Visualization of Reasoning: Debugging Techniques in ASP

by

Justin Rodriguez, B.S.

A Thesis

In

Computer Science

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

Yuanlin Zhang, Ph.D.
Chair of the Committee

Susan Mengel, Ph.D.

Mark Sheridan
Dean of the Graduate School

May, 2021

ACKNOWLEDGMENTS

There are many people who have assisted me throughout my journey in completing this master's thesis. Without them, this process would not have gone as smoothly, and the experience would not have been the same. I owe all involved a great deal of gratitude and would like to thank a few personally.

I would like to thank my thesis advisor, Yuanlin Zhang, for being an incredible mentor in the undertakings of academic research and writing. He is the one who first introduced me into the topic of logic programming and the importance of knowledge representation and reasoning as a useful skill in computer science. My first course with him was in Intelligent Systems and I have since taken many more with him throughout my graduate studies. He is incredibly skilled in the work he does and cares immensely for the students that he teaches.

I would lastly like to thank my parents for all the support they have given me in both my undergraduate and graduate studies. As a first-generation student, I would never have thought to be in graduate school completing a thesis, if it were not for all their handwork and countless support they never hesitated to give me.

TABLE OF CONTENTS

ABSTRACT

Answer Set Programming (ASP) has been an area of logic programming and knowledge representation filled with a lively community in the academic world. However, despite having such a lively community, it has become increasingly difficult to find appropriate debugging systems that satisfy a majority of what is to be expected by programmers if they were writing a program in any other commonly used language. Recent literature on the subject has focused on mainly incorporating command line and text-based systems as a form of debugging answer set programs. While these systems do indeed aid the programmer in the debugging process, due to the intricacies of logic programming, it can be difficult to present clear reasoning behind why a program or answer set to a program is inconsistent. We propose the introduction of visualization of reasoning approach to debugging logic programs. By visualization of reasoning, we mean a medium of conveying to users the reasoning behind how their programs function (or do not) that is not text-based. One new technique we propose is our "firing" model, where users can see an animation of a sequence in their program during a query. This visualization we hope will give programmers a better understanding of the reasoning behind what the query returns and why parts of their program do not work as intended. Besides applications in debugging, creating a visualization of reasoning can also help in giving students a better understanding of the intricacies of ASP, or about any subject for that matter, as well as provide greater access to learning logic programs. We believe that we have created such visualizations that achieve each one of these aspirations.

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Our research focuses on the concept of creating a visualization of reasoning in the context of debugging programs, specifically Answer Set Programming (ASP).

By visualization of reasoning, we mean the idea of being able to animate or bring to life a user's program in the hopes of providing a better understanding of their work. To put into more direct terms, create a more refined debugging process within Logic programming. The major component involves creating a visual which allows the programmer to see each step in a program's execution or some specific sequence of the program. This is analogous to stepping through frames in a video. Individually, they are nothing more than a random picture, but together they create context and clear meaning. This is one of a few techniques that will be discussed throughout this paper concerning the visualization of reasoning.

Through the use of incorporating some sort of visualization into the debugging process, we hope to give users a new way (or mindset) at examining the programs that they might not otherwise consider with conventional text-based debugging methods. When users can see their program more tangibly (besides words and numbers), it becomes easier to see where the abnormalities/errors lie within the environment they created.

Not much work has been presented on this topic, especially when it pertains to the area of logic programming, so it is with great pleasure that we can give our ideas in this area. Before we proceed with the discussion of these techniques, we will briefly go over some major concepts that lay the foundation for visualization of reasoning in debugging.

What is Answer Set Programming?

Answer Set Programming (ASP) is a form of declarative programming, whose main purpose/uniqueness lies in solving difficult search problems (Lifschitz 2008), and is based in the semantics of Logic Programming (Gelfond and Lifschitz 1988). ASP utilizes the concept of constraint programming; the process of finding some assignment of values to corresponding variables such that all constraints in a given program are satisfied**,** in order to generate the correct corresponding answer sets. ASP was created in a time where research into non-monotonic reasoning was at a high point and since has become quite useful when dealing with knowledge-intensive programs. ASP has also recently become a popular paradigm to be studied in Computer Science departments, especially when used as an introductory course into Intelligent Systems. This is partly the reasoning behind examining a visualization of reasoning for debugging in ASP, as more departments begin to introduce the programming scheme.

Debugging

The term debugging became a popular term within the computer science/engineering community by Grace Hopper, a pioneer in the Computer Science world, and her colleagues when they had to remove moths from their Harvard Mark II that

were preventing the machines from running. In a modern sense, debugging involves users tracing through a program to find snippets of code that affect the execution or solution to the program. These errors can typically be categorized into four major groups:

1. **Lexical**
    a. These are errors which occur when a part of the statement in the given language is not a valid word. In the case of programming languages, these can be found by the lexical analysis (or a tokenizer) phase of the language complier/interpreter.
2. **Syntactic**
    a. The words in the statement are valid but they do not form a valid sentence. Again, these errors can be caught in the language's complier.
3. **Semantic**
    a. The statement is well constructed but does not have a valid meaning. Depending on both the language and the error, it may be possible to detect theses at compile time or only at run time (if at all).
4. **Conceptual**
    a. The statement is technically correct but not what the user intends. "The sky is blue" is valid English but says nothing useful about the whether or not the ground is brown. These are the hardest errors to detect, so the use of verification against the given product specification u to help prevent such errors.

These four categories are useful when attempting to build different visualization techniques for debugging. When we are able to break up the different types of errors, we can better see the various

When it comes to debugging in ASP, logic programming in general, we define errors in the following manner:

1. The program has no answer set but is supposed to have on
    a. Case 1: We know what the answer set should be.
    b. Case 2: We do not know the answer set but suspect there to be one.
        i. Case 2.1: We do know one atom that must be there.
        ii. Case 2.2: We know nothing about what should be in answer set.
2. The program has an answer set:
    a. An atom exists within the answer set that should not.
    b. An atom is missing from the answer set.
    c. An answer set is missing from the solution set.
    d. An answer set of the program.

Having this breakdown of how we consider errors to be when it comes to ASP allows us to create various visualization techniques that best fit the error type. And as can be seen the major thing when debugging an ASP program, it is crucial to have some

knowledge at to what the final answer set(s) should look like. Without such knowledge, debugging ASP programs becomes an exhaustive task.

Thesis Layout

This thesis consists of four parts. Part I is used to give a brief introduction into the concept that is examined in this paper, as well as provide context for other topics related to the study of visualization of reasoning.

Part II goes over previous work that has been done relating to both debugging and visualization techniques in ASP and logic programming in general. Part III introduces our research and proposed contributions to creating a visualization of reasoning. And lastly, Part IV concludes the paper by summarizing our research and contributions.

CHAPTER II

REVIEW OF LITERATURE

Debugging techniques have been implemented for ASP in various manners, though none as in-depth towards visualizing a user's program and errors in a way which we propose. Most debuggers for logic programming work in a similar vein to those in any other programming paradigm; highlighting portions of a program to signify some syntactic or semantic error has occurred. With the syntactic errors being the easier of the two to identify, thanks to the use of IDEs (Integrated Development Environment). On the other hand, semantic errors lend themselves to being more difficult to find due to the declarative complexion of logic and answer set programming. Whereas in procedural programming, one can go line-by-line to pinpoint the cause of inconsistent behavior, ASP relies heavily on backtracking when computing results. While both error types can be shown in some fashion with current debugging techniques, in which attention is brought to the user that they have done something incorrectly, they do not produce such a sufficient explanation as to why what the programmer did was incorrect within the context of their program. The subsequent sections are examples of different types of debugging applications for logic programs that are considered to be inconsistent — those in which no answer sets exist.

Guided Model

The simplest form of debugging across any programming paradigm is what is considered the guided model (Brna, Brayshaw, et al. 1991). When a program has no answer set but a solution is known, the system works with the programmer (who are referred to as "oracles") through a series of queries to help "guide" the user to the potential error and possible solution to correct the error(s). The system works in two ways: one for when the user can identify an expected result for some terminating query and one for when they cannot. For the latter, the system then works to check for either an incorrect variable declaration or for some missing answer set. In either case, it is ultimately up to the user to determine the best course of action to take fix any errors which were guided to them by the system. This process must be repeated each time for however many errors exist within the program or until the programmer is satisfied with the outcome of the program.

Tracing Model (AKA the Byrd Box Model)

Similar to the guided method of debugging, the tracing method seeks to model the execution of a program and print out what is occurring during a sequence of goals to show where the program has reached during its execution (Clocksin and Mellish 1981). The tracing model's method of visualization is through boxes (as the name lends itself). Each call to a procedure within this method is represented as a box, with each box consisting of four parts - Call, Exit, Fail and Redo. Figure 2.1 gives an example program with a stated goal, for which Figure 2.2 gives a representation using the tracing model. In both figures, the programmer can see how the systems interpret their program hierarchy, relative to the stated goal, as well as logical process the program goals through to find the correct answer

to the stated goal. When a path is taken which doesn't lead to the correct answer, the system is even able to give notice that it is attempting to backtrack out of the current path and into the next available option. While this is still for the most part a text-based debugging system, it does offer the programmer an expansive view into their program and a clear reasoning process into solving some stated goal.

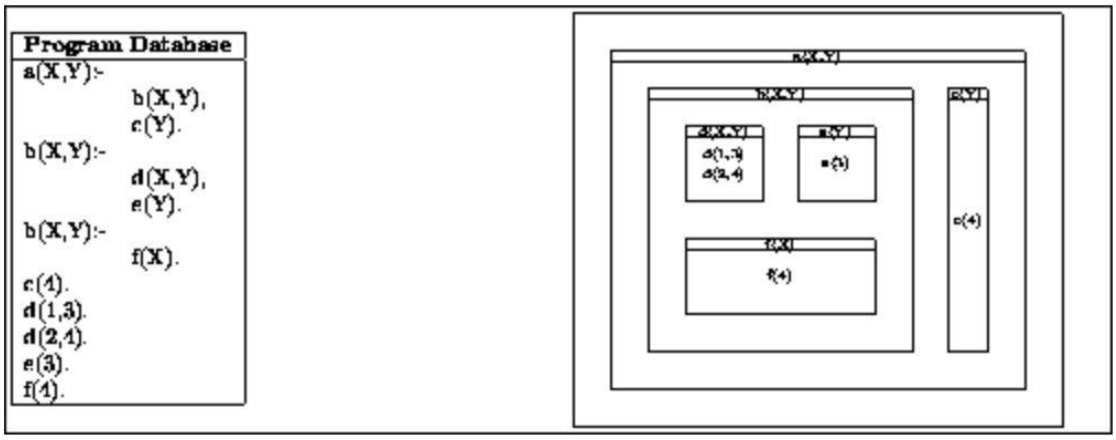


Figure 2.1: Program with Byrd Box Model (Brna, Prolog Programming: A First Course 1999)

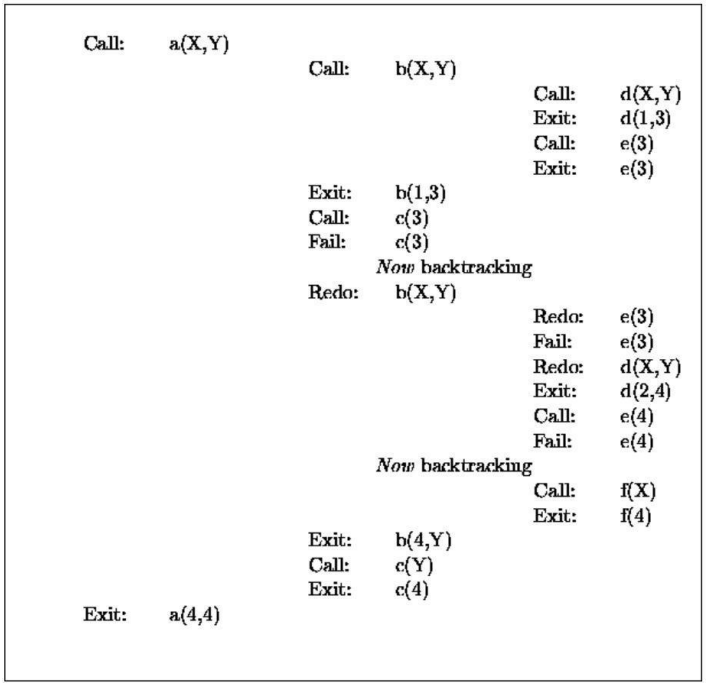


Figure 2.2: Goal being solved with model (Brna, Prolog Programming: A First Course 1999)

The Spock System

The Spock System (Gebser, et al. 2009) explores why a potential answer set is not a correct answer for a given ground program, i.e., assigning variables in the program values given as defined facts of the program. Through various command line arguments, Spock is able to output a program answer sets (see Figure 2.3), and more importantly, give the programmer a reason as to why an answer would not be valid given the constraints created within the program.

Program P2 (inconsistent):

```
r1: a :- not b.
r2: b :- not b.
```

Spock Command Line Output:

```
M25 = {a, b, unsupported(a), unsupported(b), blocked(r1), blocked(r2)}
M26 = {b, unsupported(b), blocked(r1), blocked(r2)}
M27 = {a, unfounded(a), unsatisfied(r2), applicable(r1), applicable(r2)}
M28 = {a, unsatisfied(r2), applicable(r1), applicable(r2)}
M29 = {unsatisfied(r1), unsatisfied(r2), applicable(r1), applicable(r2)}
```

Figure 2.3: Example program and Spock output (Fandinno and Schulz 2019)

This is achieved by turning the given program into a meta program (i.e., a program over a meta-language which manipulates another program over an object language) to get "conditions for the applicability of rules" in the original program.

The Ouroboros System

The Ouroboros System (Oetsch, Puhrer and Tompits , Catching the Ouroboros: On Debugging Non-Ground Answer-Set Programs 2010) is a debugging method for non-ground programs. By that, we mean programs where variables exist within the rules that have not been assigned some value from the knowledge base of the program. It requires that a user have an intended answer set already in mind, to compare with, which it then uses to build an explanation as to whether or not the intended answer set is a valid one. Thus, Ouroboros is more reliant on the user already knowing what their program should return, in terms of answer sets, rather than letting the system figure it out. Ouroboros can

handle disjunctive logic programs that use constraints, integer arithmetic, and strong negation. This a major difference from Spock and other methods, as they do not consider non-ground programs. While the Spock system does well in explaining inconsistencies for a program's given answer set, it does not explicitly handle the variables occurring in the program as it requires a nuanced grounding technique. Ouroboros builds upon Spock's use of a meta program to construct explanations of "inconsistent extended logic programs possibly compromising variables" (Oetsch, Puhrer and Tompits 2010), however as mentioned earlier, it does require that the user have an intended answer set available. Figure 2.4 depicts Ouroboros being implemented as a debugging tool within the SeaLion IDE (Oetsch, Puhrer and Tompits 2011)for ASP programs.

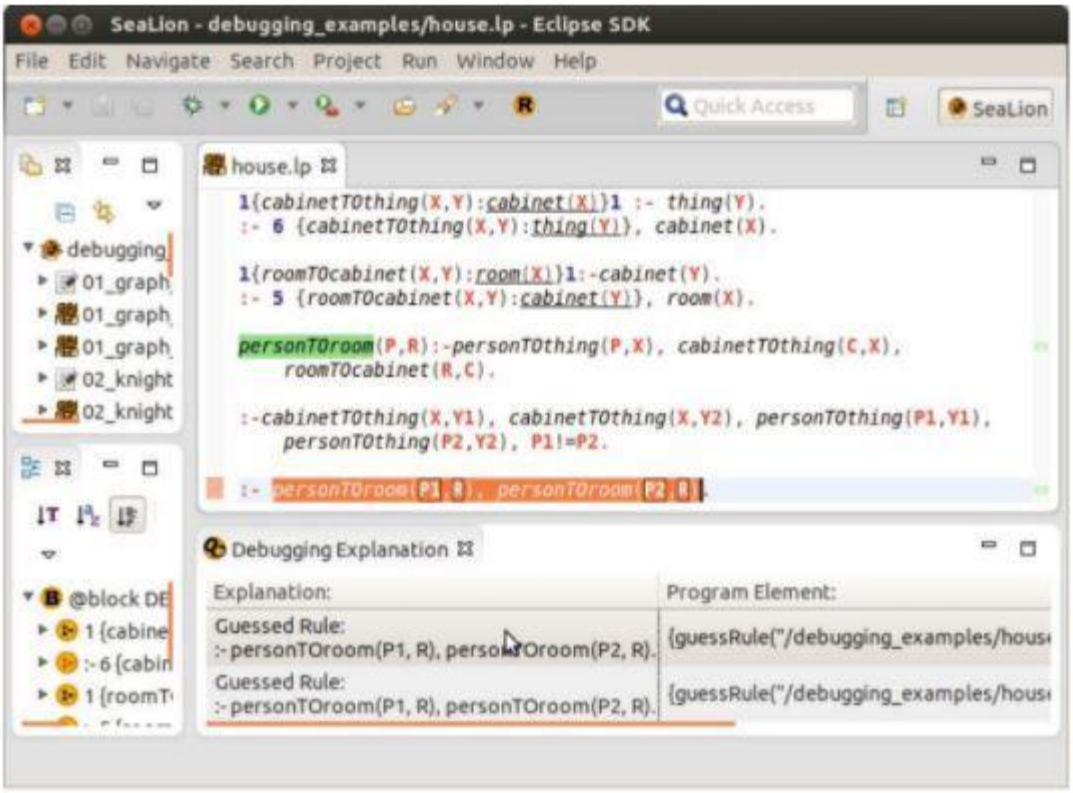


Figure 2.4: Ouroboros debugging in SeaLion (Oetsch, Puhrer and Tompits , Catching the Ouroboros: On Debugging Non-Ground Answer-Set Programs 2010)

Stepping

Extending on their work with the Ouroboros system design in debugging, Stepping (Oetsch, Puhrer and Tompits 2011) seeks to tackle the problem of being able to debug ASP in a more procedural manner, like that which is done in imperative programming languages. However, unlike imperative programming where the sequence of steps that the program runs through is predetermined, this stepping methodology allows for the programmer to choose which rules are to be considered active in each step of running through the program. The user first selects the portion of their program they want to step through by highlighting it while in the debugging mode. Once a stepping sequence has

been determined, the program starts with an empty potential answer set, and through each step of the desired sequence the user is given the rules and corresponding results that are applicable. These results from the rules that are satisfiable are then added to the answer set for the given sequence. While this means that the programmer must have some general idea about their program's functionality, it does allow a greater focus and probability of success in finding potential bugs. The stepping methodology allows for the possibility to lower the barrier of entry into ASP programming, as it gives programmers one more debugging tool for ASP (something that is not widely available in comparison to imperative programming languages). This methodology is also most popularly integrated within the *SeaLion IDE*. Figures 2.5 and 2.6 showcases a snippet of an example using the stepping method (Fandinno and Schulz 2019, 68):

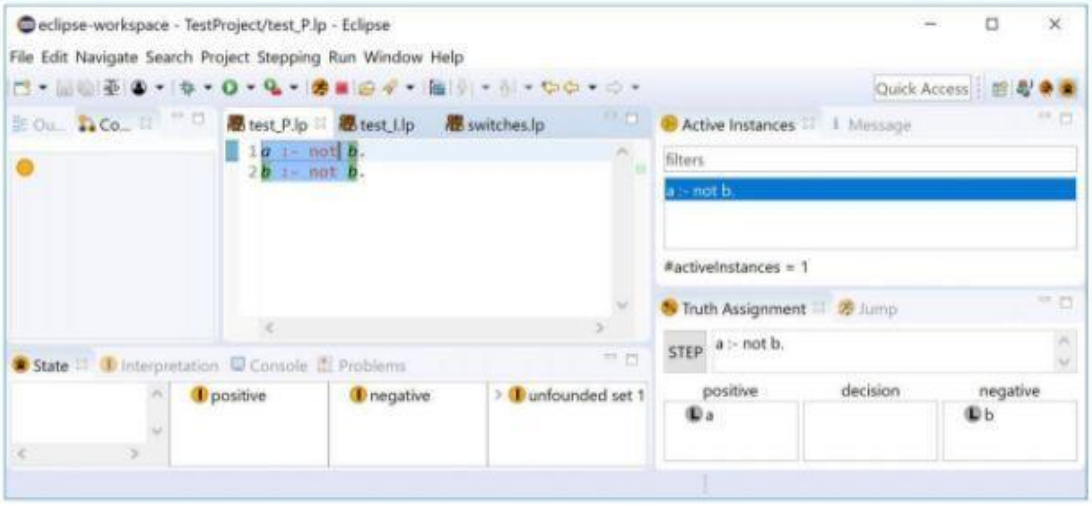


Figure 2.5: Stepping through program (Fandinno and Schulz 2019, 68)

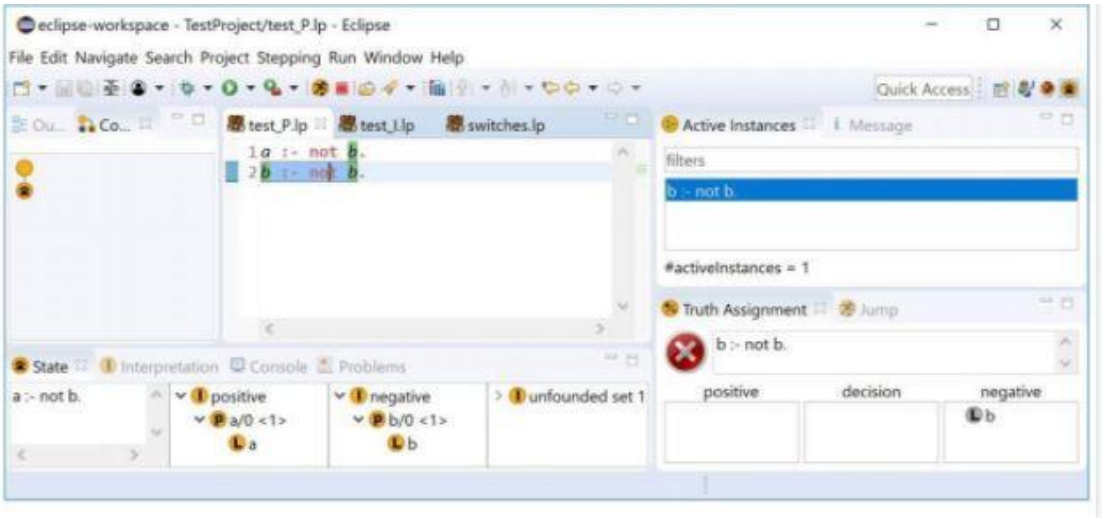
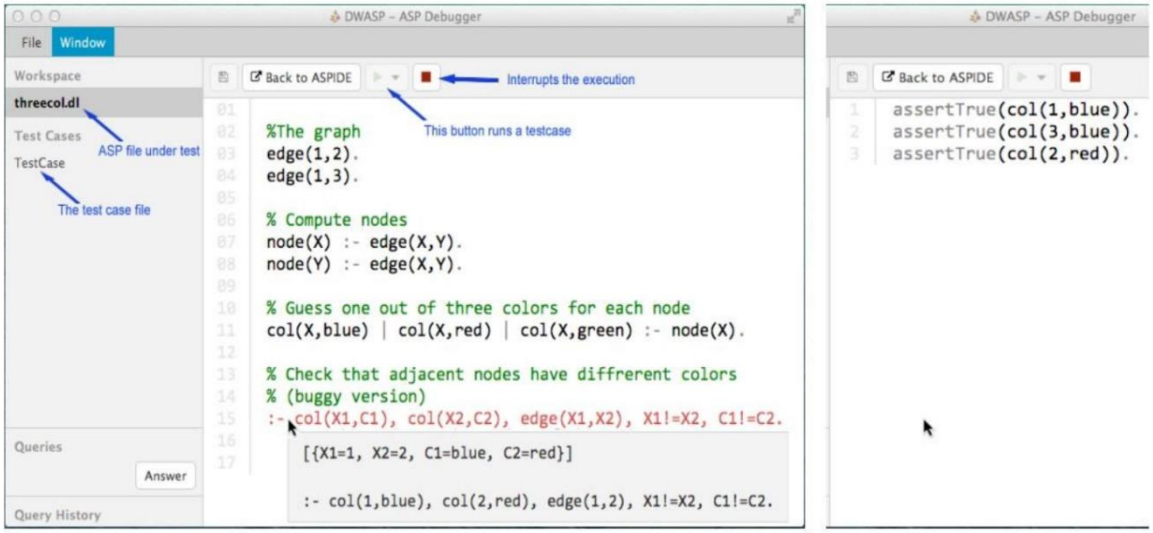


Figure 2.6: Error found using Stepping (Fandinno and Schulz 2019, 68)

Figure 2.5 shows the debugger stepping through the simple program from the start, we see that assigns the atom “a” to a positive truth assignment and the atom “b” to a negative (false) assignment. Though in Figure 2.5 we see that once we move to the next rule (line 2) the action of stepping through the rule fails. This is because the debugger detected that stepping through rule 2 leads to there being no answer set. Unlike both Ouroboros and Spock, which provide explicit explanations for inconsistencies, stepping shows the user by means of detailing truth assignments to atoms. It also does not make any recommendations into making a program become consistent. Furthermore, stepping does not require the user to input an intended answer set before it starts, which allows stepping to be particularly good at debugging both inconsistent and consistent logic programs.

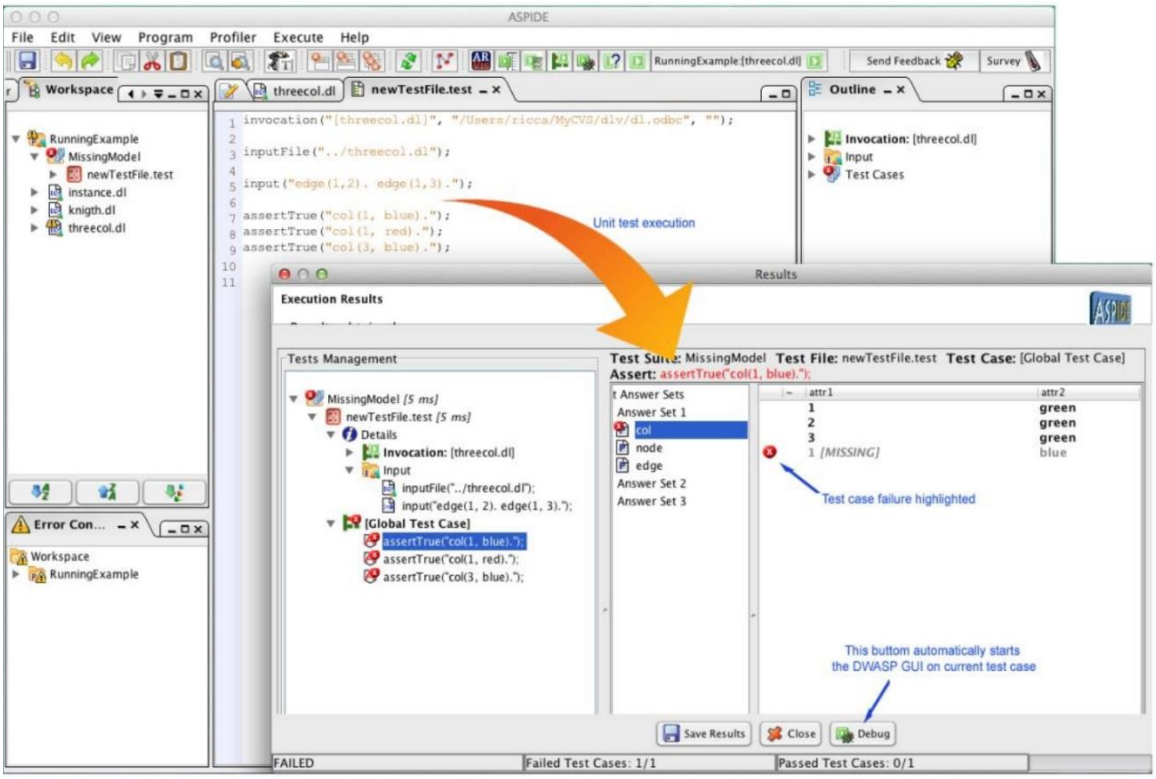
DWASP System

The DWASP System (Dodaro, et al. 2015), like Ouroboros, takes into account logic programs that make use of variables (i.e., non-ground programs). The goal of DWAPS is to allow for interactive localization of bugs in non-ground programs, wherein a set of rules that are involved in the bug(s) inconsistency can be pointed out to the user. DWAPS works by "querying the user to find relevant explanations of inconsistency to non-ground programs" (Dodaro, et al. 2015). These queries include some program P, background on the program's intended answer set, and some test cases. Unlike Ouroboros and Spock though, DWASP makes use of the solving process of ASP by using the solver WASP (Alviano, et al. 2013) to find such inconsistencies. Figures 2.7 and 2.8 show the DWASP system Graphical User Interface (GUI) being implemented to debug a simple logic program. We can see where the user adds the test cases as well as where the program fails w.r.t to given test cases, along with an explanation for why a rule is unsatisfied.



(a) Main window of the DWASP-GUI. (b) A test case in DWASP-GUI.

Figure 2.7: DWASP finding rule causing inconsistency w.r.t test cases.



(c) Unit testing and debugging (interaction with ASPIDE).

Figure 2.8: DWASP explanation for error produced.

Kara

Kara is a "System for visualizing and visual editing of interpretations for Answer-Set Programs" (Kloimullner, et al. 2011). Kara allows for visualization and editing in two ways. A custom-made visualization program is used to specify how the visualization should look with respect to a given interpretation (Figure 2.9 below). This gives the user the most control over how they wish to represent their program interpretation visually. On the other hand, Kara also allows for a more generic form (see Figures 2.10) where the user does not have to define their graphical elements. This form does not require the user to make a domain-specific visualization program, it instead represents the answer set as a labeled hypergraph whose nodes represent individuals and edges the literals from the

interpretation which are labeled by the predicate. Furthermore, edges that share predicates are shown to have the same color.

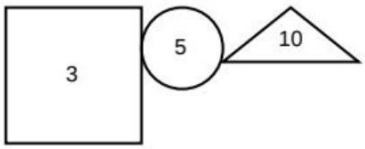


Fig. 7. Output of the visualisation program in Example 3.

```
% Generate a graph.
```
$$visgraph(g). \quad (47)$$
```
% Generate the nodes of the graph.
```
$$visellipse(X, 20, 20) :- node(X). \quad (48)$$
$$visisnode(X, g) :- node(X). \quad (49)$$
```
% Connect the nodes (edges of the input).
```
$$visconnect(f(X, Y), X, Y) :- edge(X, Y). \quad (50)$$
$$vistargetdeco(X, arrow) :- visconnect(X, \_, \_). \quad (51)$$
```
% Generate labels for the nodes.
```
$$vislabel(X, l(X)) :- node(X). \quad (52)$$
$$vistext(l(X), X) :- node(X). \quad (53)$$
$$visfontstyle(l(X), bold) :- node(X). \quad (54)$$
```
% Color the node according to the solution.
```
$$visbackgroundcolor(X, COLOR) :- node(X), color(X, COLOR). \quad (55)$$

Figure 2.9: Kara custom visualization editor. (Kloimullner, et al. 2011)

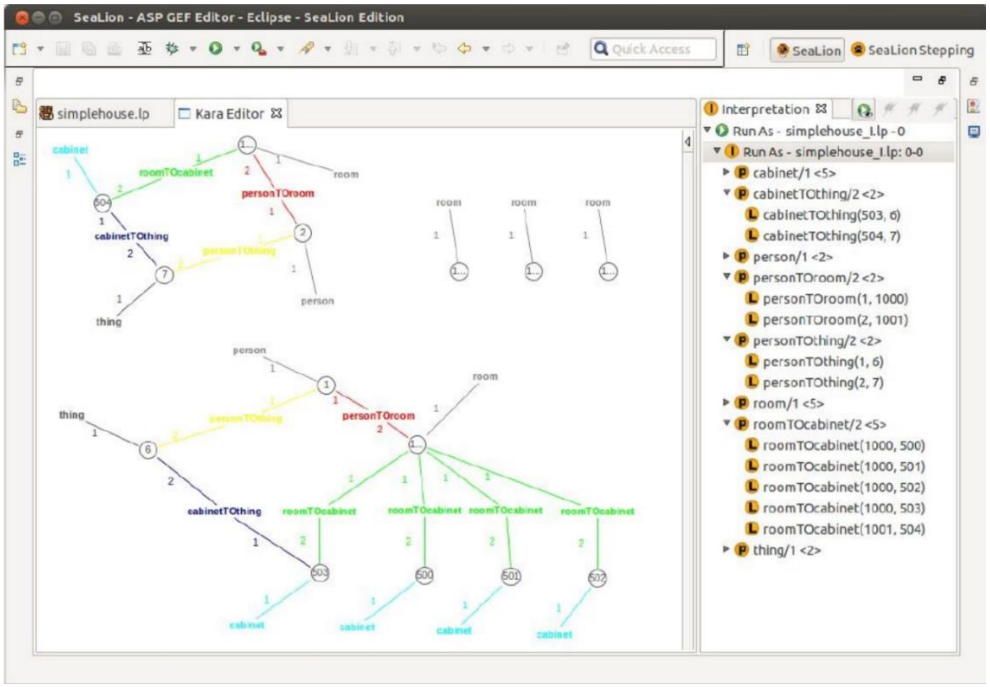


Figure 2.10: Kara's generic visualization. (Kloimullner, et al. 2011)

In Figure 2.11 a simplified version of the family problem is used to showcase Kara's generic visualization view. Here, the 1s/2s represent the order or mapping that the person belongs to with respect to the rule that the person relates to (e.g., Todd is the second variable when it comes to the father relation but is the first variable in the child relation). Kara allows for a multitude of customization options as well as the ability to export the visualization as an SVG (Scalable Vector Graphics) file.

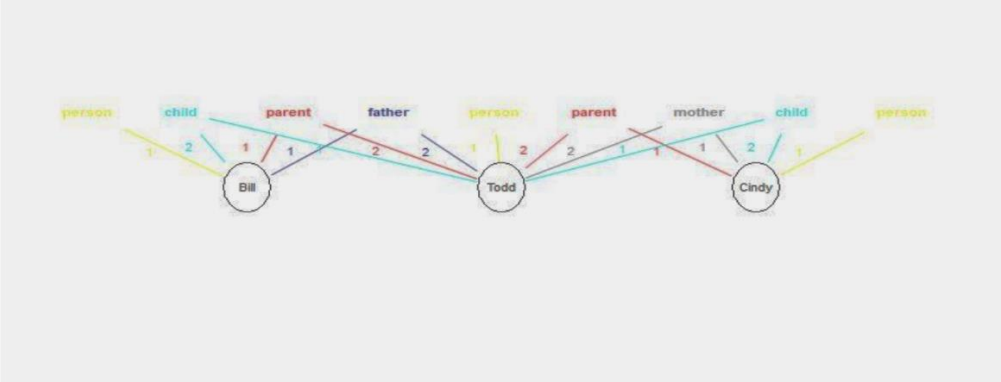


Figure 2.11: Kara generic visualization with family problem.

Discussion

Each system of debugging reviewed above all contribute integral parts to creating a debugging methodology suited for ASP, though none have put visualization of reasoning and the program as the foundation to their systems. Furthermore, each system does bring with it its own shortcomings that, if improved upon, can lead to the creation of a more versatile ecosystem of debugging in ASP.

In both the Ouroboros and stepping systems, the debugging view panel is very small and limits the amount of information shown to a programmer at once. This could become a problem when debugging larger programs. In creating a visualization of reasoning, this is one aspect to be aware of; how much information is shown to the user and how to decide what is important vs unimportant data.

In Ouroboros, the “Debugging/explanation” tab is not entirely clear to the user. It repeats and highlights some rule within the program that is causing some inconsistency

without giving some detailed explanation for as to why. Furthermore, it would be useful to have a clearer definition of what is meant by the “interpretation” tab.

In stepping, the use of atoms to show if a sequence of rules is valid does not clearly help the programmer in showing why the rule is incorrect or does not give the intended results. It would be best to define what is meant by positive and negative atoms. The use of atoms might be better suited if they were visualized.

The downsides of Kara relate to the options given to the programmer in separating the functionalities between a generic version and a more complex custom visualization. The generic version lacks essential customization to the programmer, which is more than double in the customizable version. This creates a burden on the programmer to have to learn how they write the code in the customizable version if they desire more functionality that the generic version lacks. This gatekeeping of the program significantly reduces the applicability of Kara in a more mainstream manner. Furthermore, Kara builds a visualization upon the interpretation that is given, regardless of whether it is sound or not. So even if an error exists, Kara will not discern this to the programmer (at least not in a meaningful manner).

Throughout all systems of debugging, there is significant lack of documentation to aid users in using the systems outside of white papers. This lessens the chances of each

system becoming more widely used adopted throughout the logic programming community.

Summary

In this chapter, we present a review of current literature within the area of debugging in ASP as well as a discussion about the current state of the topic. While there are a decent number of systems to choose from, there currently does not exist one that can be considered a cut above the rest. Each system has its own strengths and weaknesses that prevent it from dominating the rest of the field. When examining these systems as a whole though, the common shortcoming present was the inability to create a system for debugging that was easy and clearly understandable to the programmer. Each model required that the programmer have an extensive understanding of the intricacies of the system in order to perform basic operations. That being said, plenty of useful solutions do exist for debugging logic programs, and certain ideas will need to be carried over when creating a visualization of reasoning for it to succeed.

CHAPTER III

VISUALIZATION OF REASONING

In this chapter, we will present a few proposed solutions to creating a visualization of reasoning, in order to better illustrate and assist programmers in debugging a program written in ASP. Each system will present its own unique medium in attempting to create a visualization that helps the programmer in debugging their program and visualizing the execution of the program. That being said, all proposed solutions were constructed with the following in mind:

- The most important goal for visualization (whether it is in some guided manner or through a tree/table representation) is to clearly show the substitution of facts into rules when they lead to some inconsistency within the program.
- When showing the list of inconsistencies that occur in a program, it is important to ensure that the inconsistencies are shown to the programmer in an intuitive manner (see discussion in the previous section on issues with current debugger systems).
- From the list of given inconsistencies, the programmer should be able to click on any one of them and be given a visualization in a separate tab/window showing the reasoning behind the error along with potential solutions.

Two example buggy programs were also constructed to assist in the creation of the proposed solutions, as well as to show off their capabilities. They are given below with discussions on how they helped form the systems.

Extinct Problem

The following program is written in SPARC (Balai, Gelfond and Zhang 2013) as the ASP language. The program details an environment in which there are animals (eagle or snake) and the animal object has certain relationships that are defined. The first relationship is the *feedsOn(X,Y)* relation, which states that some animal X feeds on another animal Y, where X and Y are any elements within the animal object. The second relation, *extinct(X)*, defines that animal X is to be taken as being extinct. With these objects and relations, we are able to then create knowledge and rules which dictate what can and cannot be found in an answer set. The knowledge, i.e., what is to be taken as fact, that is given is the eagle feeds on the snake and that the snake is extinct. As for the only rule that is given for the program: an animal X is said to be extinct if it feeds on another animal Y that itself is also extinct. Our query for the program is whether or not the eagle is extinct given all the knowledge and rules of the program. The following is the program written using SPARC notation.

- **Objects:**
    - animal = {eagle, snake}
- **Relations:**
    - feedsOn(X, Y) % *Animal X feeds on animal Y.*
    - extinct(X) % *Animal X is extinct.*

- **Knowledge/Rules:**
    - feedsOn(eagle, snake). % *The eagle feeds on the snake.*
    - extinct(snake). % *The snake is extinct.*

    % *Animal X is extinct if it feeds on some animal Y, itself is extinct*
    - extinct(X) :- feedsOn(X,Y),
      extinct(Y).

As the eagle feeds on the snake, which is said to be extinct, we expect our query pertaining to the eagle's status to also be extinct. While this may seem like a straightforward example, it does allow for extensive research into how a clear visualization of reasoning about, in this case, why it is correct to declare the eagle to be extinct.

Buggy Family Example

The second example widely used to build our systems for visualization of reasoning is the buggy family problem. In this example, we introduce an artificially bugged program, so that we may show how visualization of reasoning might be useful in the debugging process. The program deals with the creation of a system consisting of two objects: person and gender. The objects have four relations with each other: genderOf, father, mother, and child. The rules state how a person can be defined as a parent, child, or brother as well as who cannot be a father.

- **Objects:**
    - person = {...}
    - gender = {male, female}

- **Relations:**
    - genderOf(person, gender) % *Person X is of gender Y.*
    - father(person, person) % *Person X is the father of person Y.*
    - mother(person, person) % *Person X is the mother of person Y.*
    - child(person, person) % *Person X is the child of person Y.*
- **Knowledge/Rules:**

    % *Person X is the Parent of person Y IF X is the the Father of Y*

    - parent(X, Y) :- father(X,Y).

    % *Person X is the Parent of person Y IF X is the the Mother of Y*

    - parent(X, Y) :- mother(X, Y).

    % *Person X is the Child of person Y IF Y is the the Parent of X*

    - child(X, Y) :- parent(X, Y).

    % *Person X is not the father of Y if they are not the father*

    - -father(X, Y) :- person(X), person(Y),

      Not father(X, Y).

    % Person X is the brother of Y if X is male and they share the same F/M.

    - brother(X, Y) :- genderOf(X, male),

      father(F, X),

      mother(M, X),

      father(F, Y),

      mother(M, Y),

      X != Y.

Within this program we have inserted a piece of knowledge that states Sarah is the father of Billy, thus our artificial error introduced to the program. This of course is contradictory information, as one of our rules, which is a constraint that dictates it is impossible for some person X to be the father of another person Y if the gender of person X is female. When attempting to get an answer set for this program and programmer might either encounter an error due to the contradiction or an answer set that contains father(sarah, billy), which should not be included. These two examples of different types of errors the programmer might encounter allow us to demonstrate how we can use a visualization of reasoning to show how the contradiction occurs as well as a potential fix to make the program bug-free.

Having now established two different ASP examples, with various qualities that make for showcasing the potential that visualization of reasoning might present, we will now move to show the various systems up for consideration.

Firing-Method

Looking at how the stepping method works at a high level, in terms of being able to go through a sequence of a code and see actions of execution that occur, as well as thinking about being able to see a movie frame by frame (rewrite this part), we have created what is locally known as the “Firing Method”. This model by giving the programmer a visualization of their program when running against some given query. The aim is to allow the programmer the ability to see how knowledge/facts are “fired” into rules and whether they are able to satisfy the head of the rules, which would mean that our answer set, based

on the query, is valid. We visualize knowledge being fired into the rules by means of lines that connect a fact to its corresponding placement within a rule. We then represent this rule filled with knowledge to be valid by turning the body green. If the body of the rule, then proceeds to satisfy the head we again fire a valid result (still in green) to mean that whatever query is being asked has been validated with an answer. If the body does not satisfy the head, then the rules changes to red and so too does the query. Figure 3.1 through 3.4 showcases our proposed solution in action on a query for whether the eagle is extinct from the family problem outlined earlier.

Before we showcase the proposed system, we have also produced what we are calling a "script" that details the problem and goals to be achieved from this proposed solution. Having such an outline allows for us to clearly come up with the visualization of reasoning needed to be useful to programmers. We use this outline to judge how well the firing model can be, as well as how it stacks up with current debugging systems revied in the previous section. The following is a script to demonstrate the firing animation procedure, with reference to the extinct problem.

Problem: We wish to know whether the eagle is extinct using defined rule(s) and given facts.

Goal: Produce an animation that showcases how a set of given atoms will result in the "firing" of a rule.

Assumptions/Rules:

We can assume the following facts:

1. extinct(snake). # This snake is extinct #
2. feedsOn(eage, snake) # The eagle feeds on the snake #

We also are given the following rule:

1. extinct(X) :- feedsOn(X, Y), extinct(Y).
    1. Object X is extinct if X feeds on object Y and Y is also extinct.

Process: We wish to convey the following process in a visualization

1. Show that feedsOn(eagle, snake) is true or given
2. Show that extinct(snake) is true or given.
3. Using feedsOn(eagle, snake) we match it feedsOn(X, Y) in the rule. As a result of the matching: X is replaced with eagle and Y is replaced with Snake. The values of X and Y are now propagated to other atoms in the rule.
    1. feedsOn(eagle, snake) could now fade away to the correct matching of values.
4. Using extinct(snake) we match it to extinct(snake) in the "new" rule. Using this matching of values, extinct(snake) can now fade away as well.
5. The body of the rule being satisfied is visualized by the matching and fading of atoms in the rule (i.e., all variables have correct values based on facts). That the head of the rule shall now be satisfied is visualized by making the head to becoming green.

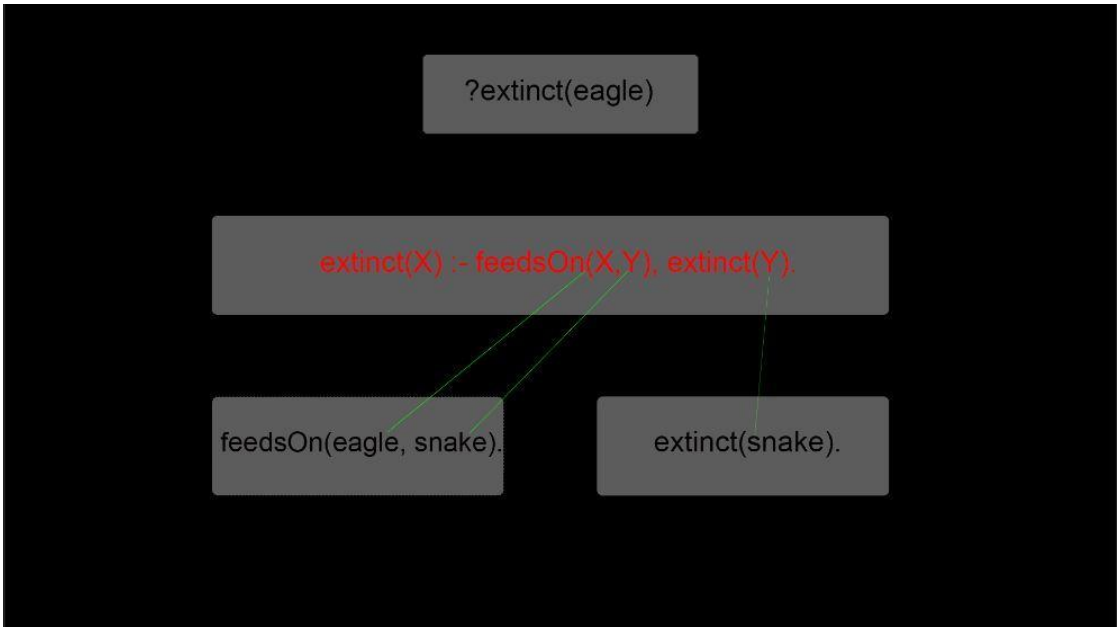


Figure 3.1: Demo of firing method firing facts into rule.

Figure 3.1 shows the beginning stages of our system. Here, the first thing that would happen when a programmer wishes to either visualize a query or need assistance in the debugging phase, would be to fire the relevant fact(s) into the corresponding rule(s). In ASP this firing represents the substitution of variables with facts/knowledge. As the eagle gets fired in the feedsOn(X,Y) it takes the spot of X and all occurrences of X now correspond to the eagle. This same applies to the snake and the variable Y. The visualization was built by making the execution of some query/portion of the program to be as clear as possible, so we felt it was best accomplished by showing in a bottom-up approach. By that we mean, starting from our most basic components (i.e., facts/knowledge), we build-up to the next part in the hierarchy (i.e., rules with variable

substitution) and then to the overall result of the goal. This gives the programmer a clear stepping of what they are looking at at any particular point.

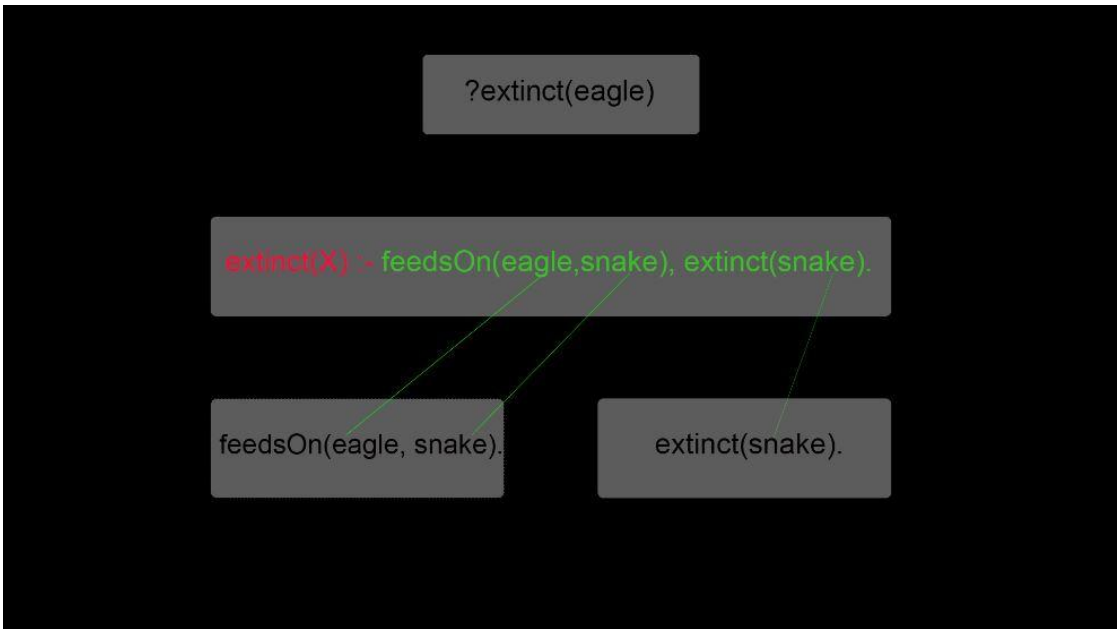


Figure 3.2: Facts satisfying arguments of body in demo.

Figure 3.2 gives the step in our system where, after the variable substitution, the body needs to be confirmed to have accepted its new arguments. This is shown to be true by means of visual cue, i.e., changing the text color of the body from red to green. Since it has not been confirmed that the entirety of the rule has been satisfied, the head does not change from being written out in the red text. Now the visualization will confirm whether what has been fired into the body of the rule satisfies its head, thus making the entire rule with substitution of variables valid.

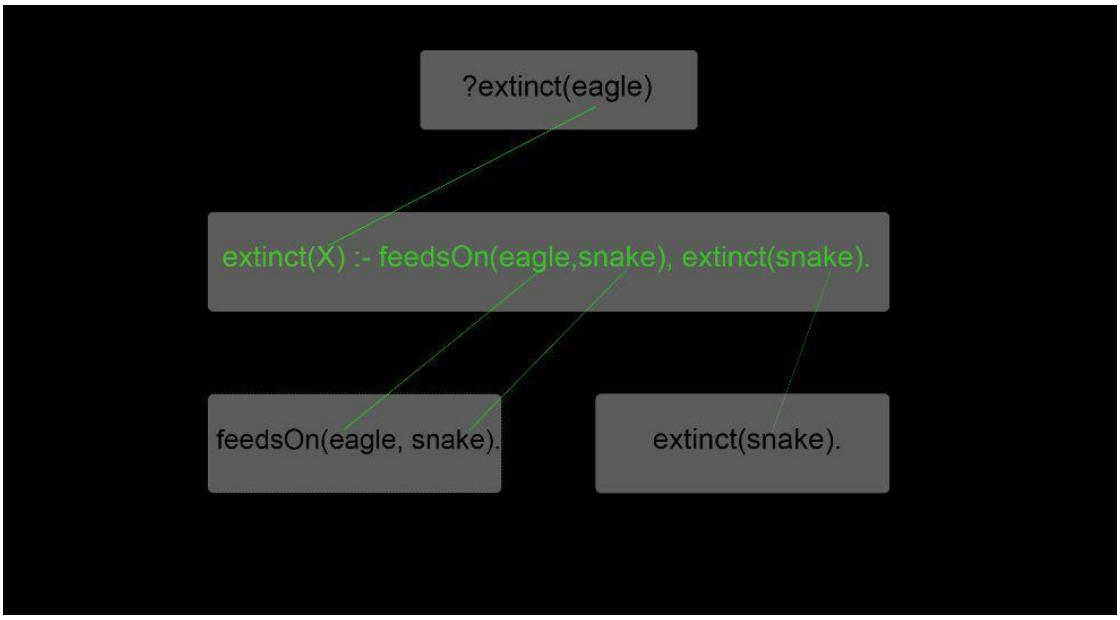


Figure 3.3: Body satisfying head of rule.

In conjunction with the process described in Figure 3.2, Figure 3.3 shows the final step in this stage where the contents in the body of the rule satisfy what is in the head of the body. We also continue to use the visual cue or changing the text color from red to green to show the rule being satisfied. If the body did not satisfy the head, then the entire rule would be changed to the red text in this step. Once we know whether the rule has been satisfied given the facts, we then fire the result into our query.

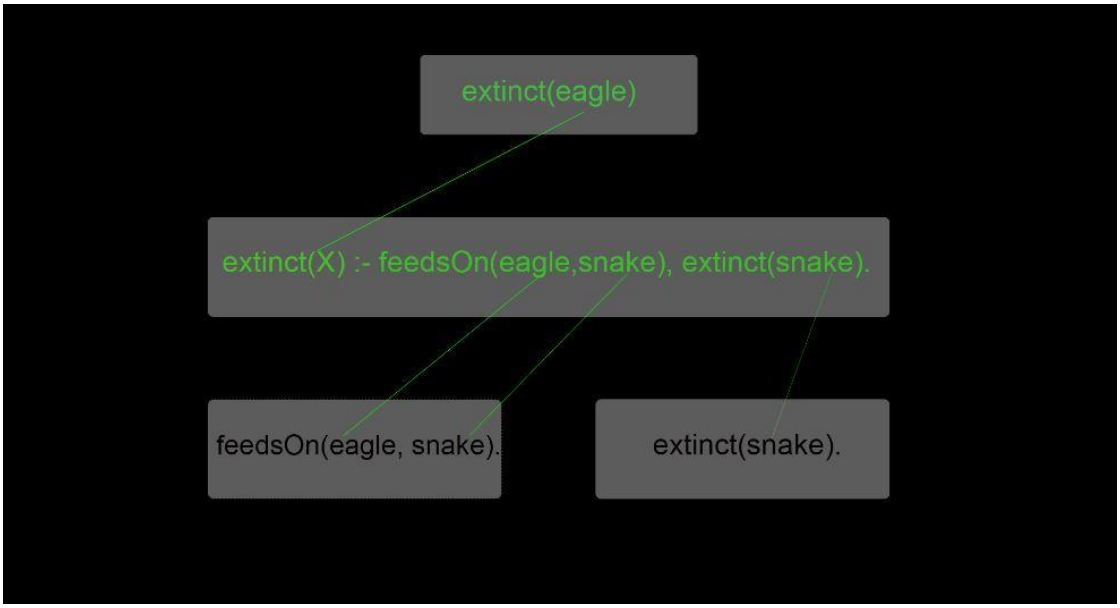


Figure 3.4: Conformation of correct query result.

This last step in our proposed system is given in Figure 3.4. Here we see that since the knowledge (i.e., feedsOn(eagle, snake) and extinct(snake)) satisfy our corresponding rule, we can then determine that our query is "?extinct(eagle)" is true. We remove the "?" which signifies a query and replace the X with the eagle to represent what the results of previous steps have come upon.

To the programmer, Figures 3.1 through 3.4 all would be presented in a more movie-like manner, likely in the form of a GIF. This will allow the programmer the ability to see every major step in the reasoning process in a fluid and coherent visualization. The firing method has the greatest potential to be a viable solution to visualization of reasoning in the efforts of aiding debugging, in that it is simple enough for any user to understand what is going along with the most straightforward design scheme. Furthermore, in the

context of helping students learn the intricacies of ASP. Teachers would now have the ability to add a layer of visualization when teaching about variable substitution and what makes a rule satisfiable when using ground literals as we saw in the extinction example program. The one downside to this proposed model is that it is currently intended to work on one rule at a time, so if a programmer wishes to view a large set of rules and facts, they will need to go through it rule by rule to see their visualizations.

Table Model

Instead of creating an animation to give a better understanding of the reasoning process and helping create an effective debugging assistant, another option is through the use of tables. The idea behind this proposed table model is that the programmer would be able to select any original rule in their program and be given a drop-down table of all available atoms to populate the rule. These atoms are the facts that are defined by the program. Once the user has selected an atom to fill into the rule, the values/literals of that atom are then mapped (grounded) and propagated to all other instances of those variables in the rule. If the selection of atoms causes the rule to not be satisfiable, then the atom will be underlined in red that the programmer can click on and be taken to where the atom violates some portion of the program w.r.t attempting to ground to the rule. Figure 3.5 gives an example of the table model being implemented using the family example given earlier in the section.

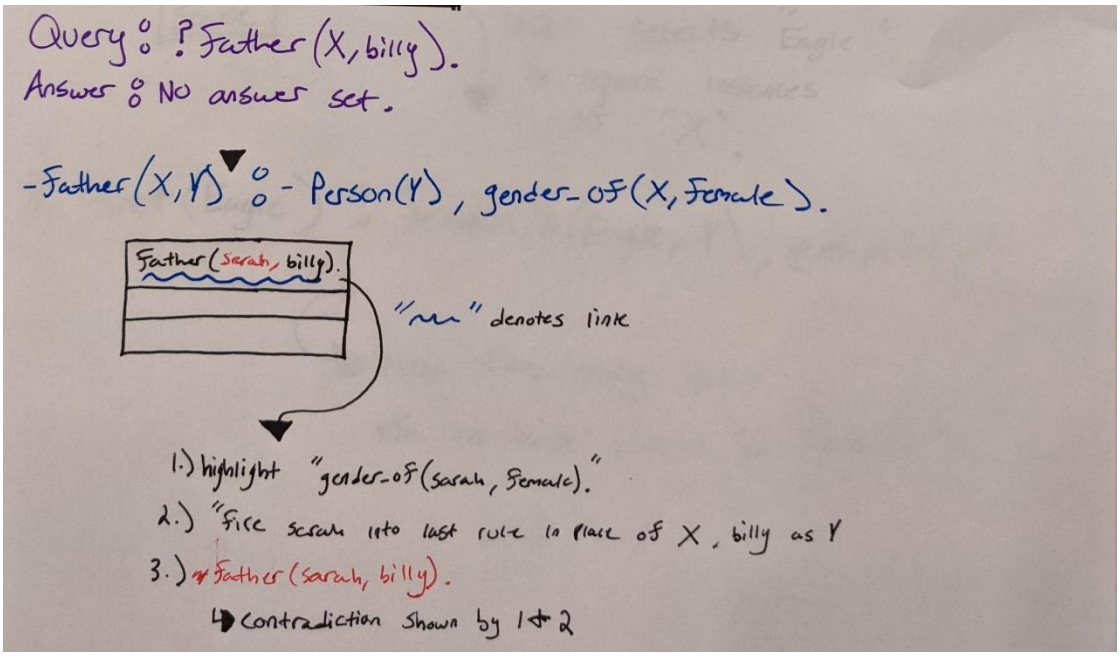


Figure 3.5: Simple Table model example.

In Figure 3.5 a simple example of the proposed table model is given. Using the family problem for this example, a query is created asking who the father of Billy and the returned answer is, “no answer set”. The scenario envisioned is the programmer wishes to see why the result of no answer set was returned and what parts of the program lead to this conclusion. Having an understanding for them, they navigate to the constraint rule, -father(X,Y) and select the drop-down to fill the rule with the atom, father(sarah, billy). In this drop-down, the programmer will see that the atom is underlined to indicate a warning, with the Sarah literal being written in red. By clicking the warned rule, the model would then highlight corresponding rules and facts that contribute to the warning. In this case, the programmer would see the fact, gender_of(sarah, billy) and the rule father(X,Y) be

highlighted since these two are what cause the contradiction and inconsistency for the program.

The intended goal for the table method is for it to be an aid in debugging when the user has a lot of initial facts. Giving the programmers the ability to quickly check if relevant facts keep the program consistent when assigning them to rules provides a convenient first defense in keeping their programs consistent and as intended to the programmer. Furthermore, the table model also allows for a more interactive format to the user, as opposed to the firing animation, since it requires the user to select what it wants the model to show execution for.

SLD-Table Model

Taking inspiration from another aspect of logic programming, the idea of using SLD resolution tress in unison with aspects from the proposed table model is explored. As the SLD-tree showcases all possible SLD-derivations of a specific goal, we can see which paths in the tree lead to success clauses as well as those that end failure. Adding the aspect of having a table available at each point in the tree's path allows us to map out if a query/goal will lead to success or failure based on given facts. Figure 3.6 gives an example of how our proposed model might look.

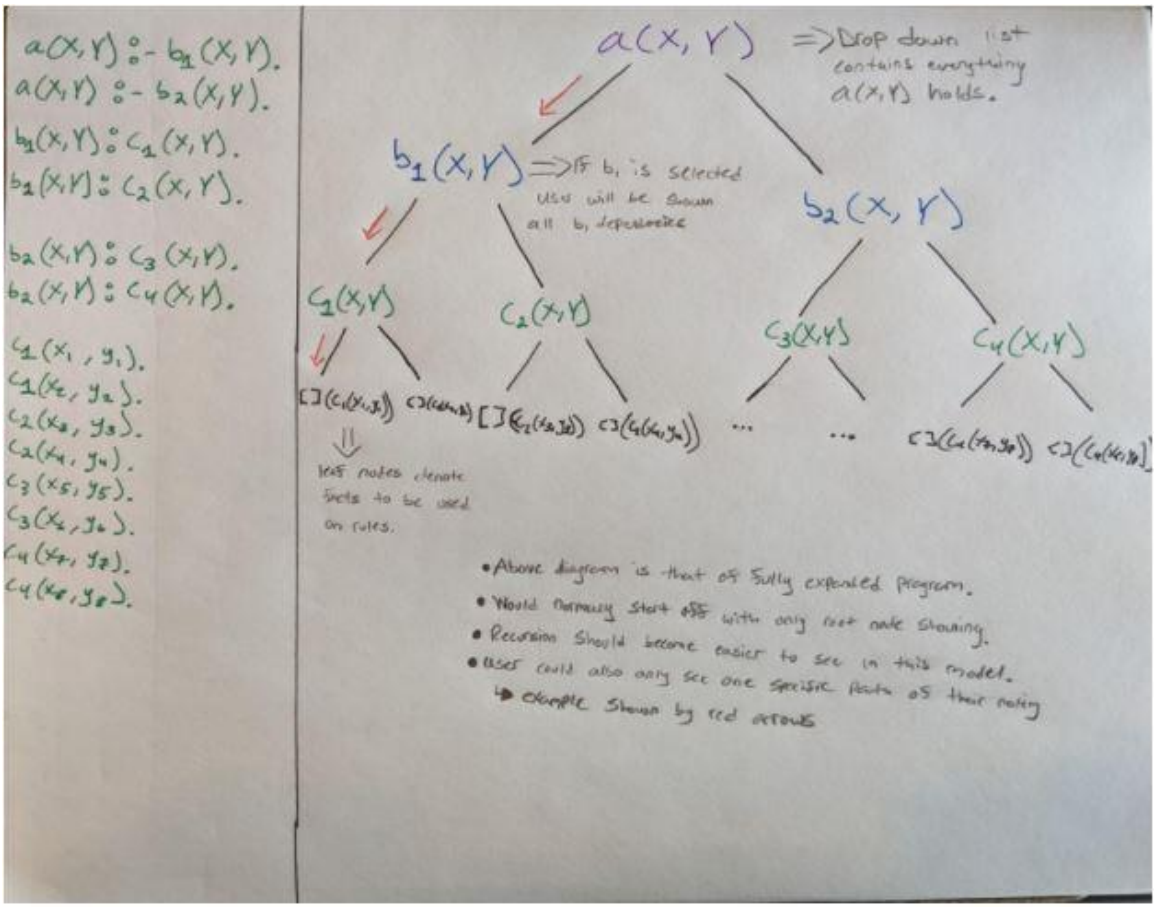


Figure 3.6: Example of SLD-Tree-Table Model

In the above figure, we can see the tree starts with our root of the atom a(X, Y) and from here the user can select everything that said atom contains. Once the user selects what they wish to fill into our root node, its children's nodes will then begin to populate based on rules and other facts. The example above is one of a program that has been fully fleshed out, meaning the entire search space of the query is shown. Initially, only the root node will be shown, and the programmer can choose to either only see a specific path of the tree (mimicked as red arrows in figure 3.6) or the entire search space of the given query.

```
person(john).
person(sam).
person(alice).
person(bill).
person(bob).
person(andy).
person(sarah).

gender(male).
gender(female).

father(john,sam).
father(john,bill).
father(bill, andy).

mother(alice,sam).
mother(alice,bill).
mother(sarah, andy).


gender_of(john,male).
gender_of(alice,female).
gender_of(sam,male).
gender_of(bill,male).
gender_of(andy, male).
gender_of(sarah, female).


parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).

child(X,Y) :- parent(Y,X).


%% This closed-world assumption for father(X,Y)
%% subsumes the above two rules.
%-father(X,Y) :- person(X), person(Y),
%                not father(X,Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Z,Y),
                 ancestor(X,Z).
```

Figure 3.7: Alt version of family problem

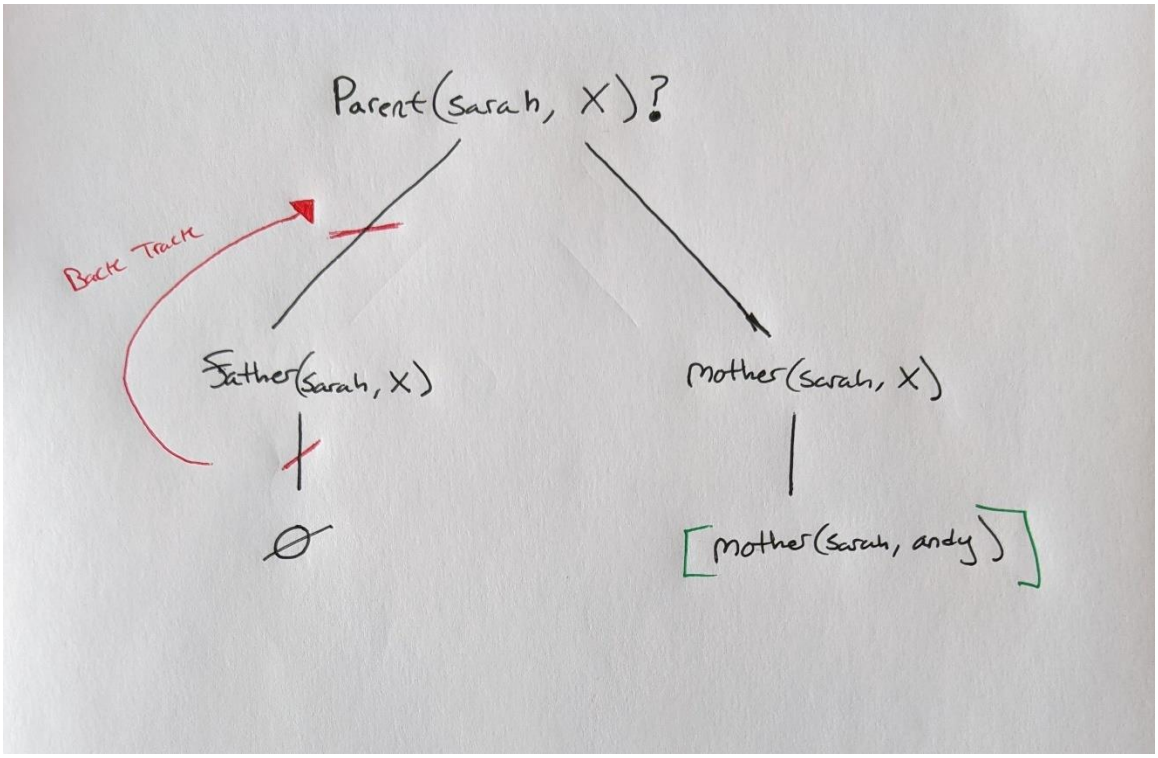


Figure 3.8: SLD-tree Example on alt. Family problem

The main goal, overall usefulness, found in this proposed system is that it allows for a clear visualization of where atoms applied to rules begin to make the program inconsistent. This is due mainly in part to how backtracking can be seen so effortlessly in a tree system. Figure 3.7 gives an alternate version of the family problem previously outlined. In this new version, we have added people to our system along with relations among each other (e.g., John is the father of Sam). Figure 3.8 gives an example of how our SLD-tree variation might look when attempting to figure out (query) who Sarah is the parent of. We place, "parent(sarah, X)?" as the root of our tree to signify it being our query and establishes our goal. From there the programmer is presented with two paths to examine for the consistency of the parent rule. The two paths are the one where the parent

rule is fulfilled if Sarah is the father of someone and the path where sarah is the mother of someone. Going down the left path (father relation) we see there is no answer set since it is impossible for Sarah to be the father of anyone, as defined by the program, so this path is closed and denoted by an empty set. Furthermore, the visualization would then cross out the paths by red lines to indicate failure and being the backtracking process. After backtracking we move to mother relation where we can show that the goal succeeds in finding that Sarah is the mother of Andy. Looking at the tree as a whole, we can see where the reasoning process clearly being displayed from failure to success. This should help programmers be more aware of what changes to atoms they create or how these atoms affect the consistency of a program when applied to a rule.

Discussion

From these three proposed systems for visualizing reasoning across ASP, we can see there are certainly benefits to having such a method to examining programs. This is especially true for ASP and other non-procedural programming paradigms, where one cannot simply rely on the order of how the program is executed in order to debug the system. Furthermore, having the ability to visualize the reasoning process behind a part or all of one's program allows for a deeper understanding of the semantics of the language (or any topic if applied in a more general manner). The idea of being able to expand creating a visualization of reasoning to apply to any subject matter (e.g., Mathematics, Chemistry, Biology, etc.), to better help students, learn the material in a more standardized manner is certainly one of many applications for these systems.

On the topic of which proposed system seems to have performed the best, since there are all just proposed and none were implemented past the design phase, it is to be considered highly subjective as to what system performed better overall. However, using the standards and goals laid out in the previous section, as to what will make a good visualization of reasoning, the firing method shows the most potential to become a quality system in the future. Its ability to give a thorough visualization of the entire query execution process, while still being able to easily show when things begin to break down is what sets it apart from the other two proposed systems. Furthermore, having the firing method be an animation instead of the more static designs shown by the table and SLD-tree models, adds for a more polished visualization to the user, which should therefore increase their readiness to use the system more often. The firing method also offers itself to be a potentially better solution given the fact that, if it were to be expanded to being used to help students understand various subjects, the animation of seeing things is certainly more approachable to younger students. This as opposed to having them look at a table of things and even viewing from a tree structure point of view.

When it comes to examining the other two proposed models, while they both offer a unique way to visualize reasoning about a program and its rules, they as bring with them many downsides that the firing method did not. The table method's biggest upside is that is the best solution when dealing with a program that contains recursive elements, as seen with the alternate family problem in Figure 3.7. Since the table method has the ability to link back to other rules and facts in the program, it can easily show that when a rule is

recursive it needs to be linked with itself when comparing atoms that are used in a query for that rule. This is something that cannot be easily done in the firing model and almost impossible to do in the tree model. However, the table model's biggest downside lies in its inability to generate a captive visualization into the reasoning process of some query or part of the program. It is akin to the more traditional guided model of debugging, in that the programmer walks through each step of the goal they are trying to solve. Even though the user would be able to see each step they have made and know when something is not consistent with the way a rule or fact is defined, it still might not be as groundbreaking as the firing model could be. Similar issues arise with the SLD-tree model as well, especially when it comes to handling negation (e.g., -father(X,Y)…) and recursion. The SLD-tree has no meaningful ability to handle constraints like the example offered of it being impossible for some X to be the father of Y if some rules are satisfied. Being unable to visualize an important construct in ASP presents a significant roadblock in this method being a viable system to look further into production. The proposed SLD-tree model does present the opportunity to view the entire search space of a query that causes an error, which is especially useful when dealing with a program that is very long.

Summary

In this chapter, we proposed three different models for attempting to create a visualization of reasoning in ASP/logic programs. We introduced the firing model, which allows users to see an animation of the program running under some goal or query. The user sees the code run from the grounding of facts to variables in rules to seeing whether these facts satisfy the rule and therefore the query. This model was by far the best model proposed in terms of potential to being widely used if fleshed out more and implemented into an IDE.

The table method was also proposed, and it offers the ability for users to walk through some execution of code of query to be debugged. The user is able to select atoms to ground to rules and see whether they are compatible given how the rule is defined as well as other rules and facts that are linked with the initial rule. The greatest strength of the table method was in its ability to visualize recursion in a manner that the other two systems were unable to do.

Lastly, we proposed the SLD-tree model where users can examine the search space of the program when examining a goal/query, or a single path along the tree. This model would seem to work best on larger programs where it is needed to see the entire search space coherently is most vital. However, this model struggled when trying to account for recursion along with visualizing how constraint rules work in ASP.

CHAPTER IV

CONCLUSION

We have studied recent literature pertaining to debugging techniques and propose three different models in the hopes of building up visualization of reasoning as a technique that will elevate debugging in logic programming. The first model, and the one with the most potential, is the firing system. The second system that was introduced deals with the proposal of a table method. The last system to be introduced as a way to visualize reasoning is the SLD-tree table model.

On Visualization of Reasoning as a Debugging Technique

We believe that visualization of reasoning can be a beneficial addition to debugging logic programs, especially if following some variation or improvement to the firing model. Having the ability for the programmer to see their program come to life in a manner where they can clearly see why one variation of atoms grounded to rules works, but another variation does not can greatly increase the user's understanding of their work. It also allows for a greater understanding of ASP and logic programming as a whole, given the fact that traditional debugging methods are not as reliable in non-procedural languages. This greater understanding of a language through visualization of reasoning provides additional applications for the subject matter. The hardest challenge for creating a visualization of reasoning in ASP seems to be ensuring that all the intricacies of language can be exported to a visualization. This can be seen as a problem when looking at how the SLD-tree table method was unable to translate constraint rules and recursion into its respective method of

visualization. Another challenge that future research into the topic will run into will be ensuring that what visualizations do meet pre-defined goals, as to what makes a good understanding of reasoning, isn't too complex that the average user needs to go and do further research about the program. These seem to be the two biggest barriers that a more widely used visualization of reasoning program would run into.

On Applications of Creating a Visualization of Reasoning

Besides being a useful debugging technique for ASP, a visualization of reasoning can be further extended into teaching students various subjects in a more streamlined manner. The idea is that you treat a subject the same way you would an ASP program with atoms, rules, predicates, constraints, and facts. By building them up in this manner, you create a streamlined way to teach any subject, given that you have the capabilities to visualize the reasoning behind what works in conjunction with each other and what does not. If you think about how chemistry as an example, you could easily see elements being atoms and rules being calculating mass and creating compounds from elements. Now giving students the ability to see these through some variation of the firing method, they could see how the atoms and rules come together to produce a valid outcome or see why some combination creates inconsistencies within the program. This all gets translates into how well the student will then understand the subject they are begin taught.

REFERENCES

Alviano, Mario, Wolfgang Faber, Carmine Dodaro, Nicola Leone, and Francesco Ricca. "WASP: A native ASP solver based on constraint learning." *International Conference on Logic Programming and Nonmonotonic Reasoning.* 2013.

Balai, Evgenii, Michael Gelfond, and Yuanlin Zhang. "Towards Answer Set Programming with Sorts." *Logic Programming and Nonmonotomic Reasoning.* Springer, 2013.

Brna, Paul. *Prolog Programming: A First Course* . Paul Brna , 1999.

Brna, Paul, Mike Brayshaw, Alan Bundy, Mark Elsom-Cook, Pat Fung, and Tony Dodd. "An overview of Prolog debugging tools." *Instructional Science* , 1991: 193-214.

Clocksin, William F, and Christopher S Mellish. *Programming in Prolog.* Springer London, Limited , 1981.

Dodaro, Carmine, Philip Gasteiger, Kristian Reale, Francesco Ricca, and Konstantin Schekotihin. "Interactive Debugging of Non-ground ASP Programs." 2015.

Fandinno, Jorge, and Claudia Schulz. "Ansawer the "why" in Answer Set Programming - A Survey of Explanation Approaches." *Theory and Practic of Logic Programming*, 2019: 114-203.

Gebser, Martin, Jorg Puhrer, Trosten Schaub, Hans Tompits, and Stefan Woltran. "spock: A Debugging Support Tool for Logic Programs under the Answer-Set Semantics." *Applications of Declarative Programming and Knowledge Management. INAP 2007, WLP 2007.* Berlin: Springer, 2009. 247-252.

Gelfond, Michael, and Vladimir Lifschitz. "The Stable Model Semantics for Logic Programming." Edited by Robert Kowalski, & Kenneth Bowen. *Proceedings of International Logic Programming Conference and Symposium.* MIT Press, 1988. 1070-1080.

Kloimullner, Christian , Johannes Oetsch, Jorg Puhrer, and Hans Tompits. "Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs." *International Conference on Applications of Declarative Programming and Knowledge Management.* Springer, 2011. 325-344.

Lifschitz, Vladimir. "What is Answer Set Programming?" *AAAI Conference on Artificial Intelligence*, 2008.

Oetsch, Johannes, Jorg Puhrer, and Hans Tompits . "Catching the Ouroboros: On Debugging Non-Ground Answer-Set Programs." *Theory and Practice of Logic Programming 10,* 2010: 513-29.

Oetsch, Johannes, Jorg Puhrer, and Hans Tompits. "Stepping through an Answer-Set Program." *Proceedings of the 11th international conference on Logic programming and nonmonotonic reasoning* . 2011.

"The SeaLion has landed: An IDE for answe-set programming-- Preliminary report ." *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management and 25th Workshop on Logic Programming (INAP 2011/WLP 2011).* 2011. 141-151.