGlCl(X,X).
onGlBase(X) :- disk(X), onGlCl(X,Y), base(Y).
:- disk(X), not base(X), not onGlBase(X).
:- ongoal(X,Y), ongoal(X,Z), Y != Z.
:- ongoal(X,Y), ongoal(Z,Y), X != Z.

#hide.
#show maxS/1.      % EVA for input generator
#show maxD/1.      % EVA for input generator
#show where/2.     % EVA for encoding
#show move/2.      % EVA for encoding
#show steps/1.     % input
#show time/1.      % input
#show disk/1.      % input
#show on0/2.       % input
#show ongoal/2.    % input
#show put/3.       % output
```

Graph Colouring

Given a graph consisting of edges and nodes, in GRAPHCOLOURING, we want to find an assignment of the nodes into a set of colours such that no two adjacent nodes have the same colour.

Input signature: node/1, link/2, colour/1

Output signature: chosenColour/2

Preconditions:

- Node names are consecutive; ascending integers starting from 1.
- The link/2 relation is symmetric.

Encoding:

```
% Guess colours.
{ chosenColour(N,C) } :- node(N), colour(C).
```

```

% At least one color per node.
:- node(X), not colored(X).
colored(X) :- chosenColour(X,Fv1).

% Only one color per node.
:- chosenColour(N,C1), chosenColour(N,C2), C1 != C2.

% No two adjacent nodes have the same colour.
:- link(X,Y), X<Y, chosenColour(X,C), chosenColour(Y,C).

```

Input generator:

```

% Graph Colouring
% s ... max number of nodes
% t ... max number of colors

domN(1..s).
domC(1..t).
1 { maxN(X) : domN(X) } 1.
1 { maxC(X) : domC(X) } 1.
node(X) :- domN(X), maxN(M), X <= M.
colour(X) :- domC(X), maxC(M), X <= M.

{ link(X,Y) } :- node(X), node(Y).
link(X,Y) :- link(Y,X).

#hide.
#show maxN/1.           % EVA for input generator
#show maxC/1.           % EVA for input generator
#show node/1.           % input
#show link/2.           % input
#show colour/1.         % input
#show chosenColour/2.   % output

```

Solitaire

Solitaire is a simple planning game that is played on a grid. Each cell on that grid either contains a marble or is empty. Each turn, the player must jump (orthogonally but not diagonally) a peg over an existing peg, the jumped peg is removed. The task in SOLITAIRE is, given an initial board configuration, to find a sequence of the given number of moves. A solution is encoded using `move/4`, where `move(T,D,X,Y)` indicates that to get to time step `T` from time step `T-1`, the peg in position `(X,Y)` is moved in direction `D` (up, down, left, or right).

Input signature: `full/2, empty/2, time/1`

Output signature: move/4

Preconditions:

- Any grid position is either full or empty in the initial configuration.
- Time points are given as a range of consecutive, ascending integers, starting at 1.

Encoding:

```
% The configuration at time 0.
photo(0,1,C,R) :- full(C,R).
photo(0,0,C,R) :- empty(C,R).

inv(0,1).
inv(1,0).

% ALL legal moves at time T+1.
move_all(T1,down,C,R) :- photo(T,1,C,R), R1=R+1,
                          photo(T,1,C,R1), R2=R+2,
                          photo(T,0,C,R2), T1=T+1, time(T1).
move_all(T1,up,C,R) :- photo(T,1,C,R), R1=R-1,
                       photo(T,1,C,R1), R2=R-2,
                       photo(T,0,C,R2), T1=T+1, time(T1).
move_all(T1,left,C,R) :- photo(T,1,C,R), C1=C-1,
                         photo(T,1,C1,R), C2=C-2,
                         photo(T,0,C2,R), T1=T+1, time(T1).
move_all(T1,right,C,R) :- photo(T,1,C,R), C1=C+1,
                          photo(T,1,C1,R), C2=C+2,
                          photo(T,0,C2,R), T1=T+1, time(T1).

% GUESS a set of legal MOVES.
{ move(T,D,C,R) } :- move_all(T,D,C,R).

% ONLY one move for each step (time) is allowed.
:- move(T,D1,Fv1,Fv2), move(T,D2,Fv3,Fv4), D1 != D2.
:- move(T,Fv1,C1,Fv2), move(T,Fv3,C2,Fv4), C1 != C2.
:- move(T,Fv1,Fv2,R1), move(T,Fv3,Fv4,R2), R1 != R2.
moveT(T) :- move(T,Fv1,Fv2,Fv3).
:- time(T), not moveT(T).

% The cells changed after the move down at time T.
changed(T,0,C,R) :- move(T,down,C,R).
changed(T,0,C,R1) :- move(T,down,C,R), R1=R+1.
changed(T,1,C,R2) :- move(T,down,C,R), R2=R+2.

% The cells changed after the move up at time T.
changed(T,0,C,R) :- move(T,up,C,R).
```

```

changed(T,0,C,R1) :- move(T,up,C,R), R1=R-1.
changed(T,1,C,R2) :- move(T,up,C,R), R2=R-2.

% The cells changed after the move left at time T.
changed(T,0,C,R) :- move(T,left,C,R).
changed(T,0,C1,R) :- move(T,left,C,R), C1=C-1.
changed(T,1,C2,R) :- move(T,left,C,R), C2=C-2.

% The cells changed after the move right at time T.
changed(T,0,C,R) :- move(T,right,C,R).
changed(T,0,C1,R) :- move(T,right,C,R), C1=C+1.
changed(T,1,C2,R) :- move(T,right,C,R), C2=C+2.

% The configuration at time T (after the ONLY move at time T).
photo(T1,S,C,R) :- photo(T,S,C,R), T1=T+1, time(T1),
                    inv(S,SI), not changed(T1,SI,C,R).
photo(T,S,C,R) :- changed(T,S,C,R).

```

Input generator: While the size of the grid is fixed for the ASP competition, we allow it to vary and thus to scale down the size of test inputs.

```

% Solitaire
% s ... size of the grid
% t ... number of moves

dom(1..s).
{ grid(X,Y) : dom(X) : dom(Y) }.

{ full(X,Y), empty(X,Y) } 1 :- grid(X,Y).

domT(1..t).
1 { maxT(X) : dom(X) } 1.
time(X) :- domT(X), X <= M, maxT(M).

#hide.
#show grid/2. % EVA for input generator
#show maxT/1. % EVA for input generator
#show full/2. % input
#show empty/2. % input
#show time/1. % input
#show move/4. % output

```

Weight-Assignment Tree

The weight-assignment tree problem WEIGHTASGTREE is concerned with a full-binary tree with n leaves such that its leaves are pairs (*weight*, *cardinality*), where *weight* and *cardinality* are integers, the right child of an inner node is a leaf, each inner node is a pair (*colour*, *weight*), where *colour* is either *green*, *red*, or *blue*, and there are inner nodes

$1, \dots, n - 1$ such that node $n - 1$ is a root node and inner node $i - 1$ is the left child of inner node i for $i = 2, \dots, n - 1$.

The weights of inner nodes are subject to constraints based on their colours, weights and the cardinalities of their children. Specifically, the weight of an inner node X satisfies the constraint:

- If the colour of X is green, the weight of X is the weight of the right child of X plus the cardinality of the right child of X .
- If the colour of X is red, the weight of X is the weight of the right child of X plus the weight of the left child of X .
- If the colour of X is blue, the weight of X is the cardinality of the right child of X plus the weight of the left child of X .

The total weight of a tree is the sum of the weights of its inner nodes.

The problem of `WEIGHTASGTREE` is: Given weights and cardinality values of n leaves and an integer mv , verify if there is a tree T with these n leaves satisfying the above conditions whose total weight is less or equal to mv .

Predicate `leafWeightCardinality(leaf, w, c)` specifies the weights and cardinality values for each leaf node. The number of leaves in the tree is given by `num(n)`, and integer mv is given by `max_total_weight(mv)`. Inner nodes are represented via `innerNode/1`. A solution is represented by means of an atom `exists/0` if there is a tree T and a specification of T using predicates `innerNodeColor(node, color)` and `innerLeftRight(node, l_leaf, r_leaf)`.

Input signature: `leafWeightCardinality/3, num/1, max_total_weight/1, innerNode/1`

Output signature: `exists/0, leafWeightCardinality/3, innerNodeColor/2, innerLeftRight/3, num/1, max_total_weight/1`

Preconditions:

- Precisely one fact `num/1` represents the number of leaf nodes.
- Inner nodes are represented via consecutive integers starting at 1.
- Precisely one weight-cardinality pair is given for each node.
- Precisely one fact `max_total_weight/1` specifies the total weight mv .

Encoding:

```
nWeight(X,W) :-leafWeightCardinality(X,W,C).
nCard(X,C) :-leafWeightCardinality(X,W,C).
leaf(X):-leafWeightCardinality(X,W,C).

%% Tree T' Generation and Test
%% GENERATE tree
{ childRight(R,P) } :- innerNode(P), leaf(R).
:- childRight(R,P), childRight(R1,P), R1 != R.
hasChildRight(P) :- childRight(R,P).
:- innerNode(P), not hasChildRight(P).

innerLeftRight(P,P1,R) :- innerNode(P), P1 = P - 1, P>1,
                           childRight(R,P).

% Inner node 1 has two leafs.
{ innerLeftRight(1,L,R) } :- leaf(L), childRight(R,1).

%% TEST tree
% Same leaf cannot be a leaf of a different parent.
:-childRight(R,P1), childRight(R,P2), P1 != P2.

% Someones right child.
child(N):-childRight(N,P).
% Left child of the first inner node may not be right child of
% someone else.
:-innerLeftRight(1,L,R), child(L).

% If color of X is green
% weight(X) = weight(right child of X)+cardinality(right child of X).
weight(green,P,W):- innerNodeColor(P,green), W= W1+C,
                    nWeight(R,W1), nCard(R,C), max_total_weight(B), W<2*B,
                    childRight(R,P), leaf(R).

% If color of X is red
% weight(X) = weight(right child of X) + weight(left child of X).
weight(red,P,W):- innerNodeColor(P,red), W= W1+W2,
                  max_total_weight(B),
                  W < 2*B, nWeight(R,W1),nWeight(L,W2),
                  innerLeftRight(P,L,R), leaf(R).

% If color(X) is blue
% weight(X) = cardinality(right child of X)+weight(left child of X).
weight(blue, P,W):- innerNodeColor(P,blue), W= W1+C,
                   max_total_weight(B), W < 2*B,
                   nCard(R,C), nWeight(L,W1),
                   innerLeftRight(P,L,R), leaf(R).
```

```

%% Each inner node in primer tree T' has a unique color.
1 { innerNodeColor(P,red), innerNodeColor(P,green),
    innerNodeColor(P,blue) } 1 :- innerNode(P).

nWeight(P,W):- weight(C,P,W).

%% Definition of a total weight of a prime tree T'.
tWeight(1,W):- nWeight(1,W).
tWeight(N,W):- W=W1+W2, N1 = N-1, tWeight(N1,W1),
    nWeight(N,W2), innerNode(N),N>1.

% Exists Definition.
exists:- tWeight(N1,W), N1 = N-1, W<=M,
    max_total_weight(M),num(N).
:- not exists.

```

Input generator: To break symmetries, node IDs are ordered according to their weights.

```

% Weight-Assignment Tree
% s ... max number of leaf nodes
% t ... domain of weights and cardinalities

domN(1..s).
1 { num(X) : domN(X) } 1.

innerNode(X) :- domN(X), X < N, num(N).

domWC(1..t).
1 { leafWeightCardinality(L,W,C) : domWC(W) : domWC(C) } 1 :-
    domN(L), L <= N, num(N).

% Break some symmetries.
:- leafWeightCardinality(L1,W1,C1),
    leafWeightCardinality(L2,W2,C2), W1 < W2, L2 < L1.

1 { max_total_weight(X) : X = 1..s } 1.

#hide.
#show childRight/2.           % EVA for encoding
#show leafWeightCardinality/3. % input
#show max_total_weight/1.     % input
#show innerNode/1.           % input
#show num/1.                 % input and output
#show exists/0.              % output
#show leafWeightCardinality/3. % output
#show innerNodeColor/2.      % output
#show innerLeftRight/3.      % output
#show max_total_weight/1.     % output

```

Problems in NP with a Non-Tight Encoding

Sokoban

A Sokoban puzzle consists of a room layout (walls and empty spaces on a grid) and a starting situation (one warehouse keeper and a number of boxes, all of which must reside on some floor location, where one box occupies precisely one location and each location can hold at most one box). The goal is to move all boxes onto dedicated storage locations. To this end, the warehouse keeper can walk on floor locations (unless occupied by some box), and push single boxes onto unoccupied floor locations.

We consider walking to a box and pushing it a certain number of locations in one direction as an atomic action. In the decision version `SOKOBANDECISION` of this problem, the question to answer is whether a solution involving exactly n walk-and-push actions exists. If so, a witness containing the sequence of actions should be produced; otherwise no solution should be output.

An initial room layout is encoded as follows: `right(l1,l2)` means location $l2$ is immediately right of location $l1$, `top(l1,l2)` means location $l2$ is immediately on top of location $l1$, `solution(l)` means location l is a storage location, `box(l)` means l initially holds a box, and `sokoban(l)` encodes that the warehouse keeper is at location l . Time steps are represented via `steps/1` and `next/2` encodes a successor relation between time steps.

Input signature: `right/2, top/2, solution/1, box/1, sokoban/1, step/1, next/2`

Output signature: `push/4`

Preconditions:

- The transitive closure of the `right/2`, resp., `top/2`, relation is irreflexive.
- Precisely one atom `sokoban/1` represents the initial position of the warehouse keeper.
- Time steps are represented as consecutive integers starting at 1.
- Time `next/2` relation between time steps is the usual successor relation between integers.

Encoding:

```
% actionstep: can push at any step but the final one.  
actionstep(S) :- next(S, S1).
```

```

% Utility predicates: left and bottom.
left(L1,L2) :- right(L2,L1).
bottom(L1,L2) :- top(L2,L1).

% Utility predicates: adjacent.
adj(L1,L2) :- right(L1,L2).
adj(L1,L2) :- left(L1,L2).
adj(L1,L2) :- top(L1,L2).
adj(L1,L2) :- bottom(L1,L2).

% Identify locations.
location(L) :- adj(L,Fv1).

% Initial configuration.
box_step(B,F) :- box(B), initial_step(F).
sokoban_step(S,F) :- sokoban(S), initial_step(F).

% push(B,D,B1,S) :
% At actionstep S push box at location B in
% direction D (right, left, up, down) until location B1.
% The sokoban must be able to get to the location "before" B in
% order to push the box, also there should not be any boxes between
% B and B1 (and also not on B1 itself at step S.
{ push(B,right,B1,S) } :- reachable(L,S), right(L,B), box_step(B,S),
    pushable_right(B,B1,S),
    good_pushlocation(B1),
    actionstep(S).
{ push(B,left,B1,S) } :- reachable(L,S), left(L,B), box_step(B,S),
    pushable_left(B,B1,S),
    good_pushlocation(B1),
    actionstep(S).
{ push(B,up,B1,S) } :- reachable(L,S), top(L,B), box_step(B,S),
    pushable_top(B,B1,S),
    good_pushlocation(B1),
    actionstep(S).
{ push(B,down,B1,S) } :- reachable(L,S), bottom(L,B), box_step(B,S),
    pushable_bottom(B,B1,S),
    good_pushlocation(B1), actionstep(S).

% reachable(L,S) :
% Identifies locations L which are reachable by the sokoban at step S.
reachable(L,S) :- sokoban_step(L,S).
reachable(L,S) :- reachable(L1,S), adj(L1,L), not box_step(L,S).

% pushable_right(B,D,S) :
% Box at B can be pushed right until D at step S.
% Analogous for left, top, bottom.
pushable_right(B,D,S) :- box_step(B,S), right(B,D),
    not box_step(D,S), actionstep(S).

```

```

pushable_right(B,D,S) :- pushable_right(B,D1,S),
                          right(D1,D), not box_step(D,S).
pushable_left(B,D,S) :- box_step(B,S), left(B,D),
                          not box_step(D,S), actionstep(S).
pushable_left(B,D,S) :- pushable_left(B,D1,S),
                          left(D1,D), not box_step(D,S).
pushable_top(B,D,S) :- box_step(B,S), top(B,D),
                        not box_step(D,S), actionstep(S).
pushable_top(B,D,S) :- pushable_top(B,D1,S),
                        top(D1,D), not box_step(D,S).
pushable_bottom(B,D,S) :- box_step(B,S), bottom(B,D),
                           not box_step(D,S), actionstep(S).
pushable_bottom(B,D,S) :- pushable_bottom(B,D1,S),
                           bottom(D1,D), not box_step(D,S).

% The sokoban is at a new location after a push and no
% longer at its original position.
sokoban_step(L,S1) :- push(Fv1,right,B1,S), next(S,S1), right(L,B1).
sokoban_step(L,S1) :- push(Fv1,left,B1,S), next(S,S1), left(L,B1).
sokoban_step(L,S1) :- push(Fv1,up,B1,S), next(S,S1), top(L,B1).
sokoban_step(L,S1) :- push(Fv1,down,B1,S), next(S,S1), bottom(L,B1).
-sokoban_step(L,S1) :- push(B,Fv1,B1,S), next(S,S1),
                       sokoban_step(L,S), B != B1.

% Also the box_step has moved after having been pushed.
box_step(B,S1) :- push(Fv1,Fv2,B,S), next(S,S1).
-box_step(B,S1) :- push(B,Fv1,B1,S), next(S,S1), B != B1.

% Inertia: Boxes and the sokoban usually remain where they are.
box_step(LB,S1) :- box_step(LB,S), next(S,S1), not -box_step(LB,S1).
sokoban_step(LS,S) :- sokoban_step(LS,S), next(S,S1),
                       not -sokoban_step(LS,S1).

% Don't push two different boxes in one step.
:- push(B,Fv1,Fv2,S), push(B1,Fv3,Fv4,S), B != B1.
% Don't push a box in different directions in one step.
:- push(B,D,Fv1,S), push(B,D1,Fv2,S), D != D1.
% Don't push a box onto different locations in one step.
:- push(B,D,B1,S), push(B,D,B11,S), B1 != B11.

% Avoid pushing boxes into dead ends. There should be a
% location to the left and right or to top and bottom. Otherwise
% the box cannot be taken out again, for instance from corners.
% Obviously if the location is a target, these restrictions do not
% apply, as the box may remain there forever.
good_pushlocation(L) :- right(L,Fv1), left(L,Fv2).
good_pushlocation(L) :- top(L,Fv1), bottom(L,Fv2).
good_pushlocation(L) :- solution(L).

```

```

final_step(S) :- step(S), not no_final_step(S).
no_final_step(S) :- next(S,S1).

initial_step(S) :- step(S), not no_initial_step(S).
no_initial_step(S) :- next(S1,S).

push_happens(S) :- push(Fv1,Fv2,Fv3,S).
push_missing :- actionstep(S), not push_happens(S).
:- push_missing.
solution_notfound_step(S) :- solution(L), step(S), not box_step(L,S).
solution_notfound :- solution_notfound_step(S), final_step(S).
:- solution_notfound.

reachablespecialbox(right,B,S) :- reachable(L,S), right(L,B),
    box_step(B,S), actionstep(S),
    not solution_notfound_step(S).
reachablespecialbox(left,B,S) :- reachable(L,S), left(L,B),
    box_step(B,S), actionstep(S),
    not solution_notfound_step(S).
reachablespecialbox(up,B,S) :- reachable(L,S), top(L,B),
    box_step(B,S), actionstep(S),
    not solution_notfound_step(S).
reachablespecialbox(down,B,S) :- reachable(L,S), bottom(L,B),
    box_step(B,S), actionstep(S),
    not solution_notfound_step(S).

nsmallestspecial(D,B) :- reachablespecialbox(D,B,S),
    reachablespecialbox(D1,B1,S), B>B1.
nsmallestspecial(D,B) :- reachablespecialbox(D,B,S),
    reachablespecialbox(D1,B,S), D>D1.
smallestspecial(D,B) :- reachablespecialbox(D,B,S),
    not nsmallestspecial(D,B).

push(B,D,B,S) :- reachablespecialbox(D,B,S), smallestspecial(D,B).

```

Input generator: To obtain EVA property for the encoding, sokoban_step/2 is declared as visible.

```

% Sokoban Decision
% s ... number of locations
% t ... number of steps

loc(1..s).
stp(1..t).

% right(l1,l2): location l2 is immediately right of location l1
% top(l1,l2): location l2 is immediately on top of location l1
% solution(l): location l is a storage location

```

```

{ right(L1,L2) : loc(L1) : loc(L2) : L1 < L2 }.
{ top(L1,L2) : loc(L1) : loc(L2) : L1 < L2 }.
{ solution(L) : loc(L) }.

% An instance also consists of a description of the initial
% configuration encoded as facts using predicates box/1 and sokoban/1:
% box(l): location l initially holds a box
% sokoban(l): the sokoban is at location l

{ box(L) : loc(L) }.

% Each instance has exactly one fact for sokoban/1.

1 { sokoban(L) : loc(L) } 1.

% An instance also contains a sequence of time-steps for warehouse
% configurations, between which the actions occur and their
% successorship relation, using predicates step/1 and next/2:
% step(s): s is a step
% next(s1,s2): step s2 is the successor of step s1

1 { maxStp(N) : stp(N) } 1.
step(S) :- stp(S), 1 <= S, S <= N, maxStp(N).
next(S1,S2) :- step(S1), step(S2), S2=S1+1.

#hide.
#show maxStp/1.           % EVA for input generator
#show sokoban_step/2.    % EVA for encoding
#show right/2.           % input
#show top/2.             % input
#show solution/1.       % input
#show box/1.            % input
#show sokoban/1.       % input
#show step/1.           % input
#show next/2.           % input
#show push/4.           % output

```

Labyrinth

We deal with a variation of the Labyrinth game by Ravensburger where we need to guide an avatar through a dynamically changing labyrinth to certain fields. The labyrinth consists of fields that can be connected in the four cardinal directions. The shape of the labyrinth changes over time as its rows and columns can be pushed either horizontally or vertically, which involves moving fields out of and into the board. To get to another field, in each turn, the avatar can follow paths that start from the avatar's location and traverse pairwise connected fields. If a field of the labyrinth is pushed out of the board (possibly hosting the avatar), it immediately returns into the board (possibly with the avatar still

residing on it) on the other end of the pushed row or column, respectively. After a push, the avatar can move to any field reachable via a path in which every (non-terminal) field has a connection to its successor, and vice versa. There is a unique starting and a unique goal field in the labyrinth. A solution of LABYRINTH is represented by pushes of the labyrinth's rows and columns such that the avatar can from its starting field (which changes its location when pushed) reach the goal field (which also changes its location when pushed) by a move along some path after each push. There is a maximum number of pushes, and the avatar must be able to reach the goal field (which can be relocated by pushes) by moving along some path after each push.

The fields of the quadratic labyrinth are represented via `field/2`. Predicate `init_on/2`, resp., `goal_on/2`, marks the starting field, resp., goal field, for the avatar. Connections between fields are represented via `connect(x, y, d)` stating that the field at position (x, y) is connected to its neighbour in the north, south, west, or east represented via `d` being `n`, `s`, `w` or, `e`. The maximal number of steps is encoded using `max_steps/1`. A solution is encoded using `push/3`, atom `push(z, d, s)` represents that in step `s` row `z`, resp., column `z`, of the labyrinth is pushed in direction `d`.

Input signature: `field/2, init_on/2, goal_on/2, connect/3, max_steps/1`

Output signature: `push/3`

Preconditions:

- Field positions are given as pairs of consecutive integers starting at 1.
- Precisely one fact `init_on/2`, resp., `goal_on/2`, specifies the avatar's initial, resp., goal, position in the labyrinth.
- Precisely one fact `max_steps/1` specifies the maximal number of steps.

Encoding:

```
dir(e). dir(w). dir(n). dir(s).
inverse(e,w). inverse(w,e).
inverse(n,s). inverse(s,n).

row(X) :- field(X,Y).
col(Y) :- field(X,Y).

num_rows(X) :- row(X), not row(XX), XX = X+1.
num_cols(Y) :- col(Y), not col(YY), YY = Y+1.

goal(X,Y,0) :- goal_on(X,Y).
reach(X,Y,0) :- init_on(X,Y).
```

```

conn(X,Y,D,0) :- connect(X,Y,D).

step(S) :- max_steps(S),      0 < S.
step(T) :- step(S), T = S-1, 1 < S.

%% Direct neighbors.
dneighbor(n,X,Y,XX,Y) :- field(X,Y), field(XX,Y), XX = X+1.
dneighbor(s,X,Y,XX,Y) :- field(X,Y), field(XX,Y), XX = X-1.
dneighbor(e,X,Y,X,YY) :- field(X,Y), field(X,YY), YY = Y+1.
dneighbor(w,X,Y,X,YY) :- field(X,Y), field(X,YY), YY = Y-1.

%% All neighboring fields.
neighbor(D,X,Y,XX,YY) :- dneighbor(D,X,Y,XX,YY).
neighbor(n,X,Y, 1, Y) :- field(X,Y), num_rows(X).
neighbor(s,1,Y, X, Y) :- field(X,Y), num_rows(X).
neighbor(e,X,Y, X, 1) :- field(X,Y), num_cols(Y).
neighbor(w,X,1, X, Y) :- field(X,Y), num_cols(Y).

%% Select a row or column to push.
neg_goal(T) :- goal(X,Y,T), not reach(X,Y,T).
rrpush(T)    :- step(T), neg_goal(S), S = T-1, not ccpush(T).
ccpush(T)    :- step(T), neg_goal(S), S = T-1, not rrpsh(T).
orpush(X,T)  :- row(X), row(XX), rpush(XX,T), X != XX.
ocpush(Y,T)  :- col(Y), col(YY), cpush(YY,T), Y != YY.
rpush(X,T)   :- row(X), rrpsh(T), not orpush(X,T).
cpush(Y,T)   :- col(Y), ccpush(T), not ocpush(Y,T).
push(X,e,T)  :- rpush(X,T), not push(X,w,T).
push(X,w,T)  :- rpush(X,T), not push(X,e,T).
push(Y,n,T)  :- cpush(Y,T), not push(Y,s,T).
push(Y,s,T)  :- cpush(Y,T), not push(Y,n,T).

%% Determine new position of a (pushed) field.
shift(XX,YY,X,Y,T) :- neighbor(e,XX,YY,X,Y), push(XX,e,T), step(T).
shift(XX,YY,X,Y,T) :- neighbor(w,XX,YY,X,Y), push(XX,w,T), step(T).
shift(XX,YY,X,Y,T) :- neighbor(n,XX,YY,X,Y), push(YY,n,T), step(T).
shift(XX,YY,X,Y,T) :- neighbor(s,XX,YY,X,Y), push(YY,s,T), step(T).
shift(X, Y,X,Y,T) :- field(X,Y), not push(X,e,T), not push(X,w,T),
                    not push(Y,n,T), not push(Y,s,T), step(T).

%% Move connections around.
conn(X,Y,D,T) :- conn(XX,YY,D,S), S = T-1, dir(D),
                  shift(XX,YY,X,Y,T), step(T).

%% Location of goal after pushing.
goal(X,Y,T) :- goal(XX,YY,S), S = T-1, shift(XX,YY,X,Y,T), step(T).

%% Locations reachable from new position.
reach(X,Y,T) :- reach(XX,YY,S), S = T-1, shift(XX,YY,X,Y,T), step(T).
reach(X,Y,T) :- reach(XX,YY,T), dneighbor(D,XX,YY,X,Y),

```

```
conn(XX,YY,D,T), conn(X,Y,E,T), inverse(D,E), step(T).
```

```
%% Goal must be reached.
:- neg_goal(S), max_steps(S).
```

Input generator: Note that a couple of predicates of the encoding are declared as visible to obtain EVA property. This involves in particular `rrpush/1`, `ccpush/1`, `orpush/2`, and `ocpush/2`.

```
% Labyrinth
% r ... max size of the grid
% t ... max number of pushes

dom(1..r).
1 { sz(X) : dom(X) } 1.

field(X,Y) :- 1 <= X, 1 <= Y, X <= S, Y <= S, dom(X), dom(Y), sz(S).

1 {init_on(X,Y) : dom(X) : dom(Y) } 1.
:- init_on(X,Y), sz(S), X > S.

:- init_on(X,Y), sz(S), Y > S.
1 {goal_on(X,Y) : dom(X) : dom(Y) } 1.
:- goal_on(X,Y), sz(S), X > S.
:- goal_on(X,Y), sz(S), Y > S.

dd(n;s;e;w).
{ connect(X,Y,D) : dd(D) } :- dom(X), dom(Y), X <= S, Y <= S, sz(S).
1 { max_steps(S) : S = 1..t } 1.

#hide.
#show sz/1.           % EVA for input generator
#show rrpsh/1.       % EVA for encoding
#show ccpsh/1.       % EVA for encoding
#show orpsh/2.       % EVA for encoding
#show ocpsh/2.       % EVA for encoding
#show field/2.       % input
#show init_on/2.     % input
#show goal_on/2.     % input
#show connect/3.     % input
#show max_steps/1.   % input
#show push/3.        % output
```

Numberlink Puzzle

In NUMBERLINK, we are given an undirected graph and a list of connections. Predicate `node/1` defines the nodes of the graph, and `edge/3` the edges, `edge(e,p,q)` means that there is an edge with name `e` between node `p` and node `q`. A connec-

tion consists of two terminal nodes t_1 and t_2 and an index number i represented as $\text{connection}(i, t_1, t_2)$. The goal of the puzzle is to find a list of paths connecting two terminal nodes for each connection such that no node is included in more than two paths. A path is represented using $\text{link}/1$ which encodes a list of involved edge names.

Input signature: $\text{node}/1, \text{edge}/3, \text{connection}/3$

Output signature: $\text{link}/1$

Preconditions:

- Nodes are represented as consecutive integers starting at 1.
- For each edge name, there is at most one fact $\text{edge}/3$.
- Index numbers for connections are consecutive integers starting at 1.
- For each index number, there is precisely one fact $\text{connection}/3$.

Encoding:

```
% Normalize graph.
uedge(Z1,X1,Y1) :- edge(Z1,Y1,X1).
uedge(Z1,X1,Y1) :- edge(Z1,X1,Y1).

% GUESS
% Guesses all possible linked edges for a connection N.
{ linked(Z,N) } :- connection(N,S,A), uedge(Z,X1,X2).

% Builds the path of the linked edges.
path(N,X1,Y1) :- connection(N,X1,Fv1), uedge(Z1,X1,Y1), linked(Z1,N).
path(N,X1,Y2) :- path(N,X1,Y1), uedge(Z1, Y1, Y2), Y1 != Y2,
                linked(Z1,N), X1 != Y2.

% CHECK
% For each connection must exists a path.
:- connection(N,X1,X2), not path(N,X1,X2).

% Each uedge can linked for exactly one connection.
:- linked(Z,N), linked(Z,M), N != M.

% Each node must belong to exactly one connection.
:- linked(Z1,N),linked(Z2,M), N != M, Z1 != Z2, uedge(Z1,X1,Y1),
    uedge(Z2,X1,Y2).

% A uedge can belong to exactly one connection.
```

```

:- path(N,X1,Y1), path(M,X1,Y2), Y1 != Y2, N != M.
:- path(N,X1,Y1), path(M,X2,Y1), X1 != X2, N != M.

% Each node can have exactly two outgoing edges.
:- linked(Z1,N), linked(Z2,N), linked(Z3,N),
   Z1 != Z2, Z2 != Z3, Z3 != Z1,
   uedge(Z1,X1,Y1), uedge(Z2,X1,Y2), uedge(Z3,X1,Y2).

% Each linked uedge must belong to exactly one path.
pippo :- linked(Z,N), uedge(Z,X,Y), not path(N,X1,X),
        connection(N,X1,Fv1), X1 != Y, X1 != X.

% OUTPUT
link(Z) :- linked(Z,Fv1).
  
```

Input Generator: To obtain EVA for the encoding, `linked/2` is declared as visible.

```

% Numberlink Puzzles
% s ... max number of nodes
% t ... max number of connections

nDom(1..s).
1 { maxN(X) : nDom(X) } 1.
node(X) :- nDom(X), X <= S, maxN(S).
{ edge(e(X,Y),X,Y) } :- node(X), node(Y), X != Y.

cDom(1..t).
1 { maxC(X) : cDom(X) } 1.
{ connection(I,X,Y) } :- cDom(I), I <= S, maxC(S),
                        node(X), node(Y), X != Y.

:- maxC(S), cDom(N), 1 <= N, N <= S,
   not 1 { connection(N,X,Y) : nDom(X) : nDom(Y) } 1.

#hide.
#show maxN/1.           % EVA for input generator
#show maxC/1.           % EVA for input generator
#show linked/2.         % EVA for encoding
#show node/1.           % input
#show edge/3.           % input
#show connection/3.     % input
#show link/1.           % output
  
```

Magic-Square Sets

We consider different types of `MAGIC SQUARE SETS`. For each, the size n of the magic square is given using predicate `size(n)`. Furthermore, contents of some cells are given via `sqr/3`. The objective is to determine the content of all remaining cells such that the

constraints of the particular magic-square problem are satisfied. The following types of problems are considered:

- Normal magic squares: a magic square of order n is an arrangement of n^2 numbers $1, \dots, n^2$ in a square such that the n numbers in all rows, all columns, and both diagonals sum to the same constant $n \cdot (n^2 + 1)/2$.
- Semimagic squares: it is a relaxed normal magic square in the sense that only the rows and columns but not necessarily the diagonals sum to the magic constant $n \cdot (n^2 + 1)/2$.
- Panmagic squares: it is a magic square with the additional property that the broken diagonals, i.e., the diagonals that wrap round at the edges of the square, also add up to the magic constant $n \cdot (n^2 + 1)/2$.
- Associative magic squares: it is a magic square for which every pair of numbers symmetrically opposite to the center sum up to the same value $n^2 + 1$.

The type of the problem is specified via `type(N)` where `N` is either `normal`, `semi`, `assoc`, or `pan`.

Input signature: `size/1, type/1, sqr/3`

Output signature: `sqr/3`

Preconditions:

- Precisely one atom `size(N)` specifies the size of the magic square.
- At least one atom `type(N)` specifies the type of the problem; `N` is either `normal`, `semi`, `assoc`, or `pan`. Type `normal` cannot be specified without type `semi`; `pan` cannot be specified without type `normal`; type `assoc` cannot be specified without type `normal`; and type `pan` cannot be specified together with type `assoc`.

Encoding:

```

assSum(S) :- size(N), Sq = N*N, S = Sq+1.
sum(S) :- assSum(As), size(N), X = N*As, S = X/2.

data(1).
data(D) :- data(D1), D = D1+1, size(N), Sq = N*N, D <= Sq.

num(X) :- data(X), size(N), X <= N.

{ sqr(I,J,A) } :- num(I), num(J), data(A).
  
```

```

:- sqr(I,J,A), sqr(I,J,B), A != B.
sqrHelper(I, A) :- sqr(I,J,A).
:- 0 { sqrHelper(I,A) } 0, data(A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% semi magic
% rows
:- type(semi), num(I), sum(S), not S [ sqr(I,J,A) = A ] S.
% columns
:- type(semi), num(J), sum(S), not S [ sqr(I,J,A) = A ] S.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% normal magic
% main diagonal
:- type(normal), sum(S), not S [ sqr(I,I,A) = A ] S.
% secondary diagonal
:- type(normal), sum(S), not S [ sqrHelper2(I,J,A) = A ] S.
sqrHelper2(I,J,A) :- num(I), size(N), D = N-I, J = D+1, sqr(I,J,A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% pan magic
% pan diagonal up
:- sum(S), panDiaUp(_,_,O), not S [ sqrHelper3(R,C,A) = A ] S.
sqrHelper3(R,C,A) :- panDiaUp(R,C,O), sqr(R,C,A).
% pan diagonal down
:- sum(S), panDiaDown(_,_,O), not S [ sqrHelper4sqr(R,C,A) = A ] S.
sqrHelper3(R,C,A) :- panDiaDown(R,C,O), sqr(R,C,A) .

panDiaUp(X,2,1) :- type(pan), size(X).
panDiaUp(X,Y,O) :- panDiaUp(X,Z,Op), O = Op+1,
                  Y = Z+1, Y <= X, size(X).
panDiaUp(X,Y,O) :- panDiaUp(Z,W,O), X = Z-1, X > 0, next(W,Y).

panDiaDown(1,2,1) :- type(pan).
panDiaDown(1,Y,O) :- panDiaDown(1,Z,Op), O = Op+1, O <= 2*X,
                  Y = Z+1, Y <= X, size(X).
panDiaDown(X,Y,O) :- panDiaDown(Z,W,O), next(Z,X), next(W,Y).

next(X,Y) :- num(X), Y = X+1, size(N), Y <= N.
next(X,1) :- size(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% associative magic
even :- type(assoc), size(N), num(X), N = X*2.
odd :- type(assoc), not even.
center(C) :- type(assoc), size(N), T = N+1, C = T/2.
% odd case
associatedOdd(X,Y) :- odd, center(C), X = C-1, Y= C+1.
associatedOdd(X,Y) :- odd, associatedOdd(X1,Y1),
                  X = X1-1, Y = Y1+1, X > 0, size(N),
                  Y <= N.

```

```

% row
:- assSum(S), center(C), associatedOdd(X,Y), sqr(C,X,N1),
   sqr(C,Y,N2), T = N1 + N2, S != T.
% column
:- assSum(S), center(C), associatedOdd(X,Y), sqr(X,C,N1),
   sqr(Y,C,N2), T = N1 + N2, S != T.
% main diagonal
:- assSum(S), associatedOdd(X,Y), sqr(X,X,N1),
   sqr(Y,Y,N2), T = N1 + N2, S != T.
% secondary diagonal
:- assSum(S), associatedOdd(X,Y),
   sqr(X,Y,N1), sqr(Y,X,N2), T = N1 + N2, S != T.
% even case
associatedEven(C,Y) :- even, center(C), Y= C+1.
associatedEven(X,Y) :- even, associatedEven(X1,Y1),
   X = X1-1, Y = Y1+1, X > 0, size(N), Y <= N.
% main diagonal
:- assSum(S), associatedEven(X,Y), sqr(X,X,N1),
   sqr(Y,Y,N2), T = N1 + N2, S != T.
% secondary diagonal
:- assSum(S), associatedEven(X,Y), sqr(X,Y,N1),
   sqr(Y,X,N2), T = N1 + N2, S != T.

```

Input generator: As `sqr/3` is both input predicate and defined in the encoding, a renamed version `inpSqr/3` is introduced in the input generator.

```

% Magic-Square Sets
% s ... max size of the grid

dom(1..s).
1 { size(X) : dom(X) } 1.

types(semi;normal;pan;assoc).
1 { type(X) : types(X) }.
:- type(normal), not type(semi).
:- type(pan), not type(normal).
:- type(assoc), not type(normal).
:- type(pan), type(assoc).

int(1..s*s).
val(X) :- int(X), 1 <= X, X <= S*S, size(S).
{ inpSqr(I,J,V) : int(V) } 1 :- dom(I), dom(J),
   I <= S, J <= S, size(S).
:- inpSqr(I,J,V), size(S), S < V.
sqr(I,J,V) :- inpSqr(I,J,V).

#hide.
#show inpSqr/3.    % renamed input / EVA for input generator
#show size/1.     % input

```

```
#show type/1.          % input
#show sqr/3.          % input and output
```

Maze Generation

The problem MAZEGENERATION is about generating a two-dimensional maze grid where each cell either contains a wall or is empty. The grid has to satisfy certain conditions, in particular:

- Each cell is empty or a wall.
- Each cell in an edge of the grid is a wall, except entrance and exit that are empty.
- There is no 2×2 square of empty cells or walls.
- If two walls are on a diagonal of a 2×2 square, then not both of their common neighbours are empty.
- No wall is completely surrounded by empty cells.
- There is a path from the entrance to every empty cell.

The input of this problem consists of facts representing the position of the entrance, entrance (X, Y) , and the exit, exit (X, Y) , of the maze, and facts that specify which cells are walls, input_wall (X, Y) , and which cells are empty, input_empty (X, Y) , in each solution. The grid itself is specified using a number of facts row/1 defining the rows in the grid. These are given as a range of consecutive, ascending integers, starting at 1. Likewise, a number of facts col/1 is used to define the columns in the grid. Instances of maxRow/1 and maxCol/1 are provided as well and represent the number of rows, resp., columns, in the grid.

The output corresponds to valid maze structures represented via empty/2 and wall/2.

Input signature: row/1, col/1, maxRow/1, maxCol/1, entrance/2, exit/2, input_empty/2, input_wall/2

Output signature: empty/2, wall/2

Preconditions:

- Precisely one fact maxRow/1, resp., maxCol/1, specifies the number of rows, resp., columns, of the grid.
- The number of rows or the number of columns is odd (or both numbers are odd).
- Row and Column indices are consecutive integers starting at 1.

- All fields specified via `input_empty/2` and `input_wall/2` are in the grid; at least one cell that has to be empty and one cell that has to be a wall has to be specified.
- A cell is either empty or a wall.
- Precisely one fact `entrance/2`, resp., `exit/2`, represents the entrance, resp., exit, of the maze; entrance and exit have to be located in an edge of the grid.
- Neither the entrance nor the exit can be a wall.
- Exit and entrance are distinct positions in the grid.

Encoding:

```

%%%%%%%% Startup
grid(X,Y) :- col(X), row(Y).

adjacent(X,Y,X,Y1) :- grid(X,Y), Y1 = Y + 1, row(Y1).
adjacent(X,Y,X,Y1) :- grid(X,Y), Y1 = Y - 1, row(Y1).
adjacent(X,Y,X1,Y) :- grid(X,Y), X1 = X + 1, col(X1).
adjacent(X,Y,X1,Y) :- grid(X,Y), X1 = X - 1, col(X1).

border(1,Y) :- row(Y).
border(X,1) :- col(X).
border(X,Y) :- row(Y), maxCol(X).
border(X,Y) :- col(X), maxRow(Y).

%%%%%%%% Input empty cells and walls.
empty(X,Y) :- input_empty(X,Y).
wall(X,Y) :- input_wall(X,Y).

%%%%%%%% Condition 1: Each cell is empty or a wall.
1 { wall(X,Y), empty(X,Y) } 1 :- grid(X,Y), not border(X,Y),
                                not entrance(X,Y), not exit(X,Y).
:- wall(X,Y), empty(X,Y).

%%%%%%%% Condition 2: Each cell in an edge of the grid is a wall,
%%%%%%%% except entrance and exit that are empty.
wall(X,Y) :- border(X,Y), not entrance(X,Y), not exit(X,Y).
empty(X,Y) :- entrance(X,Y).
empty(X,Y) :- exit(X,Y).

%%%%%%%% Condition 3: There is no 2 x 2 square of empty cells or walls.
:- wall(X,Y), wall(X1,Y), wall(X,Y1), wall(X1,Y1),
   X1 = X + 1, Y1 = Y + 1.
:- empty(X,Y), empty(X1,Y), empty(X,Y1), empty(X1,Y1),
   X1 = X + 1, Y1 = Y + 1.

```

```

% Condition 4: If two walls are on a diagonal of a 2 x 2 square,
% then not both of their common neighbors are empty.
:- wall(X,Y), wall(Xp1,Yp1), empty(Xp1,Y), empty(X,Yp1),
   Xp1 = X + 1, Yp1 = Y + 1.
:- wall(Xp1,Y), wall(X,Yp1), empty(X,Y), empty(Xp1,Yp1),
   Xp1 = X + 1, Yp1 = Y + 1.

% Condition 5: No wall is completely surrounded by empty cells.
:- wall(X,Y), not border(X,Y), not wallWithAdjacentWall(X,Y).
wallWithAdjacentWall(X,Y) :- wall(X,Y), adjacent(X,Y,W,Z), wall(W,Z).

% Condition 6: There is a path from the entrance to every
% empty cell (a path is a finite sequence of cells, in
% which each cell is horizontally or vertically
% adjacent to the next cell in the sequence).
reach(X,Y) :- entrance(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(X,Y), empty(XX,YY).
:- empty(X,Y), not reach(X,Y).

```

Input generator:

```

% maze generation
% s ... max number of rows and cols

domR(1..s).
domC(1..s).
1 { maxRow(X) : domR(X) } 1.
1 { maxCol(X) : domC(X) } 1.

:- maxRow(R), maxCol(C), R #mod 2 == 0, C #mod 2 == 0.

row(X) :- domR(X), 1 <= X, X <= R, maxRow(R).
col(X) :- domC(X), 1 <= X, X <= C, maxCol(C).

1 { input_empty(C,R) : domC(C) : domR(R) }.
1 { input_wall(C,R) : domC(C) : domR(R) }.
:- input_empty(C,R), C > CC, maxCol(CC).
:- input_empty(C,R), R > RR, maxRow(RR).
:- input_wall(C,R), C > CC, maxCol(CC).
:- input_wall(C,R), R > RR, maxRow(RR).
:- input_empty(C,R), input_wall(C,R).

1 { entrance(C,R) : domC(C) : domR(R) } 1.
:- entrance(C,R), C > CC, maxCol(CC).
:- entrance(C,R), R > RR, maxRow(RR).
okE :- entrance(1,R).
okE :- entrance(C,R), maxCol(C).
okE :- entrance(C,1).
okE :- entrance(C,R), maxRow(R).

```

```
:- not okE.

1 { exit(C,R) : domC(C) : domR(R) } 1.
:- exit(C,R), C > CC, maxCol(CC).
:- exit(C,R), R > RR, maxRow(RR).
okX :- exit(1,R).
okX :- exit(C,R), maxCol(C).
okX :- exit(C,1).
okX :- exit(C,R), maxRow(R).
:- not okX.

:- entrance(C,R), input_wall(C,R).
:- exit(C,R), input_wall(C,R).
:- entrance(C,R), exit(C,R).

#hide.
#show row/1.           % input
#show col/1.          % input
#show maxRow/1.       % input
#show maxCol/1.       % input
#show entrance/2.     % input
#show exit/2.         % input
#show input_empty/2.  % input
#show input_wall/2.   % input
#show empty/2.        % output
#show wall/2.         % output
```


Bibliography

- [1] Weronika T Adrian, Mario Alviano, Francesco Calimeri, Bernardo Cuteri, Carmine Dodaro, Wolfgang Faber, Davide Fuscà, Nicola Leone, Marco Manna, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV: Advancements and applications. *KI-Künstliche Intelligenz: Vol. 32, No. 2-3*, 2018.
- [2] Hiralal Agrawal, Richard A. Demillo, Bob Hathaway, William Hsu, Wynne Hsu, E.W. Krauser, R. J. Martin, Adytia P. Mathur, and Eugene Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Centre, 1989.
- [3] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015)*, volume 9345 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2015.
- [4] Giovanni Amendola, Tobias Berei, and Francesco Ricca. Testing in ASP: Revisited language and programming environment. In *Proceedings of the 17th European Conference on Logics in Artificial Intelligence (JELIA 2021)*, volume 12678 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2021.
- [5] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the “Small Scope Hypothesis”. Technical report, MIT CSAIL, 2003.
- [6] Marcello Balduccini. Modules and signature declarations for A-Prolog: Progress report. In *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming (SEA 2007)*, pages 41–55, 2007.
- [7] Mutsunori Banbara, Katsumi Inoue, Hiromasa Kaneyuki, Tenda Okimoto, Torsten Schaub, Takehide Soh, and Naoyuki Tamura. catnap: Generating test suites of constrained combinatorial testing with answer set programming. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017)*, volume 10377 of *Lecture Notes in Computer Science*, pages 265–278. Springer, 2017.
- [8] Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. Generating combinatorial test cases by efficient SAT encodings suitable for CDCL

SAT solvers. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)*, volume 6397 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2010.

- [9] Mutsunori Banbara, Naoyuki Tamura, and Katsumi Inoue. Generating event-sequence test cases by answer set programming with the incidence matrix. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, volume 17 of *LIPICs*, pages 86–97. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.
- [10] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, 2003.
- [11] Chitta Baral and Michael Gelfond. Reasoning agents in dynamic domains. In *Workshop on Logic-based Artificial Intelligence*, pages 257–279. Kluwer Academic Publishers, 2000.
- [12] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003.
- [13] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1999.
- [14] Fevzi Belli and Oliver Jack. Declarative paradigm of test coverage. *Software Testing, Verification and Reliability*, 8(1):15–47, 1998.
- [15] Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. Automatic music composition using answer set programming. *Theory and Practice of Logic Programming*, 11(2–3):397–427, 2011.
- [16] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 123–133. ACM, 2002.
- [17] Martin Brain, Owen Cliffe, and Marina De Vos. A pragmatic programmer’s guide to answer set programming. In *Proceedings of the 2nd International Workshop on Software Engineering for Answer Set Programming (SEA 2009)*, pages 49–63, 2009.
- [18] Martin Brain, Tom Crick, Marina De Vos, and John Fitch. TOAST: Applying answer set programming to superoptimisation. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, volume 4079 of *Lecture Notes in Computer Science*. Springer, 2006.
- [19] Martin Brain, Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jörg Pührer, Hans Tompits, and Cemal Yilmaz. Event-sequence testing using answer-set programming. *International Journal on Advances in Software*, 5(3 & 4):237–251, 2012.

- [20] Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI 2007)*, pages 385–390. AAAI Press, 2007.
- [21] Daniel R. Brooks, Esra Erdem, Selim T. Erdogan, James W. Minett, and Donald Ringe. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning*, 39(4):471–511, 2007.
- [22] Robert Brownlie, James Prowse, and Madhav S. Phadke. Robust testing of AT&T PMX/starMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, 1992.
- [23] Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pairwise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.
- [24] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *The Journal of Logic Programming*, 19 & 20:443–502, 1994.
- [25] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the Fifth Answer Set Programming Competition. *Artificial Intelligence*, 231(C):151–181, 2016.
- [26] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. Third ASP Competition: File and language formats. <http://www.mat.unical.it/aspcomp2011/files/LanguageSpecifications.pdf>.
- [27] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The Third Open Answer Set Programming Competition. *Theory and Practice of Logic Programming*, 14(1):117–135, 2012.
- [28] Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The Third Answer Set Programming Competition: Preliminary report of the system competition track. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, volume 6645 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2011.
- [29] Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010.
- [30] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

- [31] Owen Cliffe, Marina De Vos, Martin Brain, and Julian A. Padget. ASPVIZ: Declarative visualisation and animation using answer set programming. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *Lecture Notes in Computer Science*, pages 724–728. Springer, 2008.
- [32] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.
- [33] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 16th ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–139. ACM, 2007.
- [34] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.
- [35] Myra B. Cohen, Peter B. Gibbons, and Warwick B. Mugridge. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 38–48. IEEE Computer Society, 2003.
- [36] Jacek Czerwonka. Pairwise testing in real world. In *Proceedings of the 24th Pacific Northwest Software Quality Conference (PNSQC 2006)*, pages 419–430, 2006.
- [37] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE 1999)*, pages 285–295. Association for Computing Machinery, 1999.
- [38] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
- [39] Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Annotating answer-set programs in LANA. *Theory and Practice of Logic Programming*, 12(4-5):619–637, 2012.
- [40] Marina De Vos and Torsten Schaub, editors. *First International Workshop on Software Engineering for Answer Set Programming (SEA 2007)*, 2007.
- [41] Marina De Vos and Torsten Schaub, editors. *Second International Workshop on Software Engineering for Answer Set Programming (SEA 2009)*, 2009.
- [42] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

- [43] Gulsen Demiroz and Cemal Yilmaz. Cost-aware combinatorial interaction testing. In *Proceedings of the 4th International Conference on Advances in System Testing and Validation Lifecycle (VALID 2012)*, pages 9–16. Xpert Publishing Services, 2012.
- [44] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Mirosław Trzuszczński. The Second Answer Set Programming Competition. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 637–654. Springer, 2009.
- [45] DLV User Manual. www.dlvsystem.com/html/DLV_User_Manual.html.
- [46] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *Journal of Experimental & Theoretical Artificial Intelligence*, 21(2):79–121, 2009.
- [47] Rod Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, New York, 1999.
- [48] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions Software Engineering*, 10(4):438–444, 1984.
- [49] Eclipse. www.eclipse.org.
- [50] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The diagnosis frontend of the DLV system. *AI Communications*, 12(1-2):99–111, 1999.
- [51] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Planning under incomplete knowledge. In *Proceedings of the 1st International Conference on Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Computer Science*, pages 807–821. Springer, 2000.
- [52] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. The DLV^k planning system: Progress report. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA 2002)*, volume 2424 of *Lecture Notes in Computer Science*, pages 541–544. Springer, 2002.
- [53] Thomas Eiter, Georg Gottlob, and Helmut Veith. Modular logic programming and generalized quantifiers. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 1997)*, volume 1265 of *Lecture Notes in Computer Science*, pages 290–309. Springer, 1997.
- [54] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer Set Programming: A Primer. In *Reasoning Web. Semantic Technologies for Information Systems.*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

- [55] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system DLV: Progress report, comparisons and benchmarks. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR 1998)*, pages 406–417. Morgan Kaufmann, 1998.
- [56] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016.
- [57] Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jörg Pührer, Hans Tompits, and Cemal Yilmaz. Answer-set programming as a new approach to event-sequence testing. In *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, pages 25–34. Xpert Publishing Services, 2011.
- [58] Esra Erdem, Vladimir Lifschitz, and Donald Ringe. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming*, 6(5):539–558, 2006.
- [59] Flavio Everardo. Towards an automated multitrack mixing tool using answer-set programming. In *Proceedings of the 14th Sound and Music Computing Conference (SMC 2017)*, pages 422–428, 2017.
- [60] Flavio Everardo and Fernando Aguilera. Armin: Automatic trance music composition using answer set programming. *Fundamenta Informaticae*, 113(1):79–96, 2011.
- [61] Flavio Everardo, Tomi Janhunnen, Roland Kaminski, and Torsten Schaub. The Return of xorro. In *Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019)*, volume 11481 of *Lecture Notes in Computer Science*, pages 284–297. Springer, 2019.
- [62] Andreas Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich C. Teppan. Industrial applications of answer set programming. *KI-Künstliche Intelligenz*, 32(2-3):165–176, 2018.
- [63] Min Fang and Hans Tompits. An approach for representing answer sets in natural language. In *Proceedings of the Declarative Programming and Knowledge Management Conference on Declarative Programming (DECLARE 2017), Unifying INAP, WFLP, and WLP, Revised Selected Papers*, volume 10997 of *Lecture Notes in Computer Science*, pages 115–131. Springer, 2018.
- [64] Onofrio Febraro, Nicola Leone, Kristian Reale, and Francesco Ricca. Unit testing in ASPIDE. In *Proceedings of the 25th Workshop on Logic Programming (WLP 2011) and the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011), Revised Selected Papers*, volume 7773 of *Lecture Notes in Computer Science*, pages 345–364. Springer, 2013.

- [65] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. ASPIDE: Integrated development environment for answer set programming. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, volume 6645 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2011.
- [66] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. Testing ASP programs in ASPIDE. In *Proceedings of the 26th Italian Conference on Computational Logic (CILC 2011)*, volume 810 of *CEUR Workshop Proceedings*, pages 115–129. CEUR-WS.org, 2011.
- [67] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1-2):45–74, 2005.
- [68] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006.
- [69] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
- [70] Martin Gebser, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Sven Thiele. On the input language of ASP grounder gringo. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 502–508. Springer, 2009.
- [71] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 386–392. AAAI Press/The MIT Press, 2007.
- [72] Martin Gebser, Marco Maratea, and Francesco Ricca. The Sixth Answer Set Programming Competition. *Journal of Artificial Intelligence Research*, 60:41–95, 2017.
- [73] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: A new grounder for answer set programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.
- [74] Michael Gelfond and Alfredo Gabaldon. Building a knowledge base: An example. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):165–199, 1999.
- [75] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, 2014.

- [76] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, pages 1070–1080. MIT Press, 1988.
- [77] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
- [78] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. SAT-based answer set programming. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004) and the 16th Conference on Innovative Applications of Artificial Intelligence (IAAI 2004)*, pages 61–66. AAAI Press/The MIT Press, 2004.
- [79] Carla P. Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. Short XORs for model counting: From theory to practice. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007)*, volume 4501 of *Lecture Notes in Computer Science*, pages 100–106. Springer, 2007.
- [80] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, pages 51–61. AAAI Press, 2006.
- [81] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Near-uniform sampling of combinatorial spaces using XOR constraints. In *Proceedings of the 20th Annual Conference on Neural Information Processing Systems (NIPS 2006)*, pages 481–488. MIT Press, 2006.
- [82] Susanne Grell, Torsten Schaub, and Joachim Selbig. Modelling biological networks by action languages via answer set programming. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, volume 4079 of *Lecture Notes in Computer Science*, pages 285–299. Springer, 2006.
- [83] Alexander Greßler, Johannes Oetsch, and Hans Tompits. Harvey: A system for random testing in ASP. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017)*, volume 10377 of *Lecture Notes in Computer Science*, pages 229–235. Springer, 2017.
- [84] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: A survey. *Software Testing, Verification & Reliability*, 15(3):167–199, 2005.
- [85] `gringo` and `clasp`. <https://github.com/potassco/clingo>.
- [86] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
- [87] Mary Jean Harrold and Brian A. Malloy. Data flow testing of parallelized code. In *Proceedings of the 8th International Conference on Software Maintenance (ICSM 1992)*, pages 272–281. IEEE Computer Society Press, 1992.

- [88] Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1-3):149–156, 2004.
- [89] Brahim Hnich, Steven David Prestwich, Evgeny Selensky, and Barbara M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.
- [90] Jungchang Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, 1975.
- [91] Oliver Jack. *Software Testing for Conventional and Logic Programming*. Walter de Gruyter & Co. Hawthorne, NJ, USA, 1996.
- [92] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, 1996.
- [93] Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. On testing answer-set programs. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, Frontiers in Artificial Intelligence and Applications, pages 951–956. IOS Press, 2010.
- [94] Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Random vs. structure-based testing of answer-set programs: An experimental comparison. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, volume 6645 of *Lecture Notes in Computer Science*, pages 242–247. Springer, 2011.
- [95] Tomi Janhunen and Emilia Oikarinen. LPEQ and DLPEQ – Translators for automated equivalence testing of logic programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004)*, volume 2923 of *Lecture Notes in Computer Science*, pages 336–340. Springer, 2004.
- [96] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity aspects of disjunctive stable models. *Journal Artificial Intelligence Research*, 35:813–857, 2009.
- [97] Javadoc. www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.
- [98] JUnit. www.junit.org.
- [99] Richard Karp. Reducibility among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer, 1972.

- [100] Sunwoo Kim, John A. Clark, and John A. McDermid. The rigorous generation of Java mutation operators using HAZOP. In *Proceedings of the 12th International Conference on Software & Systems Engineering and their Applications (ICSSEA 1999)*, 1999.
- [101] Kim King and Jeff Offutt. A Fortran language system for mutation-based software testing. *Software – Practice and Experience*, 21(7):685–718, 1991.
- [102] Christian Kloimüller, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *Proceedings of the 25th Workshop on Logic Programming (WLP 2011) and the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011), Revised Selected Papers*, volume 7773 of *Lecture Notes in Computer Science*, pages 325–344. Springer, 2013.
- [103] D. Richard Kuhn, James M. Higdon, James Lawrence, Raghu Kacker, and Yu Lei. Combinatorial methods for event sequence testing. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST 2012)*, pages 601–609. IEEE, 2012.
- [104] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.
- [105] Marija Kulas. Annotations for Prolog – A concept and runtime handling. In *Proceedings of the 9th International Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 1999), Selected Papers*, volume 1817 of *Lecture Notes in Computer Science*, pages 234–254. Springer, 2000.
- [106] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. <https://www.cs.ucf.edu/~leavens/JML/jmldbc.pdf>, 2006.
- [107] Joohyung Lee. A model-theoretic counterpart of loop formulas. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 503–508. Professional Book Center, 2005.
- [108] Vladimir Lifschitz. Answer set programming and plan generation. *Journal of Artificial Intelligence*, 138(1-2):39–54, 2002.
- [109] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 1594–1597. AAAI Press, 2008.
- [110] Vladimir Lifschitz and Alexander A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268, 2006.
- [111] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program with SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.

- [112] Christopher M. Lott, Ashish Jain, and Siddhartha R. Dalal. Modeling requirements for combinatorial software testing. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.
- [113] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A study of interleaving coverage criteria. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 533–536. ACM, 2007.
- [114] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, pages 329–339. ACM, 2008.
- [115] Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, Artificial Intelligence, pages 375–398. Springer, 1999.
- [116] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory Report MIT-LCS-TR-921, 2003.
- [117] John McCarthy. Elaboration tolerance. In *Proceedings of the 4th Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense 98)*, pages 198–217, 1998.
- [118] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.
- [119] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, 2011.
- [120] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
- [121] Ilkka Niemelä and Patrik Simons. Smodels – An implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 1997)*, volume 1265 of *Lecture Notes in Computer Science*, pages 420–429. Springer, 1997.
- [122] Ilkka Niemelä, Patrik Simons, and Timo Soinen. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 1999)*, volume 1730 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 1999.

- [123] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL 2001)*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2001.
- [124] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988.
- [125] Kari J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004.
- [126] Johannes Oetsch and Juan-Carlos Nieves. Stable-ordered models for propositional theories with order operators. In *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA 2019)*, volume 11468 of *Lecture Notes in Computer Science*, pages 794–802. Springer, 2019.
- [127] Johannes Oetsch, Michael Prischink, Jörg Pührer, Martin Schwengerer, and Hans Tompits. On the small-scope hypothesis for testing answer-set programs. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*, pages 43–53. AAAI Press, 2012.
- [128] Johannes Oetsch, Jörg Pührer, Martin Schwengerer, and Hans Tompits. Kato: A plagiarism-detection tool for answer-set programs. In *Proceedings of the 23rd Workshop on (Constraint) Logic Programming (WLP 2010)*, pages 75–79. Universitätsverlag Potsdam, 2009.
- [129] Johannes Oetsch, Jörg Pührer, Martin Schwengerer, and Hans Tompits. The system Kato: Detecting cases of plagiarism for answer-set programs. *Theory and Practice of Logic Programming*, 10(4–6):759–775, 2010.
- [130] Johannes Oetsch, Jörg Pührer, Martina Seidl, Hans Tompits, and Patrick Zwickl. VIDEAS: A development tool for answer-set programs based on model-driven engineering technology. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, volume 6645 of *Lecture Notes in Computer Science*, pages 382–387. Springer, 2011.
- [131] Johannes Oetsch, Jörg Pührer, Martina Seidl, Hans Tompits, and Patrick Zwickl. Videas: Supporting answer-set program development using model-driven engineering techniques. In *Proceedings of the MELO 2011 Workshop: Model-Driven Engineering, Logic and Optimization: Friends or foes?*, 2011.
- [132] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Catching the Ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming*, 10(4–6):513–529, 2010.

- [133] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Let's break the rules: Interactive procedural-style debugging of answer-set programs. In *Proceedings of the 24th Workshop on (Constraint) Logic Programming (WLP 2010)*, pages 77–87. Technical Report, Faculty of Media Engineering and Technology, German University in Cairo, 2010.
- [134] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Methods and Methodologies for Developing Answer-Set Programs – Project Description. In *Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010)*, volume 7 of *LIPICs*, pages 154–161. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.
- [135] Johannes Oetsch, Jörg Pührer, and Hans Tompits. The SeaLion has landed: An IDE for answer-set programming – Preliminary report. In *Proceedings of the 25th Workshop on Logic Programming (WLP 2011)*, pages 141–151. INFSYS Research Report, 1843-11-06, 2011.
- [136] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Stepping through an answer-set program. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, volume 6645 of *Lecture Notes in Computer Science*, pages 134–147. Springer, 2011.
- [137] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Extending object-oriented languages by declarative specifications of complex objects using answer-set programming. In *Proceedings of the 26th Workshop on Logic Programming (WLP 2012)*, abs/1112.0922. CoRR, 2012.
- [138] Johannes Oetsch, Jörg Pührer, and Hans Tompits. An FLP-style answer-set semantics for abstract-constraint programs with disjunctions. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, volume 17 of *LIPICs*, pages 222–234. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.
- [139] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Stepwise debugging of description-logic programs. In *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 492–508. Springer, 2012.
- [140] Johannes Oetsch, Jörg Pührer, and Hans Tompits. The SeaLion has landed: An IDE for answer-set programming – Preliminary report. In *Proceedings of the 25th Workshop on Logic Programming (WLP 2011) and the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)*, *Revised Selected Papers*, volume 7773 of *Lecture Notes in Computer Science*, pages 305–324. Springer, 2013.
- [141] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Stepwise debugging of answer-set programs. *Theory and Practice of Logic Programming*, 18(1):30–80, 2018.

- [142] Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. Testing relativised uniform equivalence under answer-set projection in the system $cc\top$. In *Proceedings of the 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and the 21st Workshop on Logic Programming (WLP 2007), Revised Selected Papers*, volume 5437 of *Lecture Notes in Computer Science*, pages 241–246. Springer, 2007.
- [143] Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. $cc\top$ on stage: Generalised uniform equivalence testing for verifying student assignment solutions. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 382–395. Springer, 2009.
- [144] Johannes Oetsch, Hans Tompits, and Stefan Woltran. Facts do not cease to exist because they are ignored: Relativised uniform equivalence with answer-set projection. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI 2007)*, pages 458–464. AAAI Press, 2007.
- [145] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering (ICSE 1993)*, pages 100–107. IEEE Computer Society Press, 1993.
- [146] Jeff Offutt, Jeff Voas, and Jeff Payne. Mutation operators for Ada. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, 1996.
- [147] Emilia Oikarinen and Tomi Janhunen. A translation-based approach to the verification of modular equivalence. *Journal of Logic and Computation*, 19(4):591–613, 2009.
- [148] Sarah Opolka, Philipp Obermeier, and Torsten Schaub. Automatic genre-dependent composition using answer set programming. In *Proceedings of the 21st International Symposium on Electronic Art (ISEA 2015)*, 2015.
- [149] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.
- [150] Simona Perri, Francesco Ricca, Giordio Terracina, D. Cianni, and Pierfrancesco Veltri. An integrated graphic tool for developing and testing DLV programs. In *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming (SEA 2007)*, pages 86–100, 2007.
- [151] PLUnit. www.swi-prolog.org/pldoc/package/plunit.html.
- [152] Axel Polleres. Semantic Web Languages and Semantic Web Services as Application Areas for Answer Set Programming. In *Nonmonotonic Reasoning, Answer Set*

Programming and Constraints, volume 05171 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

- [153] Potassco Documentation. <https://potassco.org/doc/>.
- [154] Potassco, the Potsdam Answer Set Solving Collection. <https://potassco.org/>.
- [155] Jörg Pührer and Hans Tompits. Casting away disjunction and negation under a generalisation of strong equivalence with projection. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 264–276. Springer, 2009.
- [156] John S. Schlipf. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences*, 51(1):64–86, 1995.
- [157] Peter Schüller. Adjudication of coreference annotations via answer set optimization. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017)*, volume 10377 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2017.
- [158] Oded Shmueli. Decidability and expressiveness of logic queries. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 237–249, 1987.
- [159] Patrik Simons, Ilkka Niemelä, and Timo Soinen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
- [160] T. Soinen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL 1999)*, volume 1551 of *Lecture Notes in Computer Science*. Springer, 1999.
- [161] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.
- [162] Adrian Sureshkumar, Marina De Vos, Martin Brain, and John Fitch. APE: An AnsProlog* environment. In *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming (SEA 2007)*, pages 101–115, 2007.
- [163] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.

- [164] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.
- [165] Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. LANA: A language for annotating answer-set programs. In *Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR 2012)*, 2012.
- [166] Wenhua Wang, Sreedevi Sampath, Yu Lei, and Raghu Kacker. An interaction-based test sequence generation approach for testing web applications. In *Proceedings of the 11th IEEE International Symposium on High Assurance Systems Engineering*, pages 209–218. IEEE, 2008.
- [167] Cemal Yilmaz. Test case-aware combinatorial interaction testing. *IEEE Transactions on Software Engineering*, 39(5):684–706, 2013.
- [168] Cemal Yilmaz, Myra B. Cohen, and Adam A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1):20–34, 2006.
- [169] Linbin Yu, Yu Lei, Raghu Kacker, D. Richard Kuhn, and James Lawrence. Efficient algorithms for t-way test sequence generation. In *Proceedings of the 17th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2012)*, pages 220–229. IEEE Computer Society, 2012.
- [170] Xun Yuan, Myra B. Cohen, and Atif M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 405–408. ACM, 2007.
- [171] Kamal Y. Zamli, Rozmie R. Othman, and Mohd H. Mohamed Zabil. On sequence based interaction testing. In *Proceedings of the 2011 IEEE Symposium on Computers Informatics (ISCI 2011)*, pages 662–667, 2011.

Curriculum Vitae

Name: Johannes Oetsch

Degrees: Bakk.techn. (~ B.Sc.), Dipl.Ing. (~M.Sc.)

Date and place of birth: 15.02.1981, Vienna, Austria

Citizenship: Austria

Current affiliation: Researcher at the Institute of Logic and Computation, Technische Universität Wien.



Coordinates:

Institute of Logic and Computation

Knowledge-Based Systems Group 192-03

Technische Universität Wien

Favoritenstraße 9-11

A-1040 Wien, Austria

Web: <http://www.kr.tuwien.ac.at/staff/oetsch/>

Email: johannes.oetsch@gmail.com

Keywords: Logic-based Artificial Intelligence; Knowledge Representation and Reasoning; Answer-Set Programming; Non-monotonic Reasoning

Research interest: Current research interests are focused on ASP for hard scheduling problems in an industrial setting as well as combining ASP and machine learning.

Major career steps:

- Since February 2020: Researcher, Institute of Logic and Computation, Technische Universität Wien, in cooperation with the Bosch Center for AI; research on scheduling and combinations of machine learning, meta heuristics, and ASP.

- August 2017 – July 2019: Researcher, Department of Computer Science, Umeå University, Sweden; research on knowledge representation and explainable AI for autonomous systems that recognise, explain, and predict complex human activities.
- September 2013 – July 2017: Teaching Assistant (Univ.Ass.), Theory and Logic Group, Technische Universität Wien; involved with courses on theoretical computer science and logic, formal languages, databases, and programming.
- September 2009 – August 2013: Project Assistant (FWF), Knowledge-Based Systems Group, Technische Universität Wien; research on testing and debugging for ASP.
- February 2011 – April 2011: Internship, National Institute of Informatics, Tokyo, Japan; supervised by Katsumi Inoue; research on logic-based methods for event-sequence testing.
- March 2007 – August 2009: Teaching Assistant, Institute for Information Systems, Technische Universität Wien; involved with courses on knowledge-based systems, logic programming, and artificial intelligence.
- Oktober 2006 – February 2007: Tutor, Institute for Information Systems, Technische Universität Wien; involved with courses on ASP-based planning and diagnosis.

Education and certificates:

- Master of Science, Technische Universität Wien, Computational Intelligence (with distinction), January 2013.
- Bachelor of Science, Technische Universität Wien, Software & Information Engineering (with distinction), January 2008.
- Matura (higher school certificate), Bundesrealgymnasium Wien XIX (with distinction), June 1999.

Research projects in Sweden:

- (2017–2019) *Autonomous Systems that Recognise, Explain, and Predict Complex Human Activities*. This project was funded by the Kempe Foundation and led by Juan Carlos Nieves. I developed knowledge representation and automated reasoning techniques for recognising, explaining, and predicting complex human activities.
- (2018–2019) *CA17124 - Digital Forensics: Evidence Analysis via Intelligent Systems and Practices*. The aim of this research network with 32 countries in collaboration is to explore the potential of the application of Artificial Intelligence and Automated Reasoning in the Digital Forensics field, and creating synergies between these fields.

Research internship in Japan:

- (February 2011–April 2011) *Internship at the National Institute of Informatics, Tokyo, Japan*. Together with Katsumi Inoue, I worked on logic-based methods for combinatorial testing. In particular, ASP was used as a new approach to develop more elaboration tolerant solutions to event-sequence testing problems that take domain knowledge into account.

Research projects in Austria:

- (2014–2016) *ARiSE - Austrian Society for Rigorous Systems Engineering*; funded by the FWF under grant S11409-N23. I was mainly working on incremental SAT and QBF solving and benchmarking of incremental SAT and QBF solvers. This work was supervised by Uwe Egly.¹
- (2009–2013) *Methods and Methodologies for Developing Answer-Set Programs*; funded by the FWF under grant P21698. I coauthored 27 peer-reviewed publications on different aspects of programming support for ASP like testing, debugging, or systematic program development. Selected publications form the basis for my PhD thesis supervised by Hans Tompits.²
- (2006–2008) *Formal Methods for Comparing and Optimizing Nonmonotonic Logic Programs*; funded by the Austrian Science Fund (FWF) under grant P18019. As internship student, I coauthored 13 internationally peer-reviewed publications dealing with advanced notions of program equivalence between answer-set programs. These results formed the basis for my bachelor thesis as well as for my master thesis, both supervised by Hans Tompits.³

Other career-related activities:

- Editorship:
 - Coeditor of Applications of Declarative Programming and Knowledge Management; 19th International Conference, INAP 2011, and the 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28–30, 2011, Revised Selected Papers.
- Program Committees:
 - TAASP 2020: Workshop on Trends and Applications of Answer Set Programming;

¹<https://arise.or.at/>.

²<http://www.kr.tuwien.ac.at/research/projects/mmdasp/>.

³<http://www.kr.tuwien.ac.at/research/projects/eq/>.

- VALID 2013–2016: International Conference on Advances in System Testing and Validation Lifecycle;
- IJCAI 2011; International Joint Conference on Artificial Intelligence.
- Organising Committees:
 - Workshop of the Swedish AI Society (SAIS 2019) , Umeå, Sweden, June 18–19, 2019;
 - 25th Workshop on Logic Programming (WLP 2011), Vienna, Austria, September 28–30, 2011.
- Reviews for numerous journals, conferences and workshops, including ACM TOCL, IJCAI, AAI, KR, JELIA, ICLP, LPNMR, SAT, EPIA, INAP, CLIMA, WLP, LOPSTR, PADL, VALID, SEA, WOLLIC, and CENT.

Prizes and awards:

- I received the IEEE diploma thesis award for my work “Beyond uniform equivalence between answer-set programs: Relativisation and Projection” (July 2013).

Publications: A publication list including PDFs is available online from the publication database of the Technische Universität Wien.⁴ Publications can also be found at DBLP⁵ or Google Scholar.⁶

1. Thomas Eiter, Tobias Geibinger, Nysret Musliu, Johannes Oetsch, Peter Skocovsky, and Daria Stepanova. Answer-set programming for lexicographical makespan optimisation in parallel machine scheduling. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning (KR 2021)*, to appear.
2. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. Beyond uniform equivalence between answer-set programs. *ACM Transactions on Computational Logic (TOCL)* 22(1), 1-46, 2020.
3. Johannes Oetsch and Juan Carlos Nieves. Stable-ordered models for propositional theories with order operators. In *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA 2019)*, volume 11468 of *Lecture Notes in Artificial Intelligence*, pages 794–802, Springer, 2019.
4. Johannes Oetsch, Jörg Pührer, and Hans Tompits. Stepwise debugging of answer-set programs. *Theory and Practice of Logic Programming* 18(1): 30-80, 2018.

⁴<http://publik.tuwien.ac.at/publist.php?lang=2&dousealiases=1&zuname=Oetsch&vorname=Johannes&sort=3&inv=1&num=1&ext=1>.

⁵<https://dblp.org/pers/hd/o/Oetsch:Johannes>.

⁶<https://scholar.google.se/citations?hl=en&user=8jNGQEQAATAJ>.

5. Johannes Oetsch and Juan Carlos Nieves. A knowledge representation perspective on activity theory. *CoRR* abs/1811.05815, 2018.
6. Alexander Greßler, Johannes Oetsch, Hans Tompits. Harvey: A system for random testing in ASP. In *Proceedings of the 14th Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017)*, volume 10377 of *Lecture Notes in Artificial Intelligence*, pages 229–235, Springer, 2017.
7. Uwe Egly, Florian Lonsing, and Johannes Oetsch. Automated benchmarking of incremental SAT and QBF solvers. In *Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2015)*, volume 9450 of *Lecture Notes in Computer Science*, pages 178–186, Springer, 2015.
8. Mario Alviano, Francesco Calimeri, Günther Charwat, Minh Dao-Tran, Carmine Dodaro, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, Jörg Pührer, Christoph Redl, Francesco Ricca, Patrik Schneider, Martin Schwengerer, Lara Katharina Spendier, Johannes Peter Wallner, Guohui Xiao. The Fourth Answer Set Programming Competition: Preliminary Report. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, volume 8148 of *Lecture Notes in Computer Science*, pages 42–53, Springer, 2013.
9. Paula-Andra Busoniu, Johannes Oetsch, Jörg Pührer, Peter Skocovsky and Hans Tompits. SeaLion: An Eclipse-based IDE for answer-set programming with advanced debugging support. *Theory and Practice of Logic Programming*, Volume 13, Special Issue 4-5 (29th International Conference on Logic Programming), pages 657–673, Cambridge University Press, 2013.
10. Johannes Oetsch, Jörg Pührer, and Hans Tompits. The SeaLion has landed: An IDE for answer-set programming – Preliminary Report. In *Proceedings of the 25th Workshop on Logic Programming (WLP 2011) and the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)*, Revised Selected Papers, *Lecture Notes in Artificial Intelligence*, pages 305–324, Springer, 2013.
11. Christian Kloimüller, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *Proceedings of the 25th Workshop on Logic Programming (WLP 2011) and the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)*, Revised Selected Papers, *Lecture Notes in Artificial Intelligence*, pages 325–344, Springer, 2013.
12. Johannes Oetsch. Beyond uniform equivalence between answer-set programs: Relativisation and projection. *Master’s thesis*, Technische Universität Wien, 2012.

13. Martin Brain, Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jörg Pührer, Hans Tompits, and Cemal Yilmaz. Event-sequence testing using answer-set programming. *International Journal on Advances in Software*, 5(3 & 4):237–251, 2012.
14. Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Annotating answer-set programs in LANA. *Theory and Practice of Logic Programming*, 12(4-5):619–637, 2012.
15. Johannes Oetsch, Jörg Pührer, and Hans Tompits. Stepwise debugging of description-logic programs. In *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*, pages 492–508. Springer, 2012.
16. Johannes Oetsch, Jörg Pührer, and Hans Tompits. An FLP-style answer-set semantics for abstract-constraint programs with disjunctions. In *Proceedings of the 28th International Conference on Logic Programming (ICLP 2012)*, *Technical Communications*, volume 17 of *LIPICs*, pages 222–234. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.
17. Johannes Oetsch, Michael Prischink, Jörg Pührer, Martin Schwengerer, and Hans Tompits. On the small-scope hypothesis for testing answer-set programs. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*, pages 43–53. AAAI Press, 2012.
18. Johannes Oetsch, Jörg Pührer, and Hans Tompits. Extending object-oriented languages by declarative specifications of complex objects using answer-set programming. In *Proceedings of the 26th Workshop on Logic Programming (WLP 2012)*, abs/1112.0922. CoRR, 2012.
19. Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. LANA: A language for annotating answer-set programs. In *Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR 2012)*, 2012.
20. Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jörg Pührer, Hans Tompits, and Cemal Yilmaz. Answer-set programming as a new approach to event-sequence testing. In *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, pages 25–34. Xpert Publishing Services, 2011.
21. Johannes Oetsch, Jörg Pührer, and Hans Tompits. Stepping through an answer-set program. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, volume 6645 of *Lecture Notes in Computer Science*, pages 134–147. Springer, 2011.
22. Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Random vs. structure-based testing of answer-set programs: An experimental

comparison. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR 2011), volume 6645 of *Lecture Notes in Computer Science*, pages 242–247. Springer, 2011.

23. Johannes Oetsch and Hans Tompits. Gentzen-type refutation systems for three-valued logics with an application to disproving strong equivalence. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR 2011), volume 6645 of *Lecture Notes in Computer Science*, pages 254–259. Springer, 2011.
24. Johannes Oetsch, Jörg Pührer, Martina Seidl, Hans Tompits, and Patrick Zwickl. VIDEAS: A development tool for answer-set programs based on model-driven engineering technology. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR 2011), volume 6645 of *Lecture Notes in Computer Science*, pages 382–387. Springer, 2011.
25. Christian Kloimüller, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *Proceedings of the 25th Workshop on Logic Programming* (WLP 2011), pages 152–164. INFSYS Research Report 1843-11-06, 2011.
26. Johannes Oetsch, Jörg Pührer, and Hans Tompits. The SeaLion has landed: An IDE for answer-set programming – Preliminary report. In *Proceedings of the 25th Workshop on Logic Programming* (WLP 2011), pages 141–151. INFSYS Research Report, 1843-11-06, 2011.
27. Johannes Oetsch, Jörg Pührer, Martina Seidl, Hans Tompits, and Patrick Zwickl. VIDEAS: Supporting answer-set program development using model-driven engineering techniques. In *Proceedings of the MELO 2011 Workshop: Model-Driven Engineering, Logic and Optimization: Friends or Foes?*, 2011.
28. Johannes Oetsch, Jörg Pührer, and Hans Tompits. Catching the Ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming*, 10(4-6):513–529, 2010.
29. Johannes Oetsch, Jörg Pührer, Martin Schwengerer, and Hans Tompits. The system Kato: Detecting cases of plagiarism for answer-set programs. *Theory and Practice of Logic Programming*, 10(4-6):759–775, 2010.
30. Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. On testing answer-set programs. In *Proceedings of the 19th European Conference on Artificial Intelligence* (ECAI 2010), volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 951–956. IOS Press, 2010.
31. Johannes Oetsch, Jörg Pührer, and Hans Tompits. Methods and methodologies for developing answer-set programs—Project description. In *Proceedings of the 26th*

International Conference on logic programming (ICLP 2010), *Technical Communications*, volume 7 of *LIPICs*, pages 154–161. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

32. Johannes Oetsch, Jörg Pührer, and Hans Tompits. Let’s break the rules: Interactive procedural-style debugging of answer-set programs. In *Proceedings of the 24th Workshop on (Constraint) Logic Programming (WLP 2010)*, pages 77–87. Technical Report, Faculty of Media Engineering and Technology, German University in Cairo, 2010.
33. Johannes Oetsch and Hans Tompits. Gentzen-type refutation systems for three-valued logics. In *Proceedings of the 24th Workshop on (Constraint) Logic Programming (WLP 2010)*, pages 88–98. Technical Report, Faculty of Media Engineering and Technology, German University in Cairo, 2010.
34. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. $cc\top$ on stage: Generalised uniform equivalence testing for verifying student assignment solutions. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 382–395. Springer, 2009.
35. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. Testing relativised uniform equivalence under answer-set projection in the system $cc\top$. In *Proceedings of 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007)*, Revised Selected Papers, volume 5437 of *Lecture Notes in Computer Science*, pages 241–246. Springer, 2009.
36. Johannes Oetsch, Jörg Pührer, Martin Schwengerer, and Hans Tompits. Kato: A plagiarism-detection tool for answer-set programs. In *Proceedings of the 23rd Workshop on (Constraint) Logic Programming (WLP 2010)*, pages 75–79. Universitätsverlag Potsdam, 2009.
37. Johannes Oetsch and Hans Tompits. Program correspondence under the answer-set semantics: The non-ground case. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *Lecture Notes in Computer Science*, pages 591–605. Springer, 2008.
38. Johannes Oetsch and Hans Tompits. A generalised program-correspondence framework: Preliminary report. In *Proceedings of the 22nd Workshop on Logic Programming (WLP 2008)*, pages 72–82. Technical Report, University Halle-Wittenberg, Institute of Computer Science, Martin-Luther-University Halle-Wittenberg, 2008.
39. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. An extension of the system $cc\top$ for testing relativised uniform equivalence under answer-set projection. In *Proceedings of the 16th International Conference on Computing (CIC 2007)*, 2007.

40. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. Testing relativised uniform equivalence under answer-set projection in the system $cc\top$. In *Proceedings of the 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and the 21st Workshop on Logic Programming (WLP 2007)*, volume 5437 of *Lecture Notes in Computer Science*, pages 241–246. Springer, 2007.
41. Johannes Oetsch, Hans Tompits, and Stefan Woltran. Facts do not cease to exist because they are ignored: Relativised uniform equivalence with answer-set projection. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI 2007)*, pages 458–464. AAAI Press, 2007.
42. Johannes Oetsch, Hans Tompits, and Stefan Woltran. Facts do not cease to exist because they are ignored: Relativised uniform equivalence with answer-set projection. Poster at the *22nd AAAI Conference on Artificial Intelligence (AAAI 2007)*, 2007.
43. Johannes Oetsch, Hans Tompits, and Stefan Woltran. Facts do not cease to exist because they are Ignored: Relativised uniform equivalence with answer-set projection. In *Proceedings of the LPNMR 2007 Workshop on Correspondence and Equivalence for Nonmonotonic Theories (CENT 2007)*, pages 25–36. 2007.
44. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. $cc\top$: A tool for checking advanced correspondence problems in answer-set programming. In *Proceedings 15th International Conference on Computing (CIC 2006)*, pages 3–11. IEEE Computer Society, 2006.
45. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. $cc\top$: A correspondence-checking tool for logic programs under the answer-set semantics. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA 2006)*, volume 4160 of *Lecture Notes in Computer Science*, pages 502–505. Springer, 2006.
46. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. $cc\top$: A tool for checking advanced correspondence problems in answer-set programming. In *Proceedings of the 1st International Workshop on Logic and Search (LaSh 2006)*, 2006.
47. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. A tool for advanced correspondence checking in answer-set programming. In *Proceedings of the 11th International Workshop on Nonmonotonic Reasoning (NMR 2006)*, volume IfI-06-04 of *IfI Technical Report Series*, pages 20–28. Institut für Informatik, Technische Universität Clausthal, 2006.
48. Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. A tool for advanced correspondence checking in answer-set programming: Preliminary experimental results. In *Proceedings of the 20th Workshop on Logic Programming (WLP*

2006), volume 1843-06-02 of *INFSYS Research Report*, pages 200–205. Technische Universität Wien, Austria, 2006.