Institute of Computer Science
Knowledge Processing and Information Systems

Titel

Bachelorthesis

Jakob Westphal

28. Juli 2022

Erstgutachter: Prof. Dr. Torsten Schaub
Zweitgutachter: Tobias Stolzmann

Abstract

Abstract in english This is my bachelorthesis

Zusammenfassung

Zusammenfassung auf Deutsch

Acknowledgments

Danksagung!

Contents

1 Introduction

- Answer Set Programming is of growing importance in both academic and industry work (source?) and clingo is a popular solver for this

- Workflow Tools that make working with such programs easier and more comfortable barely exist. Research into these topics is only now getting more traction (source?)

- Testing is a very important part of a conventional software design approach.(source?)

- The topic of code coverage and it's use for conventional programming languages has been shown (source?)

- With this work I want to build on the previous work done on code coverage in answer set programming in this paper by Janhunen et al. [Jan+10]

- To this end I developped a way to efficiently implement the coverage metrics defined in the paper.

- My implementation allows me to compute coverage using answer set programming. It also extends the given metrics to function with almost all existing language construct instead of just for propositional programs.

2 Preliminaries

2.1 Answer Set Programming

- Basic introduction to asp giving all the relevant definitions (with examples + sources (for all/some?))
  - rule, head, (positive/negative)body
  - interpretation that satisfies (positive/negative)body, rule, program
  - Def(a), SuppR(P,I)
  - answer sets, brave/cautious consequence
  - positive atom dependency graph, loops, strongly connected components
  -> adjust these so they work for variables etc. or do this later? (is this needed?/ is it a big adjustment?)

2.2 Testing in Answer Set Programming

- What is input and output of a program?
  - what is a testcase and a testsuite
  - exhaustive test suite for P -> all possible testcases
  - maybe Specification -> the correct (expected) output for every input, what does it mean for a program to "pass/be compliant with" a testcase, when is a program "correct" with respect to a specification (not actually needed for coverage as coverage does not care about specification!)

3 Coverage metrics

- Analogous to concepts of coverage in other (conventional) programming languages (source) I introduce path-like and branch-like coverage metrics according to Janhunen et al. [Jan+10]

3.1 Coverage functions

- general definition of coverage functions (maybe additional source? / compare to what paper did)
  - talk about the difference between all objects and coverable objects
  - trivial/clairvoyant coverage functions maybe not important to discuss?

3.2 Path-like coverage

- general introduction to path like coverage in conventional programming languages (source)
  - use the controlflow graph and cover every possible path through this graph
  - generally the most "complete" coverage metric
  -> these are generally very computationally expensive (exponentially many possible paths)

3.2.1 Program coverage

- Analogous to the conventional path-like coverage I define program coverage
  - Definition + example
  - show that total program coverage means all possible answer sets get produced by the testsuite
  - talk about the problems here? (complexity + not possible for programs with variables as it is necessary to enumerate all possible inputs to find maximum coverage)

3.3 Branch-like coverage

- general introduction to branch-like coverage in conventional programming languages (source)
  - use the controlflow graph and cover every possible branch through this graph
  - less complete but easier to compute, still very potent(?) (source)
  - there are different types of branch like coverage even in conventional programming languages (source), the same is true here

3.3.1 Rule coverage

- Definition + example
  - similar to program coverage in some ways but less complex!
  - some rules may sometimes (or always) not be coverable -> examples -> ties back to beginning of the chapter / thats why coverage is defined on coverable objects
  (- total program coverage implies total rule coverage -> not so relevant but maybe interesting to mention?)

Constraint coverage

(is this an extra section or should this be in the rule coverage section?)
  - constraints are a special type of rule and have to be handled slightly differently because when the body of a constraint is true (=normal rule coverage) this will imply false and therefore not create an answer set / create an unsatisfiable solve call -> we cant check constraints the same way we check other rules
  - solution: (following the suggestion in the paper by Janhunen et al. [Jan+11]) remove the constraint from the program in order to check for its covereage!
  - Definition + examples
  (- maybe reminder that these coverage metrics dont really care about what the output of the program is and whether its according to the specification, therefore removing constraints is okay even though it might completely destroy the functionality of the program)

3.3.2 Loop coverage

- loops play an important role in ASP as seen in (source). Therefore constructing a coverage metric that focuses on them makes sense
  - Definition + example

- generally number of loops in a program in exponential in the number of rules -> expensive to compute! -> introduce 2 more coverage metrics that approximate loop coverage! (one for minimal (singleton) loops and one for maximal loops (strongyl connected components))

(- no real relation to rule coverage (neither implies the other)

- total program coverage implies total loop coverage -> not so relevant but maybe interesting to mention?)

3.3.3 Definition coverage

- 2 ways to introduce definition coverage:

- as a coverage metric for singleton loops (minimal loops) and therefore a special case of loop coverage

- as a representation of the disjunctions in the program (if an atom "a" is defined in multiple rules you could rewrite this as a if B1 v B2 v ...) -> this coverage metric covers these implicit disjunctions -> discuss both but in which order?

- Definition + example

(- this is different to rule coverage! -> total positive rule coverage implies total positive definition coverage, not the other way around and no connection for negative coverage!

- total loop coverage implies total definition coverage

- total program coverage implies total definition coverage)

3.3.4 Component coverage

- strongly connected components are the maximal loops of the program, therefore this is an approximation of loop coverage

- Definition + example

- Definition of negative coverage is different from loop coverage!

- because of this different definition positive/negative component coverage of a specific component implies positive/negative loop coverage for all subset loops of the component (would not be the case for negative coverage if definition is different)

(- total positive loop coverage implies total positive component coverage, however this is not true for negative coverage because of the different definition (see above)

- total program coverage implies total component coverage)

4 Implementation

(maybe include a short history of what i tried but didnt work? -> meta programming with reify output)

4.1 General approach

- the goal is to compute the coverage using ASP

-> introduce labels for each coverage metric -> add them to the program in a specific way (based on the coverage metric) -> solve normally using clingo -> if the label is in the answer set, the corresponding object is covered

- I only add new rules, dont change or take away existing ones (except constraints -> see above) and these new rules only produce labels not predicates that are part of the original program -> the resulting program is still equivalent! (maybe proof?)

- with this method, to find total positive and negative coverage it is not necessary to look at every model in the solve call. Instead use one solve call with brave consequences and one with cautious consequences! -> more efficient than looking at every model!

- labels can also be used to add the testcases to the program -> this way coverage for all testcases can be computed in the same solve call instead of needing one call per testcase

-> this is done by: adding a choice rule $\{\_i0; ...; \_in\}$. for n testcases in the testsuite, adding rules $a_i :- i_j$. for every atom i in the testcase j. (- why does this work? - why does this not change the program?)

- maximum coverage needs to be computed to get accurate coverage numbers!

-> done by calculating the coverage normally but instead of testcases adding a rule $\{a_0; ...; a_n\}$. where $a_0 ... a_n$ are all possible inputatoms (see definition input/output) -> these have to be specified in advance by the user!

4.1.1 Rule coverage

- $\_ri$ label for every rule i

- add new rule to program for each label: $\{\_ri\}$ is the head, the body is identical to the body of the rule i

- example

- if the body of rule i is true in some answer set (aka $X| = B(r_i)$) then rule i is positively covered and $\_ri$ will appear in the answer set

-> $\_ri$ in answer set <=> rule i is positively covered

-> $\_ri$ not in answer set <=> rule i is negatively covered (proof for these?)

4.1.2 Definition coverage

- $\_di$ label for every atom i

- if atom i is definable (it appears in the head of a rule, aka $Def(a_i) \neq \emptyset$) add new rule to program: $\_di :- \_rj.$ for every rule j that defines atom i (every rule j that has atom i in its head, aka every $r_j \in Def(a_i)$)

- example

- if one of the rules is covered (its body is true) then atom i is covered and $\_di$ will appear in the answer set

-> $\_di$ in answer set <=> atom i is positively covered

-> $\_di$ not in answer set <=> atom i is negatively covered (proof for these?)

4.1.3 Loop coverage

- $\_li$ label for every loop i

- first need to find all the loops in the program! -> build positive atom dependency graph, find sccs and then find subsets of sccs that are loops

- for each loop i that consists of atoms a_m to a_n add new rule to program: $\_li :- \_dm, ..., \_dn.$

- if all the atoms a_m to a_n that constitute the loop i are defined (aka definition covered), then all the $\_dm$ to $\_dn$ are true, then loop i is covered and $\_li$ will be in the answer set.

-> $\_li$ in answer set <=> loop i is positively covered

- if any of the atoms a_m to a_n are not defined the loop is negatively covered and $\_li$ will not be in the answer set

-> $\_li$ not in answer set <=> loop i is negatively covered

4.1.4 Component coverage

- $\_si$ label for every strongly connected component i

- find sccs same way as loops

- construct new rules same way as for loops: $\_si : -\_dm, ..., \_dn.$

-> $\_si$ in answer set <=> scc i is positively covered

- HOWEVER! $\_si$ not in answer set -/> scc i is negatively covered! (see definition of component coverage)

-> add additional rules to program: $\_nsi : -not\_dm, ..., not\_dn.$

- if NONE of the atoms a_m to a_n are defined (aka definition covered), scc i is negatively covered and $\_nsi$ will be in the anser set

-> $\_nsi$ in answer set <=> scc i is negatively covered

4.1.5 Program coverage

- use the $\_ri$ labels from rule coverage, no new labels needed!

- a subprogram $P' \subseteq P$ is covered if exactly all rules contained in P' are covered and no other rules are covered

-> each answer set covers exactly one subprogram -> it is necessary to look at every answer set instead of just brave/cautious like with the other coverage metrics -> has to be computed seperately from the other coverage metrics!

- for $P = \{r_1, ..., r_n\}$, $\{\_rx, ..., \_ry\}$ are the rule labels in an answer set <=> $P' = \{r_x, ..., r_y\}$ is covered

4.2 Implementing coverage for further program classes

- the given definitions of the coverage metrics are only for propositional programs but this is not very practical as most programs are more complex than that. They contain many complex language constructs supported by ASP/clingo

-> to make this coverage check actually usable these metrics have to be extended to work for all these constructs

- the label approach allows me to easily apply these coverage metrics to further program classes with very little adjustments!

- go through all the additional constructs one by one, explain how they should work, why they do work like that or not? -> table

- Big difference: maximal coverage can not be computed as this requires listing all possible inputs. This is not possible with variables as there can be infinitely many -> give coverage as covered/total existing (!!! is this even a correct coverage function???) (!!! possible extension: allow user to specify domain for each variable -> if domain is not infinite then computing max cov is possible !!!)

-> thats all the difference!?!

5 Outlook

- As mentioned in the previous section, theory atoms and constraint terms are currently not considered when calculating definition, loop and component coverage -> should be an easy extension!

- rebuilding the app to work with the ClingoApp could make working with external atoms, constants and different program parts possible or easier (by allowing interaction with the clingo solver through the command line)

- in this work, complexity and efficiency are not priorities, therefore the current program is not very optimized!

- for this prototype I tried to take the existing definitions of coverage in ASP and extend them to more complex programs with as little changes to the definitions as possible. This might not be the best way! Some of the definitions might need more changes:

- Definition coverage and choice rules: if an atom is only defined in a choice rule (ex. $\{a\}$.) it will only be considered positively covered, not negatively covered, if the body is true, even though the atom will be false in some answer sets (therefore acting as if it was negatively covered -> the "path" where the atom is negatively covered is executed, but the atom is not considered negatively covered)

- loop/component coverage: a loop is considered covered, if all atoms contained in the loop are definition covered. Due to the nature of definition coverage it is however possible for an atom to be covered at multiple places! -> it is thus possible to cover all atoms in a loop without ever "executing" the loop -> this can lead to problems that are caused by a loop not being discovered by a testcase, even though that testcase has total loop coverage! (example!)

- simply checking coverage for a given testsuite is only one use case for coverage metrics! They can also be used to automatically generate testcases that are meant to catch a maximum amount of errors. The idea is explored in [Jan+11]. This can certainly also be done with my implementation.

- these coverage metrics have not really been tested! In the paper [Jan+11] only rule and definition coverage have been tested for their practicality in a "real world" scenario. It is unknown how effective loop and component coverage are. -> This needs

testing! During these tests potential changes to the definitions could also be evaluated.

- also in the line of testing the coverage metrics it would be interesting to see how well definition and component coverage approximate loop coverage and whether program coverage does actually give the best results given that it is the most "complete" metric -> maybe figure out a guideline on which metrics to use when. The current setup where mixing any metrics is possible does not make much sense (as for example definition coverage is fully contained in loop coverage)!

- based on the idea of adding labels to the existing program in order to compute coverage with ASP it should be relatively easy to realise any changes or even add potentially new coverage metrics

6 Conclusion

- I managed to implement the coverage metrics defined by Janhunen et al. [Jan+10] in a way that makes it possible to compute the coverage of an entire testsuite with the help of just 2 clingo solve calls

- I also extended the coverage metrics to cover almost all existing language constructs in ASP/clingo, making them more "real world" applicable

- with this I laid the ground work to one day implement coverage checks and maybe coverage-based testgeneration in a full unit testing api for ASP programs

- the simple nature of my approach should make it easy to extend to program with new or improved coverage metrics

7 Test

1. Bla.
   (Bla.)

2. Bla. (Bla.)

Chapter 1 Referenz zu Kapitel. Section 7.1 Referenz zu Unterkapitel. Figure 7.1 Referenz zu Bild. Listing 7.1 Referenz zu Listing Janhunen et al. [Jan+10] Zitat mit Namen der Autoren [Jan+10] Zitat nur mit Abkürzung Answer Set Programming (ASP) Link zu Abkürzungen *symbol grounding problem* Randkommentar [1] Fussnote Symbol

7.1 Perzepte und Symbole

Bla.

$$\text{match}(\sigma, \gamma) \Leftrightarrow \forall p \in \sigma \; \exists \phi \in \text{feat}(\gamma) : \; g(p, \phi, \gamma(\phi))$$

Proof. Zu jeder Teilmenge $M \subseteq A = \{a, b, c\}$ ist $P^M = P$. Die Teilmengen $\emptyset$, $\{a\}$, $\{c\}$, $\{a, b\}$ und $\{b, c\}$ sind keine Modelle von P^M. $\{a, c\}$, $\{a, b, c\}$ und $\{b\}$ sind Modelle von P^M. $\{a, b, c\}$ ist kein minimales Modell von P^M, da $\{b\} \subseteq \{a, b, c\}$. Da $\{a, c\} \nsubseteq \{b\}$ und $\{b\} \nsubseteq \{a, c\}$, sind beide Modelle minimal und damit stabile Modelle von P. □

[1] Bla

M	P_1^M	$Cn(P_1^M)$
$\emptyset$	$\{a \leftarrow a, b \leftarrow\}$	$\{b\}$
$\{a\}$	$\{a \leftarrow a\}$	$\emptyset$
$\{b\}$	$\{a \leftarrow a, b \leftarrow\}$	$\{b\}$
$\{a, b\}$	$\{a \leftarrow a\}$	$\emptyset$

Table 7.1: $P_1 = \{a \leftarrow a, b \leftarrow naf a\}$ hat ein stabiles Modell $\{b\}$.

Figure 7.1: Ein Kamerabild mit eingezeichneten Perzepten.

```
symbol(cup_1; cup_2; cup_3; spoon; diningtable).

is_on(
    cup_1, diningtable;
    cup_2, diningtable;
    cup_3, diningtable
).

is_inside_of(spoon, cup_3).

contains(
    cup_1, coffee;
    cup_2, coffee;
    cup_3, hot_chocolate
).
```

Listing 7.1: Eine symbolische Beschreibung der Objekte in bla.

`#show p(X,Y) : q(X).`
Test für `#show p(X)` in einer Zeile.

$$
\begin{aligned}
X &= \{\text{cup}_1, \text{cup}_2\} \\
\Pi &= \{\pi_1, \pi_2, \pi_3\} \\
\Phi &= \{\text{coffee}, \text{tea}, \text{hot}, \text{cold}\} \\
T &= \{t_1, t_2\} \\
\beta(\text{cup}_1, t_1) &= \{\text{coffee}\} \\
\beta(\text{cup}_2, t_1) &= \emptyset \\
\beta(\pi_1, t_1) &= \{\text{coffee}\} \\
\beta(\pi_2, t_1) &= \{\text{tea}, \text{cold}\} \\
\beta(\pi_3, t_2) &= \{\text{tea}\}
\end{aligned}
$$

Abbreviations

ASP Answer Set Programming

List of Figures

List of Tables

Listings

Bibliography

[Jan+10] Tomi Janhunen et al. “On testing answer-set programs”. In: *ECAI 2010*. IOS Press, 2010, pp. 951–956 (cit. on pp. 1, 5, 15, 17).

[Jan+11] Tomi Janhunen et al. “Random vs. Structure-Based Testing of Answer-Set Programs: An Experimental Comparison”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by James P. Delgrande and Wolfgang Faber. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 242–247. ISBN: 978-3-642-20895-9 (cit. on pp. 6, 13).

Declaration

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

Die „Richtlinie zur Sicherung guter wissenschaftlicher Praxis für Studierende an der Universität Potsdam (Plagiatsrichtlinie) - Vom 20. Oktober 2010“, im Internet unter http://uni-potsdam.de/ambek/ambek2011/1/Seite7.pdf, habe ich zur Kenntnis genommen.

Berlin, 28. Juli 2022

Jakob Westphal