On the Small-Scope Hypothesis for Testing Answer-Set Programs*

Johannes Oetsch, Michael Prischink, Jörg Pührer, Martin Schwengerer, and **Hans Tompits**
Institut für Informationssysteme 184/3
Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,prischink,puehrer,schwengerer,tompits}@kr.tuwien.ac.at

Abstract

In software testing, the *small-scope hypothesis* states that a high proportion of errors can be found by testing a program for all test inputs within some small scope. In this paper, we evaluate the small-scope hypothesis for answer-set programming (ASP). To this end, we follow work in traditional testing and base our evaluation on mutation analysis. In fact, we show that a rather limited scope is sufficient for testing ASP encodings from a representative set of benchmark problems. Our experimental evaluation facilitates effective methods for testing in ASP. Also, it gives some justification to analyse programs at the propositional level after grounding them over a small domain.

Introduction

Answer-set programming (ASP) has emerged as a successful formalism in knowledge representation and nonmonotonic reasoning. As a general problem solving paradigm, ASP facilitates means to declaratively specify solutions to a given problem and to use ASP solvers to compute respective models, referred to as the *answer sets*, of these specifications. Increasingly efficient ASP solver technology has allowed ASP to become a viable computational approach in many areas like semantic-web reasoning (Polleres 2005), systems biology (Grell, Schaub, and Selbig 2006), planning (Eiter et al. 2000), diagnosis (Eiter et al. 1999; Nogueira et al. 2001), configuration (Soininen and Niemelä 1999), multi-agent systems (Baral and Gelfond 2000), cladistics (Erdem, Lifschitz, and Ringe 2006; Brooks et al. 2007), super optimisation (Brain et al. 2006), and others.

It is an arguable 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 do sneak even into such high-level specifications. Thus, adequate means for validation and verification are clearly needed for ASP.

*This work was partially supported by the Austrian Science Fund (FWF) under grant P21698.

Though testing is the prevalent means to find errors in traditional software development, this subject has been addressed for ASP only recently (Janhunen et al. 2010; 2011; Febbraro et al. 2011). In particular, 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 (Jackson and Damon 1996), 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.

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 ASP is used). 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`:

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

ity 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, e.g., 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 paper. To this end, we follow Andoni et al. (2002) 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 (Marinov et al. 2003)). Their evaluation is based on *mutation analysis* (DeMillo, Lipton, and Sayward 1978) 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. (2002) for ASP. Our main contribution is twofold:

- We introduce a mutation model for ASP. Such a mutation model has its worth for itself beyond evaluating the small-scope hypothesis, 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, and we expect that such test inputs are also quite effective for testing in ASP.
- Based on mutation analysis, we evaluate the small-scope hypothesis using benchmark problems used for the third ASP competition (Calimeri et al. 2011). 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.

Background

We deal with logic programs that correspond to a subset of the input language of the state-of-the-art ASP grounder `gringo` (Gebser, Schaub, and Thiele 2007; Gebser et al. 2009). 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

$$l \,\#\texttt{count}\,\{\, l_1, \ldots, l_m \,\}\, u \quad \text{or}$$
$$l \,\mathrm{op}\, [\, l_1 = w_1, \ldots, l_m = w_m \,]\, u\,,$$

where $l_1, \ldots, l_m$ form a multiset of ordinary literals, each w_i is an integer weight assigned to l_i, $\mathrm{op} \in \{\#\texttt{sum}, \#\texttt{min}, \#\texttt{max}, \#\texttt{avg}\}$, 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 $\#\texttt{count}$ and $\#\texttt{sum}$ may be omitted. A *literal* is is either an atom or a default negated atom. Note that the `gringo` language also comprises the usual arithmetic operations as well as comparison predicates.

A *rule*, r, is a pair of form

$$\mathtt{a_0} \texttt{ :- } \mathtt{a_1}, \ldots, \mathtt{a_m}, \texttt{not } \mathtt{a_{m+1}}, \ldots, \texttt{not } \mathtt{a_n}\,,$$

where $\mathtt{a_0}, \ldots, \mathtt{a_n}$ are atoms. The atom $\mathtt{a_0}$ may be absent in r in which case r is called a *constraint*. If $\mathtt{a_0}$ is present but $n = 0$, then r is a *fact*. If r is neither a constraint nor a fact, then it is referred to as a *proper rule*. We call $\mathrm{B}^+(r) = \{a_1, \ldots, a_m\}$ the *positive body*, $\mathrm{B}^-(r) = \{a_{m+1}, \ldots, a_n\}$ the *negative body*, and $\mathrm{H}(r) = \{a_0\}$ the *head* of rule r. Intuitively, facts state what is known to be true. If all atoms in the positive body of a proper rule are known to be true and there is no evidence for any of the atoms in the negative body, then the head atom has to be true as well. Similarly, a constraint states that it must never be the case that all positive but none of the negative body atoms are jointly true.

A rule r is *safe* if all variables occurring in r also occur in the positive body of r. We assume in the remainder of the paper that all rules are safe. A *program* is a finite set of (safe) rules. As usual, 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.

An *interpretation*, I, is a set of ordinary atoms that does not include an atom and its strong negation. The semantics of the considered 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, specific interpretations, the *answer sets* of P, are defined for $grnd(P)$. Basically, they are defined corresponding to the *stable-model semantics* for normal logic programs (Gelfond and Lifschitz 1988) augmented with the *choice semantics* for aggregate atoms in rule heads (Ferraris and Lifschitz 2005; Simons, Niemelä, and Soininen 2002). We denote the collection of all answer sets of a program P by $\mathrm{AS}(P)$.

We next introduce some basic definitions related to testing in ASP which lift respective definitions for propositional programs from previous work (Janhunen et al. 2010) to the first-order level. For each program P, we assume two finite sets of predicate symbols, $\mathbb{I}_P$ and $\mathbb{O}_P$, which are the program's *input* and *output signature*, respectively.

Definition 1. *Let P be a program with input signature $\mathbb{I}_P$ and output signature $\mathbb{O}_P$. A* test input*, or simply* input*, of P is a finite set of ground atoms over $\mathbb{I}_P$. Given some input I of P, the* output *of P on I, $P[I]$, is defined as*

$$P[I] = \{S|_{\mathbb{O}_P} \mid S \in \mathrm{AS}(P \cup I)\},$$

where $S|_{\mathbb{O}_P}$ denotes the projection of S to the atoms in $\mathbb{O}_P$.

For testing, usually not all possible inputs over a program's input signature are of interest. Consider, for example, an encoding P for some graph problem similar to the one from the introduction. To be more specific, 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 P. Hence, the output of P is only required to match the expected output for inputs that are trees, or, more generally, that satisfy a program's precondition—we refer to such inputs as *admissible* (Oetsch et al. 2009; Janhunen et al. 2011).

Our evaluation approach is based on mutation analysis. Hence, we will use an ASP program itself as a test oracle to determine the expected output for any input. In particular, 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. Finally, we define the scope of an input as follows:

Definition 2. *Let P be a program. Then, an input of P has* scope n *if at most n different constant symbols occur in I.*

Prelude: A Key Observation

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 of 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 (Shmueli 1987), which is the following task: given two Horn programs P and Q such that $\mathbb{O}_P = \mathbb{O}_Q$, consisting of a single dedicated goal predicate, and $\mathbb{I}_P = \mathbb{I}_Q$, containing all predicate symbols that only occur in rule bodies of P or Q, determine whether $P[I] = Q[I]$ for any I over $\mathbb{I}_P$. Towards a contradiction, assume testing P positively with all inputs of 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 if $P[I] = Q[I]$ for any I with 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. □

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

The remainder of the paper is structured as follows: In the next section, we introduce a mutation model for ASP, providing the basis for our evaluation. Afterwards, we outline our experimental setup in more detail and present the results of our evaluation. The paper concludes with pointers for future work.

A Mutation Model for ASP

Mutation operations were introduced for many programming languages (Agrawal et al. 1989; Kim, Clark, and McDermid 1999; King and Offutt 1991; Offutt, Voas, and Payne 1996). In this section, 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 (Kim, Clark, and McDermid 1999), our selection is more based on programming experience as in the approach of Agrawal et al. (1989). Simple mutation operations for propositional answer-set programs have already been introduced in previous work (Janhunen et al. 2011).

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* (DeMillo, Lipton, and Sayward 1978). Also, all mutated programs have to be syntactically correct, otherwise they cannot be used for testing. Mutants which cause syntax errors are sometimes called *stillborn* (Offutt, Voas, and Payne 1996). 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 the latest ASP solver generation does 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

Table 1: Mutation operations for ASP.

		Example	
Name	Operation	Original Rules	Mutant
RDP	delete proper rule	`p(X) :- q(X). q(X) :- r(X).`	`p(X) :- q(X).`
RDC	delete constraint	`good(x-men). :- good(X),evil(X).`	`good(x-men).`
RDF	delete fact	`good(x-men). :- good(X),evil(X).`	`:- good(X), evil(X).`
LDB	delete body literal	`norm(X) :- p(X), not ab(X).`	`norm(X) :- p(X).`
LDH	delete head literal	`norm(X) :- p(X), not ab(X).`	`:- p(X),not ab(X).`
LAD	add default negation	`p :- q(X,Y),  t(Y).`	`p :- q(X,Y),  not t(Y).`
LRD	remove default negation	`p :- q(X,Y), not r(X).`	`p :- q(X,Y), r(X).`
LAS	add strong negation	`person(X) :- mutant(X).`	`person(X) :- -mutant(X).`
LRS	remove strong negation	`norm(X) :- -mutant(X).`	`norm(X) :- mutant(X).`
LRP	rename predicate	`r(X,Y):- e(X,Y). :- e(a,b).`	`r(X,Y):- e(X,Y). :- r(a,b).`
LRC	replace comparison relation	`:- succ(X,Y), Y > X.`	`:- succ(X,Y), Y >= X.`
ARO	replace arithmetic operator	`next(X,Y) :- r(X;Y),Y=X+1.`	`next(X,Y) :- r(X;Y),Y=X*1.`
ATV	twiddle variable domain	`:- s(X,Y), X==(Y+X)*2.`	`:- s(X,Y), X==((Y+1)+X)*2.`
ATA	twiddle aggregate bound	`1{guess(X)}N :- max(N).`	`1{guess(X)}(N-1) :- max(N).`
ATW	twiddle aggregate weight	`ok :- 2 [val(P,V)=V] 8.`	`ok :- 2 [val(P,V)=(V+1)] 8.`
TST	swap terms in literals	`less(X,Y) :- n(X,Y),X < Y.`	`less(Y,X) :- n(X,Y),X < Y.`
TVC	change variable to constant	`p(a;b). fail :- p(X).`	`p(a;b). fail :- p(a).`
TRV	rename variable	`first(X) :- rel(X,Y).`	`first(Y) :- rel(X,Y).`
TCV	change constant to variable	`ok :- exit(1,Y),grid(X,Y).`	`ok :- exit(X,Y), grid(X,Y).`
TRC	rename constant	`first(alpha). last(omega).`	`first(omega). last(omega).`

and requires a post-processing step where they are filtered out.

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. 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 unlikely that they semantically simulate each other.

Following Agrawal et al. (1989), 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 1 summarises 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., <, !=, and >, otherwise resulting mutants tend to be equivalent due to symmetries.

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

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

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.

Experimental Setup

To evaluate the small-scope hypothesis for ASP, we follow the approach of Andoni et al. (2002). In particular, we apply our mutation model on a set of representative benchmark instances taken from the third ASP solver competition (Calimeri et al. 2011). 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 Web.[2] These benchmarks are a representative cross section of challenging problems for both modelling in ASP as well as solving. In particular, we consider all problems in P and in NP for which reference encodings are published on the competition's Web page. In particular, 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. The ASP-Core and the ASP-RFC language can be regarded as simple language fragments common to most ASP solvers; the latter comprises aggregate expressions, common to most ASP solvers. A respective documentation document can be found at the Web page of the competition.

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 threat for retaining the validity of our experiments for other classes of ASP language dialects. In particular, we

- rewrite guesses expressed using disjunction into choice rules,
- replace <> with !=,
- rewrite aggregate expressions from the ASP-RFC syntax into `gringo` syntax, and
- rewrite 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, and
3. we exhaustively test each mutant for fixed scopes and assess the catching rates.

[2] https://www.mat.unical.it/aspcomp2011.

```
% 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 1: ASP encoding of GRAPHCOLOURING.

We exemplify our methods using the simple benchmark problem GRAPHCOLOURING. The encoding appears in Figure 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 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. In particular, 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 from 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 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` and `t` gives the scope n.[3]

In Step 2, each mutant is generated by applying precisely one randomly selected mutation operation from Table 1 to the benchmark program. Stillborn mutants are filtered out in a post-processing step, where we check whether a mutant

[3] Technically, the scope is given by the maximum of `s` and `t`. However, we count both the integer ranges defined by `s` and `t` 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 2: Input generator for GRAPHCOLOURING.

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 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 takes almost no time and a mutant cannot be grounded within seconds, it's sensible to assume that grounding will not terminate at all in practice.

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 with scope n, we need to check if $P' \cup IG[P,n]$ and $P \cup IG[P,n]$, or equally $grnd(P' \cup IG[P,n])$ 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) (Oetsch, Tompits, and Woltran 2007). As an aside, we note that Offutt, Voas, and Payne (1996) considered also the notion of weak equivalence for mutants of nondeterministic 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) (Oetsch, Tompits, and Woltran 2007) 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 nondeterministic mutant that is trivially weakly equivalent with the program under test. Also, the correctness notion for nondeterministic programs by Offutt, Voas, and Payne (1996) 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, viz. *EVA* (*enough visible atoms*), that allows to treat PQEPs as special case of *modular equivalence* (Oikarinen and Janhunen 2009; Janhunen and Oikarinen 2004). 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 coNP complexity of deciding modular equivalence allows a reduction approach to ASP itself while deciding PQEPs is hard for a problem class from the third level of the polynomial hierarchy. A respective reduction approach from modular equivalence to ASP is realised by the tool `lpeq` (Oikarinen and Janhunen 2009), 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 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 (Oetsch et al. 2009). In that case study, the correspondence checking tool `cc`$\top$ (Oetsch et al. 2007) 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 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* (Gomes, Sabharwal, and Selman 2006) from model counting in SAT for ASP which is based on *parity constraints*. A parity constraint on a set S of atoms expresses that an even (or odd) number of atoms from S have to be true. Basically, we add s parity constraints on the output atoms to an input generator and test whether it yields some answer sets. 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.

Results of the Evaluation

We classified our used benchmark programs according to the hardness of the respective encoded problem. To wit, Table 2 summarises the results for the benchmark problems that are solvable in polynomial time and Tables 3 and 4 contain the results for the NP-hard problems. While the benchmarks in Table 4 contain at least one recursive predicate, the problem encodings in Table 3 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

Table 2: Evaluation results for benchmark instances in P.

Problem Name	No. of Mutants	Scope	No. of Inputs	Time (Sec.)	Catching Rate
REACHABILITY	31	1	2	0.2	0.25
		2	72	0.2	0.96
		3	4770	0.3	1.00
GRAMMAR-BASED INFORMATION EXTRACTION	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
PARTNERUNITS POLYNOMIAL	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

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 (Lin and Zhao 2004; Lifschitz and Razborov 2006). 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 2, 3, and 4, 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.

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%.

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 (Shmueli 1987). A hint to know when (almost) all non-equivalent mutants are caught is when reaching a fixedpoint 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

Table 3: Evaluation results for benchmark instances in NP with tight encoding.

Problem Name	No. of Mutants	Scope	No. of Inputs	Time per Mutant (Sec.)	Catching Rate
FASTFOOD OPTIMALITYCHECK	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
DISJUNCTIVE SCHEDULING	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
MULTICONTEXT SYSTEMQUERYING	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
WEIGHT-ASSIGNMENTTREE	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

`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; a more general methodology regarding this issue shall be addressed in future work. 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 GRAMMARBASEDINFORMATIONEXTRACTION, 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., GRAMMARBASEDINFORMATIONEXTRACTION 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 if no reference program is available, one may use other test input selection strategies like random testing or structure-based testing (Janhunen et al. 2010; 2011).

Table 4: Evaluation results for benchmark instances in NP with non-tight encoding.

Problem Name	No. of Mutants	Scope	No. of Inputs	Time per Mutant (Sec.)	Catching 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
		8	16777216	352.0	0.94
LABYRINTH	293	6	$4.3 \cdot 10^{12\dagger}$	3.4	0.06
		7	$8.7 \cdot 10^{12\dagger}$	4.0	0.07
		8	$4.7 \cdot 10^{21\dagger}$	23.9	0.94
NUMBERLINK	222	4	24	2.7	0.36
		5	56	3.2	0.36
		6	16568	20.9	0.63
		7	99576	30.4	0.63
		8	92751096	517.2	0.67
MAGICSQUARESETS	270	5	332	3.0	0.97
		9	1048908	5.9	1.00
MAZEGENERATION	292	3	468840	8.9	0.76
		4	42063108	16.0	0.82
		5	$2.1 \cdot 10^{12\dagger}$	41.8	0.95

Conclusion and Future Work

In this paper, we empirically evaluated the small-scope hypothesis for ASP. Our experiments are based on mutation analysis, a respective mutation model for ASP was introduced for this purpose. Indeed, it showed that inputs from a small scope, i.e., inputs defined over a relatively small number of constant symbols, are sufficient to catch all mutants of a program under test. The small-scope hypothesis is supported by concrete scopes that could be obtained for different kinds of encodings. 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 work also shows 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 upcoming 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.

For future work, we plan, on the one hand, to refine our mutation model by some further operations to take unrestricted function symbols, disjunction, weak constraints, minimise statements, queries, other forms of aggregates, etc., into account. Thereby, we would extend our mutation model and make it applicable to other solver dialects like that of DLV. On the other 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* (Offutt, Rothermel, and Zapf 1993).

Also, the relation between mutation operations and code coverage in ASP needs to be studied. It is known that mutation analysis subsumes structure-based testing (Offutt and Voas 1996). Hence, for any code coverage condition, we can introduce respective mutation operations such that mutation adequate input collections satisfy the considered coverage condition. Notions of code coverage for ASP have been introduced in previous work (Janhunen et al. 2010). We believe that mutation analysis could be used to find good criteria to improve our structure-based coverage notions defined there.

We also want to improve the theoretic basis of the small-scope hypothesis by identifying sufficient conditions that allow to restrict the input domain such that equivalence between a program under test and a mutant on the restricted domain implies equivalence on the unrestricted infinite domain. Although this is not possible in general (cf. Theorem 1), we conjecture that this can be shown at least for tight programs (note that the majority of NP benchmarks in our experiments are tight).

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. Related approaches for test input generation, like `Korat` (Boyapati, Khurshid, and Marinov 2002), avoid such isomorphic inputs when computing test inputs. To extend ASP solvers in a way that isomorphic answer sets are avoided during search without the need to change a program encoding seems to be a promising direction for future work as well.

References

Agrawal, H.; Demillo, R.; Hathaway, R.; Hsu, W.; Hsu, W.; Krauser, E.; Martin, R. J.; Mathur, A.; and Spafford, E. 1989. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Centre.

Andoni, A.; Daniliuc, D.; Khurshid, S.; and Marinov, D. 2002. Evaluating the "small scope hypothesis". Unpublished draft, available at http://sdg.lcs.mit.edu/pubs/2002/SSH.pdf.

Baral, C., and Gelfond, M. 2000. Reasoning agents in dynamic domains. In *Logic-based Artificial Intelligence*, 257–279. Kluwer Academic Publishers.

Boyapati, C.; Khurshid, S.; and Marinov, D. 2002. Korat: Automated testing based on Java predicates. *SIGSOFT Software Engineering Notes* 27:123–133.

Brain, M.; Crick, T.; De Vos, M.; and Fitch, J. 2006. 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.

Brooks, D. R.; Erdem, E.; Erdogan, S. T.; Minett, J. W.; and Ringe, D. 2007. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning* 39(4):471–511.

Calimeri, F.; Ianni, G.; Ricca, F.; Alviano, M.; Bria, A.; Catalano, G.; Cozza, S.; Faber, W.; Febbraro, O.; Leone, N.; Manna, M.; Martello, A.; Panetta, C.; Perri, S.; Reale, K.; Santoro, M. C.; Sirianni, M.; Terracina, G.; and Veltri, P. 2011. 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*, 388–403. Springer.

DeMillo, R. A.; Lipton, R. J.; and Sayward, F. G. 1978. Hints on test data selection help for the practicing programmer. *IEEE Computer* 11(4):34–41.

Eiter, T.; Faber, W.; Leone, N.; and Pfeifer, G. 1999. The diagnosis frontend of the DLV system. *AI Communications* 12(1-2):99–111.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2000. Planning under incomplete knowledge. In *Proceedings of the 1st International Conference on Computational Logic* (*CL 2000*), volume 1861 of *Lecture Notes in Computer Science*, 807–821. Springer.

Erdem, E.; Lifschitz, V.; and Ringe, D. 2006. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming* 6(5):539–558.

Febbraro, O.; Leone, N.; Reale, K.; and Ricca, F. 2011. Unit testing in ASPIDE. In *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (*INAP 2011*) *and the 25th Workshop on Logic Programming* (*WLP 2011*)*, INFSYS RESEARCH REPORT 1843-11-06*, 165–176.

Ferraris, P., and Lifschitz, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming* 5(1-2):45–74.

Gebser, M.; Kaminski, R.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2009. 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*, 502–508. Springer.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. 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*, 266–271. Springer.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th Logic Programming Symposium*, 1070–1080.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2006. Model counting: A new strategy for obtaining good bounds. In *In Proceedings of the 21st National Conference on Artificial Intelligence* (*AAAI 2006*), 51–61. AAAI Press.

Grell, S.; Schaub, T.; and Selbig, J. 2006. 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*, 285–299. Springer.

Jackson, D., and Damon, C. 1996. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering* 22(7):484–495.

Janhunen, T., and Oikarinen, E. 2004. 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*, 336–340. Springer.

Janhunen, T.; Niemelä, I.; Oetsch, J.; Pührer, J.; and Tompits, H. 2010. 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*, 951–956. IOS Press.

Janhunen, T.; Niemelä, I.; Oetsch, J.; Pührer, J.; and Tompits, H. 2011. 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*, 242–247. Springer.

Kim, S.; Clark, J. A.; and McDermid, J. A. 1999. 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*).

King, K. N., and Offutt, A. J. 1991. A Fortran language system for mutation-based software testing. *Software—Practice and Experience* 21(7):685–718.

Lifschitz, V., and Razborov, A. A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic* 7(2):261–268.

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

Marinov, D.; Andoni, A.; Daniliuc, D.; Khurshid, S.; and Rinard, M. 2003. An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory Report MIT-LCS-TR-921.

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. 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*, 169–183. Springer.

Oetsch, J.; Seidl, M.; Tompits, H.; and Woltran, S. 2007. Testing relativised uniform equivalence under answer-set projection in the system ccT. 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*, 241–246. Springer.

Oetsch, J.; Seidl, M.; Tompits, H.; and Woltran, S. 2009. ccT 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*, 382–395. Springer.

Oetsch, J.; Tompits, H.; and Woltran, S. 2007. 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*), 458–464. AAAI Press.

Offutt, A. J., and Voas, J. M. 1996. Subsumption of condition coverage techniques by mutation testing. Technical Report ISSE-TR-96-01, Department of Information and Software Engineering, George Mason University, 4400 University Drive MS 4A5, Fairfax, VA 22030-4444 USA.

Offutt, A. J.; Rothermel, G.; and Zapf, C. 1993. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering*, 100–107. Baltimore, MY: IEEE Computer Society Press.

Offutt, J.; Voas, J.; and Payne, J. 1996. Mutation operators for ada. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University.

Oikarinen, E., and Janhunen, T. 2009. A translation-based approach to the verification of modular equivalence. *Journal of Logic and Computation* 19(4):591–613.

Polleres, A. 2005. Semantic Web languages and Semantic Web services as application areas for answer set programming. In *Nonmonotonic Reasoning, Answer Set Programming and Constraints*, number 05171 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Shmueli, O. 1987. Decidability and expressiveness of logic queries. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 237–249.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1–2):181–234.

Soininen, T., and Niemelä, I. 1999. 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*, 305–319. Springer.