A Short Tutorial on s(CASP), a Goal-directed Execution of Constraint Answer Set Programs

Joaquín Arias[1], Gopal Gupta[2] and Manuel Carro[3,4]

[1]CETINIA, Universidad Rey Juan Carlos, Madrid, Spain
[2]University of Texas at Dallas, Richardson, USA
[3]IMDEA Software Institute, Madrid, Spain
[4]Universidad Politécnica de Madrid, Spain

Abstract

This paper presents a short tutorial on s(CASP), highlighting some of the novel aspects and the reasons for its design. The most important aspect of s(CASP) is its goal-directed top-down execution model that implements Constraints Answer Set Programming. The execution strategy of s(CAPS) avoids the grounding phase, present in most ASP systems, and can constraint variables that, as in CLP, are kept during the execution and in the answer sets. Additionally, s(CASP) generates a human-understandable justifications (in natural language) of the resulting answer sets. s(CASP) is implemented in Prolog (currently available for Ciao and SWI Prolog) and has been used in several applications, including medical advisors, avionic, legal reasoner, XAI, and natural language processing.

Keywords

Answer Set Programming, Constraint, Goal-directed, s(CASP), Tutorial

1. Introduction

s(CASP) [1] is a novel non-monotonic reasoner, developed by Joaquín Arias in collaboration with IMDEA Software Institute and the University of Texas at Dallas. It is a re-implementation of s(ASP) [2] by Kyle Marple, extended with constraints. The main contribution of s(CASP) is its ability to evaluate Constraint Answer Set Programs without a grounding phase, either before or during execution. s(CASP) supports predicates and thus retains logical variables (and constraints) both during the execution and in the answer sets. The operational semantics of s(CASP) relies on backward chaining, which is intuitive to follow and lends itself to generating explanations that can be translated into natural language [3]. As the execution of an s(CASP) program returns partial stable models, that are the relevant subsets of the stable models [4], i.e., include only the (negated) literals necessary to support the initial query. To the best of our knowledge, s(CASP) is the only system that exhibits the property of relevance [5].

s(CASP) has been already applied in relevant fields mainly related to the representation of commonsense reasoning:

GDE'21: ICLP'21 (Virtual) Workshop on Goal-directed Execution of Answer Set Programs, September 20, 2021
joaquin.arias@urjc.es (J. Arias); gupta@utdallas.edu (G. Gupta); manuel.carro@imdea.org (M. Carro)
http://www.ia.urjc.es/GIA/joaquin-arias/ (J. Arias); https://personal.utdallas.edu/~gupta/ (G. Gupta)
0000-0003-4148-311X (J. Arias); 0000-0001-9727-0362 (G. Gupta); 0000-0001-5199-3135 (M. Carro)



CEUR Workshop Proceedings (CEUR-WS.org)

- An automated reasoner that uses Event Calculus (EC) [6], available at http://bit.ly/EventCalculus. The expressiveness of s(CASP) allows deductive reasoning tasks in domains featuring constraints involving dense time and fluents with continuous properties. It is being used to model real-world avionics systems, to verify (timed) properties as well as to identify gaps with respect to system requirements [7].
- s(CASP) justification framework has been used to bring Explainable Artificial Intelligence (XAI) principles to rule-based systems capturing expert knowledge [8, 3], ILP systems that generate ASP programs [9], and concurrent imperative programs based on behavioral, observable specifications [10].
- Two natural language understanding systems [11]: SQuARE, a Semantic-based Question Answering and Reasoning Engine, and StaCACK, Stateful Conversational Agent using Commonsense Knowledge. They use the s(CASP) engine to "truly understand" and perform reasoning while generating a natural language explanation for their responses. Building on these systems Kinjal, leader of one of the nine teams selected to participate in Amazon Alexa Socialbot Grand Challenge 4[1], is developing a conversational AI chatbot.
- Jason Morris' team[2] has developed an expert system at the SMU Centre for Computational Law at Singapore. They have coded rule 34 of Singapore's Legal Profession. The front-end of the system is a web interview that will collect information from a user, run the query on s(CASP), and display the results with a corresponding explanation.
- s(LAW), an administrative and judicial discretion reasoner[12], which allows modeling legal rules involving ambiguity and infers conclusion, providing (natural language)justifications based on them.

In addition, in January 2021, during the HackReason[3] at UT Dallas, 17 teams developed applications that rely on simulating human-style commonsense reasoning using s(CASP).

This tutoral provides a brief description of s(CASP). Section 2 describes the installation process. Section 3 presents the language supported by s(CASP). Section 4 provides some ticks to tune the size, and/or the number of resulting answer sets and to adjust the constraint's accuracy. Section 5 gives a brief description of the available flags to generate justifications and/or encoding (in natural language) with different levels of detail. In Section 6 we present the Dynamic Consistency Check (DCC), still a work in progress, and how can be activated for testing. Finally, in Section 7 we propose some ideas for improving s(CASP).

2. s(CASP) Installation on Ciao or SWI-Prolog

As we mentioned before, s(CASP) is currently available to be executed under Ciao http://ciao-lang.org/, or SWI-Prolog https://www.swi-prolog.org/. The s(CASP) source code for Ciao is available at https://gitlab.software.imdea.org/ciao-lang/sCASP and for SWI-Prolog, thanks to Jan Wielemaker, at https://github.com/JanWielemaker/sCASP.

[1] https://cs.utdallas.edu/computer-scientists-enhance-alexas-small-talk-skills/
[2] https://github.com/smucclaw/r34_sCASP
[3] https://hackreason.aisutd.org/

Installation on Ciao Ciao is a programming language that is builds from a logic-based simple kernel and is designed to be extensible and modular. The system implements some advanced features such as separate and incremental compilation, global program analysis and static debugging and optimization (via source to source program transformation, CiaoPP preprocessor), a build automation system, documentation generator, debugger, and (Emacs-based) development environment.

The installation of Ciao can be done using an interactive assistant from the command line in three steps: first, we run the interactive assistant [4], then, we activate the added Ciao path, and finally, we install s(CASP) as a bundle:

```
curl https://ciao-lang.org/boot -sSfL | sh
source ~/.washrag
ciao get gitlab.software.imdea.org/ciao-lang/sCASP
```

When the installation succeeds, a folder with the source code and s(CASP) examples is created in the ciao workspace (the default path is `~/.ciao/sCASP/`). The `--help_all` flag generates the list of all available flags [5] to consult and/or adapt s(CASP) behavior.

Installation on SWI-Prolog SWI-Prolog offers a comprehensive free Prolog environment. Since its start in 1987, SWI-Prolog development has been driven by the needs of real-world applications. SWI-Prolog is widely used in research and education as well as commercial applications. Additionally, it offers SWISH (https://swish.swi-prolog.org/), an online environment for teaching and exchanging ideas. Therefore, thanks to Jan Wielemaker, we can not only generate a standalone file, but also we have the opportunity to use s(CASP) directly in SWISH.

To generate the standalone file we have to use the latest development release version of SWI-Prolog available at https://www.swi-prolog.org/download/devel – there are already compiled SWI-Prolog binaries for Linux, Windows, and MacOS. Then, to generate the executable file of s(CASP), we just need to run the `make` file available in the corresponding s(CASP) repository for SWI-Prolog (https://github.com/JanWielemaker/sCASP).

3. Getting started

s(CASP) extends the expressiveness of Answer Set Programming systems by featuring predicates, constraints among non-ground variables, uninterpreted functions, and, most importantly, a top-down, query-driven execution strategy. These features make it possible to return answers with non-ground variables (possibly including constraints among them) and to compute partial models by returning only the fragment of a stable model that is necessary to support the answer to a given query.

In s(CASP), and unlike Prolog's negation as failure and ASP default negation, `not p(X)` can return bindings for `X` on success, i.e., bindings for which the call `p(X)` would have failed.

[4] For Linux and MacOS use the sh-terminal, for Windows the "Windows Subsystem for Linux" is required.
[5] For the reader's convenience the *help_all* output is available in Appendix A.

Example 1. *Considering the program p.pl*[6]

```
1 p(a).
```

let's run s(CASP) in the iterative mode by invoking `scasp -i p.pl`.

Then, we introduce the query `?- not p(X).` *and s(CASP) would return the binding* `X\=a`, *where* `\=` *is a disequality constraint meaning* $X \neq a$, *and the model* `{not p(X |{X\=a})}` *that represents the set of* $not\ p(X)$ *can be proven only when* $X \neq a$.[7]

Thanks to the interface of s(CASP) with constraint solvers, sound non-monotonic reasoning with constraints is possible.

Example 2. *Consider the program p2.pl:*

```
1 p(X):- X > 0.
```

which for the same query as above, returns the model `{not p(X | {X ≤ 0})}`.

As we mentioned before, s(CASP) supports uninterpreted functions (with the same conventions as Prolog), e.g., `[f(a)|Rest]` denotes a list with head `f(a)` and tail `Rest`. While in conventional ASP implementations this could give rise to an infinite grounded program, the s(CASP) execution model can deal with them similarly to Prolog, with the added power of the using constructive negation in the execution and returned models.

Example 3. *For the program member.pl*

```
1 member(X, [X|Xs]).
2 member(X, [_|Xs]):- member(X, Xs).
3
4 list([1,2,3,4,5]).
5
6 ?- list(A), not member(B, A).
```

the predicate `member/2` *models the membership of a list as usual in (classical) logic programming. The query derives the conditions for an element* `B` *not to belong to a given list* `A`.

Since we include the query as part of the program, invoking s(CASP) in automatic mode[8] *it returns the following model and binding:*

```
{ list([1,2,3,4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [1,2,3,4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [2,3,4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [3,4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [4,5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, [5]),
  not member(B | {B ≠ 1,B ≠ 2,B ≠ 3,B ≠ 4,B ≠ 5}, []) }
```

[6] All programs shown in this paper are available at http://platon.etsii.urjc.es/~jarias/papers/scasp-gde21/ and the results described have been obtained using s(CASP) under Ciao (version 0.21.08.25).

[7] Uniqueness of names is assumed for constants and function names: any two constants or functions with different names represent different objects.

[8] The automatic mode is selected by default so the flag `-a` or `--auto` can be omitted, i.e., `scasp member.pl`.

$A = [1,2,3,4,5],\ B \neq 1,\ B \neq 2,\ B \neq 3,\ B \neq 4,\ B \neq 5$

s(CASP), like the ASP implementations, is based on stable model semantics [4] and, unlike Prolog, supports non-stratified negation.

Example 4. *The following program, in weekend.pl:*

```
1 opera(saturday) :- not home(saturday).
2 home(saturday) :- not opera(saturday).
3 dinner(sunday).
```

models that on Saturday, Bob either goes to the opera or stays home and on Sunday he has dinner. The top-down evaluation of the non-stratified negation in lines 1-2 makes a loop with an even number of intervening negations (even loop). When the s(CASP) metainterpreter detects an even loop, the truth/falsehood of the atoms involved is assumed to generate different models, the consistency of which is subsequently checked. In this example, there are two possible models, and given a query s(CASP) returns (if exists) the relevant partial model for the query:

```
?- opera(saturday)          returns {opera(saturday),not home(saturday)}.
?- home(saturday)           returns {home(saturday),not opera(saturday)}.
?- dinner(sunday)                                  returns {dinner(sunday)}.
?- opera(saturday),home(saturday)                          returns no models.
```

Note that `opera(saturday)` *and* `home(saturday)` *cannot be part of the same model simultaneously, so for the last query, there are no (partial) models.*

In addition to default negation, s(CASP) supports classical negation (using the prefix '-') to capture the explicit evidence that a literal is false, e.g. `not opera(saturday)` means that we have no evidence that Bob goes to the opera (we can not prove it), and `-opera(saturday)` means that we have explicit evidence that Bob does not go to the opera (there is a proof for it).

4. Tuning of the Partial Answer Sets Output

In this section, we describe directives and flags that can be used to tune the output of s(CASP) to select the atoms that appear in the partial model, avoid consistency checks, generate a specific number of answers, and/or determine the accuracy of the real numbers.

Select the atoms that appear in the partial model: The directive `#show` can be used to select which atoms should appear in the partial models.

Example 5. *Consider the program weekend_show.pl that includes the directive:*

```
1 #show opera/1, home/1, dinner/1.
```

to the program in Example 4. Now, in the partial models returned by s(CASP) only positive atoms appear, so for the query `?- opera(saturday)` *it returns* `{opera(saturday}`.

Negated atoms can also be selected, e.g., `#show not home/1,-opera/1` is also valid.

```
1  opera(D) :- not home(D).          % A day D, Bob either goes to the opera...
2  home(D) :- not opera(D).          %                          ... or stays home.
3  home(monday).                     % On Monday, Bob stays at home.
4
5  :- baby(D), opera(D).             % When Bob's best friend comes with her baby, it is
6                                    % not a good idea to take the baby to the opera.
7  baby(tuesday).                    % They come on Tuesday.
8
9  ?- opera(D).                      % QUERY: When might Bob go to the opera?
```

Figure 1: s(CASP) encoding of opera.pl.

Denials: In ASP, *denials* are constructions of the form `:- p, q`, i.e., rules without head, and are used to express that the conjunction of atoms $p \wedge q$ cannot be true: either p, q, or both, have to be false in any stable model. Additionally, the s(CASP) compiler also detects statically rules of the form `r:-q,not r.`, called olon rules, and introduces denials to ensure that models satisfy $\neg q \vee r$ even if the atoms r or q are not needed to solve the query. Let us look at two examples.

Example 6. *For the following program in olon.pl*

```
1  p :- not q.        2  q :- not p.        3  r :- not r.
```

the compiler introduces the denial `:-not r` *which is checked for consistency after any given query. Therefore, s(CASP) returns no models, regardless of the initial query.*

Example 7. *Fig. 1 shows the code of opera.pl, an extended version of weekend.pl.*

The denial in line 5 expresses that the conjunction of atoms $baby(D) \wedge opera(D)$ *cannot be simultaneously true for any value of* D*. Thus for the query in line 9 the resulting partial model is:*

```
{ opera(D | {D \= monday,D \= tuesday}),  not home(D | {D \= monday,D \= tuesday}),
  not baby(Var1 | {Var1 \= tuesday}),  baby(tuesday),  not opera(tuesday),  home(tuesday) }
```

where the atom `opera({D |{D\=monday,D\=tuesday})` *means that Bob can go to the opera any day except on Monday or Tuesday.*

For debugging purposes, s(CASP) provides flags to disable consistency checks: (i) `--no_olon` disables the consistency check of denials introduced by the compiler due to olon rules but checks the user-defined denials; (ii) `--no_nmr` disables consistency check of all non-monotonic rules. In Example 6, runing s(CASP) by invoking `scasp --no_nmr -i olon.pl`, for the query `?- p` returns the model `{p,not q}` which may be useful for debugging.

Since consistency checks are evaluated when a tentative partial model is encountered, they introduce a run-time penalty. To reduce this overhead we propose a Dynamic Consistency Check (DCC) [13] which triggers NMR checks as soon a literal involving them is added to the partial model. In Section 6 we present promising preliminary results of this technique.

Partial Answer Sets: While in mainstream ASP implementations each answer corresponds to a stable model, in s(CASP) each answer is a partial answer set containing the subset of (negated) atoms that supports the query with specific binding of the free variables in the query. Consequently, two answer sets for a query may (not) correspond to the same stable model.

Example 8. *Considering the member.pl program of Example 3, for the query* `?- list(A), member(B,A)` *s(CASP) generates five answers sets, corresponding to the five possible bindings of* `B`*, i.e.,* `B=1`*,* `B=2`*,* `B=3`*,* `B=4`*, and* `B=5`*. While all five answer sets correspond to a single stable model, each partial model is different because they contain only the atoms needed to support the query with that specific binding, i.e., the partial answer set for* `B=1` *is:*

```
{ list([1,2,3,4,5]), member(1,[1,2,3,4,5]) }
```

and the partial answer set for `B=2` *is:*

```
{ list([1,2,3,4,5]), member(2,[1,2,3,4,5]), member(2,[2,3,4,5]) }
```

The `-sN` or `-nN` flag can be used to specify the number `N` of answer sets that s(CASP) should return in automated mode, and by adding the `-s0` or `-n0` flag s(CASP) would return all the possible answers sets.

Constraint precision: The s(CASP) system has a generic interface to enable plugging in constraint solvers. s(CASP) currently includes the CLP($\mathbb{Q}$) linear constraints solver [14], that supports the arithmetic constraints $<, >, =, \leq, \geq$. The selection of CLP($\mathbb{Q}$) instead of the faster CLP($\mathbb{R}$) is motivated by sound reasons. To transform the output from rationals to floating-point numbers, s(CASP) provides the flag `-r[=d]`, where `d` can be used to determine the (maximum) number of decimal places (default is 5 decimal places).

Example 9. *For the following program, in rationals.pl*

```
1 s(X,Y) :- X #= Y * 7/53.
```

the query `?- s(X,4.35)` *returns the binding* `X=609/1060`*, invoking s(CASP) without flags, and returns the binding* `X=0.574` *invoking* `scasp -r=3 rationals.pl`*.*

5. Justification and encoding (in natural language)

In the context of the European Union approved the General Data Protection Regulation (GDPR) [15] and Explainable Artificial Intelligence [16] (XAI), we are facing challenges that demand XAI to understand, appropriately trust, and effectively manage an emerging generation of artificially intelligent machine partners. However, justifying why an answer is a consequence from an ASP program may be non-trivial, more so when the user is an expert in a given domain, but not necessarily knowledgeable in ASP.

Justifications: s(CASP) uses top-down evaluation trees to generate minimal justifications in which it is possible to control which literals should appear.

```
JUSTIFICATION_TREE:
we assume that Bob goes to the opera on a day D not equal monday, nor tuesday, because
    Bob does not stay at home on D not equal monday, nor tuesday.
The global constraints hold, because
    the global constraint number 1 holds, because
        there is no evidence that 'baby' holds (for Var1), with Var1 not equal tuesday, and
        'baby' holds (for tuesday), and
        we assume that there is no evidence that Bob goes to the opera on the day tuesday, because
            'home' holds (for tuesday).
```

Figure 3: Human-readable justification of opera.pl.

Example 10. *(cont. Example 7) Consider the program opera.pl in Figure 1.*

When s(CASP) is run by invoking `scasp --tree opera.pl`*, in addition to the partial model, it generates the justification shown in Figure 2.*

The justification shows that, as we mentioned in Example 7, in this partial model, the atom `opera(D |{D \= monday, D \= tuesday})` *is assumed to holds, and this assumption is consistent with the denial. To check the denial, i.e., that the conjunction of atoms* `baby(D)` $\wedge$ `opera(D)` *is not simultaneously true for any value of* `D`*, s(CASP) checks that* $\forall$ `D ( not baby(D)` $\vee$ `(baby(D)` $\wedge$ `not opera(D)) )`.

```
JUSTIFICATION_TREE:
assume(opera(D | {D \= monday,D \= tuesday})):-
    not home(D | {D \= monday,D \= tuesday}).
denial :-
    not o_chk_1 :-
        not baby(Var1 | {Var1 \= tuesday}),
        baby(tuesday),
        assume(not opera(tuesday)) :-
            home(tuesday).
```

Figure 2: Justification of opera.pl.

The flags used to control the literals that appear in the justification are: (i) `mid`*, used by default, which only displays the user-defined predicates; (ii)* `long` *that displays all predicates, including auxiliary predicates such as the* `forall/2` *used to check this denial; and (iii)* `short` *which only displays the annotated literals. Additionally, the flag* `neg`*, used by default, includes the default-negated version of the annotated/selected predicates, whereas the flag* `pos` *does not.*

The s(CASP) justification framework provides a mechanism for presenting natural language justifications using a generic translation and the possibility of customize it with directives that provide translation patterns. Both plain text and expandable, user-friendly HTML files can be generated.

Example 11. *(cont. Example 10) Consider the following module in opera.pred.*

```
1  #pred opera(D) :: 'Bob goes to the opera on @(D:day)'.
2  #pred not home(D) :: 'Bob does not stay at home on @(D)'.
```

When executed s(CASP) by invoking `scasp --tree --human opera.pl opera.pred`*, the natural language justification is obtained (Figure 3). Note that lines 1, 2, and 7 follow the translation patterns defined in* `opera.pred`.

Additionally, adding the `--html=bob` flag, s(CASP) generates the HTML file bob.html, which includes the query, the bindings and an expandable justification tree that can be expanded and/or collapsed to facilitate its analysis.

Compiled code: Given an ASP program, a dual program compiled by s(CASP) will include the rules that express the constructive negation of the predicates in the original ASP program [2]. These dual rules provide a means to constructively determine the constraints under which a predicate would fail.

During the generation of the duals, for each clause, the compiler generates independent clauses with the "negated" literals in its body, and to avoid redundant answers, every i^{th} clause for a negated literal includes calls to any j^{th} literal with $j < i$.

Example 12. *For the clause* `h(X,Y):-r(X),not s(X,Y),q(Y)`*, its dual is:*

```
1 not h(X,Y) :- not r(X).
2 not h(X,Y) :- r(X), s(X,Y).
3 not h(X,Y) :- r(X), not s(X,Y), not q(Y).
```

In propositional programs this optimization is not desirable because we may lose relevant answers, so s(CASP) provides the `-d` or `--plaindual` flag to generate the duals with single-goal clauses.

When we want to modify the dual program, we can use the `--code`[9] flag to output the dual program, and then load an updated version with the `-c` or `--compiled` flag.

Additionally, translation patterns can be used to display the compiled program in natural language, thereby making it easier for experts without a programming background to understand both the program and the results of its execution.

Example 13. *(cont. Example 11) By invoking `scasp --code --human opera.pl opera.pred`, s(CASP) would display the dual program (including the duals and denials), following the translation patterns and/or the generic translation.*

6. Dynamic Consistency Check (Work in Progress)

As we mentioned before, the denials are evaluated after the query, ensuring that the tentative partial model is consistent with them. In cases where this evaluation fails the execution backtracks to check other alternatives. The goal of the Dynamic Consistency Check (DCC) is to check the denials as soon as atoms involved are added to the tentative partial model.

Our proposal is based on the constraint propagation algorithm, which, when the domain of a decision variable is modified, examines the constraints containing that variable to determine whether any values in the domains of other decision variables are now inconsistent. During ASP program evaluation, when an atom is added to the tentative partial model, the denials containing that atom are examined to determine if the presence of other atoms makes the tentative partial model inconsistent. In this case, the evaluation fails and is immediately backtracked to avoid computational waste. We believe that the benefits of the denial propagation in detecting

[9] By adding the `--raw` flag, the system prints the clauses in the same order as the compiler does.

```
1  % Graph
2  vertex(a).
3  vertex(b).
4  vertex(c).
5  vertex(d).
6  edge(b, a).
7  edge(b, d).
8  edge(a, c).
9  edge(a, b).
10 edge(b, c).
11 edge(c, d).
12 edge(d, a).
13 edge(c, a).
14 edge(a, d).
15 edge(d, b).
```

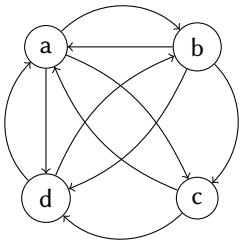

Figure 4: s(CASP) encoding and graph of graph.pl.

inconsistencies as early as possible outweigh the overhead introduced (i) during compilation to generate the DCC rules used to trigger denials involving specific atoms, and (ii) due to the extra work done to check denials when there are no inconsistencies.

Example 14. *Consider the standard ASP code for the Hamiltonian problem, in hamiltonian.pl*

```
1  #show chosen/2.
2  reachable(V) :- chosen(V, a).
3  reachable(V) :- chosen(V,U), reachable(U).
4  % Choose or not an edge of the graph.
5  chosen(U,V) :- edge(U,V), not other(U,V).
6  other(U,V) :- edge(U,V), not chosen(U,V).
7  % Every vertex must be reachable.
8  :- vertex(U), not reachable(U).
9  % Do not choose edges to/from the same vertex
10 :- chosen(U,W), U \= V, chosen(V,W).
11 :- chosen(W,U), U \= V, chosen(W,V).
12 ?- reachable(a).
```

where the denials in lines 10-11 are used to discard those tentative models that have chosen edges violating the properties of the Hamiltonian cycle. For the query in line 12, using the graph in Fig. 4 there are three stable models, one for each Hamiltonian cycle:

```
1  { chosen(a,c), chosen(c,d), chosen(d,b), chosen(b,a),... }
2  { chosen(a,b), chosen(b,c), chosen(c,d), chosen(d,a),... }
3  { chosen(a,d), chosen(d,b), chosen(b,c), chosen(c,a),... }
```

However, s(CASP) follows the generate-and-test paradigm, i.e., it generates multiple cycles that are checked for consistency when a cycle that reaches all vertex is complete. As a consequence, if the evaluation chose two edges to/from the same vertex, trying combinations (on backtracking) with the rest of edges would be waste of effort.

Evaluation: We compared the performance of s(CASP) with and without DCC using a MacOS 11.5.2 Intel Core i7 at 2.6GHz. We observe that using DCC, for the evaluation of the Hamiltonian problem (Example 14) with the graph in Figure 4, we obtain a speedup of 6.8:

- Without DCC: invoking `scasp --prev_forall -n0 hamiltonian.pl graph.pl`.[10] the evaluation of the three models takes 8.266s.
- With DCC: invoking `scasp --prev_forall -n0 --dcc hamiltonian.pl graph.pl hamiltonian_dcc.pl`,[11] the evaluation, using this preliminary DCC implementation, takes only 1.215s.

[10]Note that we are using a specific `forall/2` implementation to run this example (more details in Section 7).

[11]Note that the DCC implementation is unfinished and the DCC rules for the denials in lines 10-11 are included manually in hamiltonian_dcc.pl. In the future, they will be compiled automatically.

7. Future Work

As already mentioned, the implementation has been improved in several aspect but can still be substantially improved, and in particular we are planning to:

- Finish the DCC implementation and work on using analysis to optimize the compilation of the DCC rules, being able to interleave their execution with the top-down strategy to discard models as soon as they are shown to be inconsistent.
- Use dependency analysis to improve the generation of the dual programs.
- Apply partial evaluation and better compilation techniques to remove (part of) the overhead brought about by the interpreting approach.
- Optimize the implementation of the *c-forall* algorithm using tabling [17]. Currently there are four alternatives (by default, `all_c_forall`, `prev_forall`, and `sasp_forall`) but all of them enter loops, generate redundant answers and/or discard solutions.
- In addition, tabling, and ATCLP [18], can be use to (i) collect the minimal partial models, increasing performance and readability, and (ii) increase the range of supported programs, by handling positive variant loops that are now detected and halted. Note that halting variant loops may cause a loss of solutions, so s(CASP) provides the `--variant` flag to disable this behavior which can lead to loops.
- Improve the disequality constraint solver to handle pending cases. The `-w` or `--warning` flag can be used to warn if an unsupported constraint or variant loop is detected.
- Finally, although s(CASP) provides flags (`v`, `v0`, `v1`, and `v2`) to trace a program evaluation, better integration with the Ciao and SWI-Prolog debuggers is desirable.

References

[1] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint Answer Set Programming without Grounding, Theory and Practice of Logic Programming 18 (2018) 337–354. doi:10.1017/S1471068418000285.

[2] K. Marple, E. Salazar, G. Gupta, Computing Stable Models of Normal Logic Programs Without Grounding, arXiv 1709.00501 (2017). URL: http://arxiv.org/abs/1709.00501. arXiv:1709.00501.

[3] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for goal-directed constraint answer set programming, in: Proceedings 36th International Conference on Logic Programming (Technical Communications), volume 325 of *EPTCS*, Open Publishing Association, 2020, pp. 59–72. doi:10.4204/EPTCS.325.12.

[4] M. Gelfond, V. Lifschitz, The Stable Model Semantics for Logic Programming, in: 5th International Conference on Logic Programming, 1988, pp. 1070–1080. URL: http://www.cse.unsw.edu.au/~cs4415/2010/resources/stable.pdf.

[5] L. M. Pereira, J. N. Aparício, Relevant counterfactuals, in: EPIA 89, 4th Portuguese Conference on Artificial Intelligence, Lisbon, Portugal, September 26-29, 1989, Proceedings, 1989, pp. 107–118. doi:10.1007/3-540-51665-4_78.

[6] J. Arias, Z. Chen, M. Carro, G. Gupta, Modeling and Reasoning in Event Calculus Using Goal-Directed Constraint Answer Set Programming, in: Pre-Proc. of the 29th Int'l. Symposium on Logic-based Program Synthesis and Transformation, 2019.

[7] B. Hall, S. C. Varanasi, J. Fiedor, J. Arias, K. Basu, F. Li, D. Bhatt, K. Driscoll, E. Salazar, G. Gupta, Knowledge-Assisted Reasoning of Model-Augmented System Requirements with Event Calculus and Goal-Directed Answer Set Programming, in: Proc. 8th Workshop on Horn Clause Verification and Synthesis, 2021.

[8] Z. Chen, K. Marple, E. Salazar, G. Gupta, L. Tamil, A Physician Advisory System for Chronic Heart Failure Management Based on Knowledge Patterns, Theory and Practice of Logic Programming 16 (2016) 604–618. doi:10.1017/S1471068416000429.

[9] F. Shakerin, G. Gupta, Induction of Non-Monotonic Logic Programs to Explain Boosted Tree Models Using LIME, in: AAAI 2019, 2019, pp. 3052–3059. doi:10.1609/aaai.v33i01.33013052.

[10] S. C. Varanasi, E. Salazar, N. Mittal, G. Gupta, Synthesizing Imperative Code from Answer Set Programming Specifications, in: LOPSTR, volume 12042 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 75–89. doi:10.1007/978-3-030-45260-5_5.

[11] K. Basu, S. Varanasi, F. Shakerin, J. Arias, G. Gupta, Knowledge-driven Natural Language Understanding of English Text and its Applications, AAAI'21 (2021).

[12] J. Arias, M. Moreno-Rebato, J. A. Rodriguez-García, S. Ossowski, Modeling Administrative Discretion Using Goal-Directed Answer Set Programming, in: Advances in Artificial Intelligence, CAEPIA 20/21, Springer International Publishing, Cham, 2021, pp. 258–267.

[13] K. Marple, G. Gupta, Dynamic Consistency Checking in Goal-Directed Answer Set Programming, Theory and Practice of Loging Programming 14 (2014) 415–427. URL: https://doi.org/10.1017/S1471068414000118. doi:10.1017/S1471068414000118.

[14] C. Holzbaur, OFAI CLP(Q,R) Manual, Edition 1.3.3, Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[15] European Union, General Data Protection Regulation (GDPR), Regulation (EU) 2016/679 of the European Parliament and of the Council, 2016. https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[16] DARPA, Explainable Artificial Intelligence (XAI), Defense Advanced Research Projects Agency, 2017. https://www.darpa.mil/program/explainable-artificial-intelligence.

[17] J. Arias, M. Carro, Description, Implementation, and Evaluation of a Generic Design for Tabled CLP, Theory and Practice of Logic Programming 19 (2019) 412–448. URL: https://arxiv.org/abs/1809.05771. doi:10.1017/S1471068418000571.

[18] J. Arias, M. Carro, Incremental evaluation of lattice-based aggregates in logic programming using modular TCLP, in: J. J. Alferes, M. Johansson (Eds.), 21st Int'l. Symposium on Practical Aspects of Declarative Languages, volume 11372 of *LNCS*, Springer, 2019, pp. 98–114. URL: https://doi.org/10.1007/978-3-030-05998-9_7. doi:10.1007/978-3-030-05998-9_7.

A. Output of scasp –help_all

```
-h, -?, --help          Print this help message and terminate.
--help_all              Print extended help.
-i, --interactive       Run in interactive mode (REP loop).
-a, --auto              Run in batch mode (no user interaction).
-sN, -nN                Compute N answer sets, where N >= 0. N = 0 means 'all'.
-c, --compiled          Load compiled files (e.g. extracted using --code).
-d, --plaindual         Generate dual program with single-goal clauses
                        (for propositional programs).
-r[=d]                  Output rational numbers as real numbers.
                        [d] determines precision. Defaults to d = 5.

--code                  Print program with dual clauses and exit.
--tree                  Print justification tree for each answer (if any).

--plain                 Output code / justification tree as literals (default).
--human                 Output code / justification tree in natural language.

--long                  Output long version of justification.
--mid                   Output mid-sized version of justification (default) .
--short                 Short version of justification.

--pos                   Only display the selected literals in the justification.
--neg                   Add the negated literals in the justification (default).

--html[=name]           Generate HTML file for the justification. [name]:
                        use 'name.html'. Default: first InputFile name.

-v, --verbose           Trace user-predicate calls.
-v0                     Trace user-predicate calls (show tree).
-v1                     Trace user-predicate failures.
-v2                     Trace user-predicate failures (show tree).
--update                Automatically update s(CASP).
--version               Output the current version of s(CASP)

--dcc                   Activate the Dynamic Consistency Check.

--all_c_forall          Exhaustive evaluation of c_forall/2.
--prev_forall           Deprecated evaluation of forall/2.
--sasp_forall           Deprecated evaluation of forall/2.

--no_olon               Do not compile olon rules (for debugging purposes).
--no_nmr                Do not compile NMR checks (for debugging purposes).
-w, --warning           Enable warning messages (failures in variant loops / disequality).
--variant               Do not fail in the presence of variant loops.

-m, --minimal           Collect only the minimal models (TABLING required).
--raw                   Sort the clauses as s(ASP) does (use with --code).
```