

Theory Revision with Goal-directed ASP

Elmer Salazar¹

¹University of Texas at Dallas, 800 W. Campbell Road, Richardson, Texas 75080-3021

Abstract

The s(CASP) system uses a proof-theoretic approach to ASP. We can use its feature to generate a justification tree to identify needed changes to a program.

Keywords

Theory Revision, Goal-directed, Stable Model Semantics, ASP

1. Introduction

The s(CASP) system, based on s(ASP)[1], is a top-down, goal-directed system[2]. This means it computes, for each query, a proof (if possible) that that query is true. If instead we want the query to be false (such as asking if some safety constraint is violated), then by looking at the proof, we can determine how to change the program so that the proof would fail.

The “system” we want to check needs to first be encoded as a s(CASP) program. Currently, only propositional programs are supported. We can view the system as having sensors to monitor its environment and actuators to act upon it. This does not mean, however, the system must model a cyber-physical system. The sensors are really just knowledge the system can know, but not determine, and the actuators are knowledge the system determines. We must identify the knowledge detected by these sensors. This knowledge represents a state of the systems environment, and can be represented as abducibles. An abducible is a proposition for which we have the choice of making it true or false, as needed. Finally, we must identify the properties we wish to enforce on the system. These will be encoded as propositions, with rules that determine when the properties are satisfied. We illustrate this in figure 1.

2. Finding Suggested Defeaters

Now that we have the system encoded, we can look for *defeaters*. The concept of defeaters in this work comes from research in software assurance.[3] For our purposes, though, a defeater can be thought of as a change that could be made to the system to fix a violation of the expected properties. To find such defeaters we run the program with s(CASP), querying the negation of our properties. In this example, that would be “?- burndown.”. We will use s(CASP)’s “--tree” option to get the proof tree, and look for propositions with rules that we can change. Since


ICLP'21: 37th International Conference on Logic Programming, September 20–27, 2021

✉ elmer.salazar@utdallas.edu (E. Salazar)

ORCID 0000-0002-3042-474X (E. Salazar)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

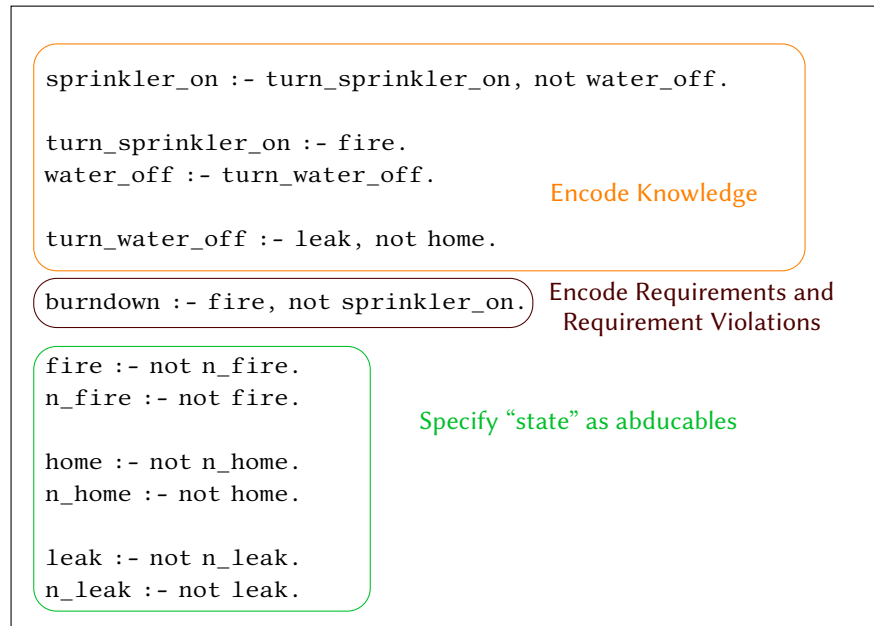


Figure 1: Program 1

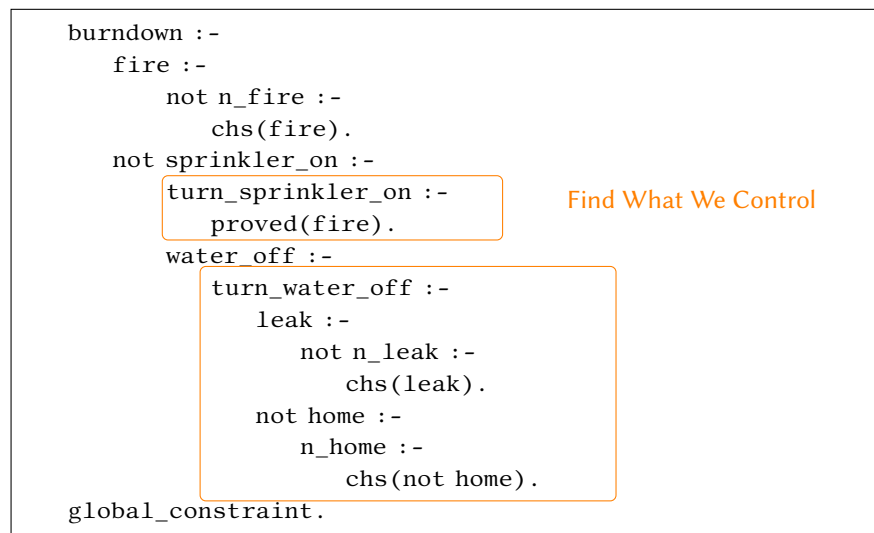


Figure 2: Justification Tree for Program 1

the only way the system can affect the knowledgebase is through the “actuator” propositions, we will limit ourselves to rules for these propositions. In this example those propositions are “turn_sprinkler_on” and “turn_water_off” as shown in figure 2.

Once we have identified the subtrees we can use, we look for a proposition (or its negation) that is false in the model associated with the tree provided by s(CASP). If the subtree or any of its ancestors are not dependent on that proposition, it becomes a suggested defeater. It turns

out ignoring all such propositions can lead to missing valid defeaters in some cases. Further work is being done to determine better heuristics for filter out possible suggestions. With this, our example has the following possible defeaters:

- For rule: `turn_water_off :- leak, not home.`
 - ADD: `not fire`
- For rule: `turn_sprinkler_on :- fire.`
 - ADD: `not leak, or`
 - ADD: `home`

3. The Suggestions

Each suggestion, when used to modify the program, will ensure that the proof cannot succeed. This makes no guarantee the query will not still succeed, nor that the changes will make sense according to our interpretation of the program. To combat the former case, we run the above algorithm for every tree the query causes. Each of these trees represent a different way the query can succeed. The generated suggestions are grouped together with their proof tree and model and used to generate a knowledgebase to be used with s(CASP). This knowledgebase can then be combined with some common sense and domain specific knowledge to reason about the “best” defeater. Once we have the “best” defeaters, we can modify the program and try again. After all, the change itself may introduce a new way for the query to succeed.

The second case, of suggestions that do not make sense, can be easily encountered. In the example above, we can ensure this proof fails by ensuring `turn_sprinkler_on` fails. This is counterproductive. We know that by not turning on the sprinklers when there is a fire, the house will always burn down – regardless of the rest of the state. To make a more intelligent choice, the possible defeaters, along with the associated model and the original program, are combined, as data, with another s(CASP) program. The purpose of this program is to filter out the irrelevant defeaters. For the above example, we may add a rule that does not keep the second set of defeaters. There are three parts to this stage. First, is the defeaters, models, and program as stated above. These are presented as data in the program to be analysed and processes. Secondly, a driver program that contains non-domain specific knowledge that contains the following query: `?- suggest(A)`. As of this writing, the driver program provides a default implementation for `suggest` that simply provides each defeater, as is. The third part is the domain specific logic. This program is defined for the specific system and defines `suggest/1`. For the above example, we can provide the following implementation for `suggest/1`:

```
suggest(A) :- defeaters(_,Defs), find_suggestions(Defs,A).

find_suggestions([H|T], H) :- H=defeater(proof(neg,_,_),_).
find_suggestions([H|T], H) :- H=defeater(proof(pos,Pred,_,_),_),
    ↪ Pred\=turn_sprinkler_on.
find_suggestions([_|T], S) :- find_suggestions(T,S).
```

The first rule defines `suggest/1`. The call to `defeaters/2`, provided when generating the knowledgebase, gets the list of defeaters for a proof tree. The `find_suggestions/2` call loops through these defeaters binding `A` to a valid defeater. Each binding of `A` is a possible way of making the proof fail. The other three rules define what we think are valid suggestions. The third rule is a simple recursive case that allows us to traverse the list of defeaters. The decision is being made by the first two rules. The first rule for `find_suggestions/2` allows suggestions for duals (the negation of a proposition). The second rule considers positive literals (propositions without negation). However, It only accepts such defeaters if the proposition is not `turn_sprinkler_on`. Since we know that `turn_sprinkler_on` can only be true if there is a fire, by disallowing it from being made false we are enforcing the constraint "The sprinkler must turn on when there is a fire". Using this definition of `suggest/1`, the second set of defeaters (from the output given above) will not be given. This gives us only the suggestion:

- For rule: `turn_water_off :- leak, not home.`
- `not fire`

This answer makes the most sense according to our interpretation of the code.

4. Defeaters for Dual Rules

The above example illustrates how the system behaves when a proposition needs to be made false. However, `s(CASP)` completes the program using dual rules, allowing for constructive negate. It is possible that instead of a falsifying a proposition, we will want to falsify its dual. We consider three ways of falsifying a dual. The first is to remove a goal, for which the dual is dependent, from the body of the original rule. The second is to add a new rule for the proposition that depends on some proposition (or its negation) that is true in the model. The third way is similar to the second, but we also include the body of the dual rule. Consider the following example.

```
fed :- eatfood.  
fed :- not hungry.  
  
eatfood :- hungry.  
warm :- blanket.  
blanket :- cold.  
  
hungry :- not n_hungry.  
n_hungry :- not hungry.  
cold :- not n_cold.  
n_cold :- not cold.  
  
happydog :- fed, warm.
```

One set of suggestions given is:

- For rule: `blanket :- cold.`
 - Remove `cold`
 - Add new rule: `blanket :- hungry.`
 - Add new rule: `blanket :- not cold, hungry.`

These suggestions can be interpreted as “use the blanket all the time, even when it is not cold” and “use the blanket when hungry”. Neither of which really makes sense. The true problem is that we did not specify that if it is not cold then the dog is warm even without a blanket. We could consider the warm proposition as an “actuator”. Still, since warm depends on cold (through `blanket`) we will get no suggestions.

5. Further Work

Many problems we face can be categorized as not having enough information to make proper decision. The future of this work will be in a knowledge driven approach. Theories about the domain can be constructed, preferably by a domain expert, and used to guide the search for defeaters. With theories, we can better determine if a defeater makes sense while still making connections to disconnected knowledge in our code. The earliest work will consider *claims* that our “system” can use to establish its behavior. Theories that directly challenges the ability to use these claims can ensure they are sufficiently supported, and can be used to identify defeaters.

As another avenue of improvement, we want to bring this algorithm from the propositional to the predicate case. As is, if `s(CASP)` outputs a ground justification tree and model then the algorithm will work even with a non-propositional program. Each instance of a predicate, however, is treated as an independent proposition. So, the fact that some predicate instances are generated from the same rule (and therefore affected by the same defeater) is ignored. We would like to make this process a bit more intelligent and even go further, allowing for variables in the output with CLPQ constraints.

6. Conclusion

The `s(CASP)` system computes a proof that a query is true. We can take advantage of this by querying something we would like to fail, such as a violation of safety property. By traversing the tree we can identify changes to the rules in the program that would cause this proof to fail. This allows us to identify possible defeaters we can use to revise the program.

References

- [1] K. Marple, E. Salazar, G. Gupta, Computing stable models of normal logic programs without grounding, arXiv preprint arXiv:1709.00501 (2017).
- [2] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint answer set programming without grounding, TPLP 18 (2018) 337–354. doi:10.1017/S1471068418000285.
- [3] R. Bloomfield, J. Rushby, Assurance 2.0: A manifesto, 2020. ArXiv:2004.10474v2.