DETECTING SUSPICIOUS INPUT IN INTELLIGENT
SYSTEMS USING ANSWER SET PROGRAMMING

by

NICHOLAS GIANOUTSOS, 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

Michael Gelfond
Chairperson of the Committee

Richard G. Watson

Joseph Nelson Rushton

Accepted

John Borrelli
Dean of the Graduate School

May, 2005
ACKNOWLEDGMENTS

Special thanks to Dr. Michael Gelfond, my advisor and my mentor. Your incredible patience and knowledge have amazed me since we first met. You have had a direct impact on the way I analyze and think about the world. From our time together I have learned a little bit about artificial intelligence and a lot about life. Thank you for investing so much time in me and for giving me the opportunity to do research. I would also like to express my special thanks to Dr. Nelson Rushton and Dr. Richard Watson for serving on my committee and for their detailed reviews and helpful suggestions.

I would like to thank the members of the Knowledge Representation Lab for their help and contributions. I greatly enjoyed the environment in which I worked and the students from many different cultures that represent the KR Lab. First, I need to thank Marcello for his work in CR-Prolog, and the time he spent explaining the functionality and limitations of the language. Next, I would like to thank Loveleen for her work with CRMModels, which improved the inference engine needed to perform this research. I’m also grateful to Sandeep for his help with \LaTeX and throughout the writing process. I enjoyed our 85 mile canoe trip down the Rio Grande. I would like to thank Greg for his encouragement, and I enjoyed the time spent together at Cricket’s discussing ideas, problems and solutions. Lastly, I would like to thank Ricardo for his discussions and deep thoughts on religion and philosophy.
I would like to thank my family for their support during my time in graduate school, especially my sister, Jamie, who provided encouragement and support throughout the entire process. I would also like to extend my appreciation to all my friends who came to the thesis defense. I know that without any background knowledge in this field, it was difficult to understand the majority of my talk. I really appreciated you guys showing support.

Finally, I would like to thank the people who take the time to read this thesis and expand on this research. I wish you the very best, and I hope that this work may be of use to you in the problems you attempt to solve.
TABLE OF CONTENTS

ACKNOWLEDGMENTS .................................................. ii

ABSTRACT ............................................................. viii

I INTRODUCTION ....................................................... 1

1.1 General Purpose .................................................. 1

1.2 What is an Outlier? ............................................... 2

1.3 Example of an Outlier .......................................... 2

1.3.1 Store Example ................................................. 3

1.3.2 An Intelligent Agent and His Methods of Reasoning .... 3

1.3.3 Simple Store Scenario ....................................... 4

1.3.4 Complex Store Scenario ..................................... 4

1.4 Statement of the Problem ...................................... 6

II BACKGROUND ....................................................... 7

2.1 Description of a Knowledge Base ............................. 7

2.2 Description of CR-Prolog ...................................... 7

2.3 Designing Knowledge Bases in CR-Prolog .................. 8

2.3.1 Informal Introduction to CR-Prolog ...................... 8

2.3.2 Minimal Set Inclusion ..................................... 10

2.3.3 Preferences in CR-Prolog ................................. 10

2.3.4 Minimum Cardinality ..................................... 11

2.3.5 Preferring Different Types of Exceptions ................ 12
ABSTRACT

When presented with bad information people tend to make bad decisions. Even a rational person is unable to consistently make good decisions when presented with unsound information. The same holds true for intelligent agents. If at any point an agent accepts bad information into his reasoning process, the soundness of his decision making ability will begin to corrode. The purpose of this work is to develop programming methods that give intelligent systems the ability to handle potentially false information in a reasonable manner.

In this research, we propose methods for detecting unsound information, which we call outliers, and methods for detecting the sources of these outliers. An outlier is informally defined as an observation or any combination of observations that are outside the realm of plausibility of a given state of the environment. With such reasoning ability, an intelligent agent is capable of not only learning about his environment, but he is also capable of learning about the reliability of the sources reporting the information. Throughout this work we introduce programming methods that enable intelligent agents to detect outliers in input information, as well as, learn about the accuracy of the sources submitting information.
CHAPTER I

INTRODUCTION

1.1 General Purpose

When people are presented with bad information, they normally make bad decisions. Even a rational person is unable to consistently make good decisions when presented with unsound information. The same holds true for intelligent agents. An intelligent agent is programmed with a general understanding of the world called a knowledge base. The agent uses his knowledge base in combination with observations to draw conclusions about the world. Observations may be observed by the agent, or they may be reported from outside sources which the agent sometimes accepts blindly.

The majority of intelligent systems designed in past years work reasonably well when sound information is input into the system. However, if at any point an agent accepts bad information into his reasoning process, the soundness of his decision making ability will begin to corrode. The majority of these systems do not employ methods for the purpose of filtering out erroneous information. Thus, the purpose of this work is to introduce methods that enable intelligent systems to handle potentially false information by designing software capable of detecting unsound information.

In this research, we propose methods for detecting unsound information, which we call outliers, and methods for detecting the sources of these outliers. With such
reasoning ability, an agent is capable of not only learning about his environment but is also capable of learning about the reliability of the sources reporting information. When an agent learns that a source of information is unreliable, the agent can then process the information accordingly. By revealing unsound sources of information, the agent is able to maintain the reliability of his knowledge base.

1.2 What is an Outlier?

Contained within an agent’s knowledge base are rules and constraints that govern the limits of acceptable behavior for objects within an environment. An outlier is informally defined as an observation or any combination of observations that are outside the realm of plausibility of a given state of the environment. Intuitively, such an observation should be inconsistent with the union of the agent’s knowledge base and all other reported observations.

1.3 Example of an Outlier

Let us look at an example in which unobservable information important to the decision-making process may be obtained from outside sources. This outside information allows the agent to make decisions within the environment, determine outliers in the scenario and gain insight in establishing the reliability of the sources.
1.3.1 Store Example

Normally the local store in town is open for business. However, the store may be closed for a few different reasons. The store closes on Sunday because the owner spends the day with his family. The store is also closed on holidays and on days when bad weather conditions exist. When considering taking a trip to the store, one can ask his neighbor, John, and his other neighbor, Bill, for information concerning the store being open or conditions that may cause the store to be closed. John and Bill normally give accurate information but may occasionally be mistaken.

1.3.2 An Intelligent Agent and His Methods of Reasoning

Let us introduce an intelligent reasoning agent into this environment. The agent understands that “normally the store is open.” Exceptions to this default include Sundays, holidays and bad weather conditions. If the agent knows that today is not Sunday, not a holiday and the weather is nice, he will conclude the store is open. The agent may also ask the neighbors for their input to gain knowledge about the environment, as well as to learn about their reliability. Since neighbors “normally tell the truth,” the agent believes the neighbors’ statements to be true unless their reports conflict with each other, or with other information the agent has received. When two reports are contradictory, the agent is forced to choose which statement is correct and which statement is an outlier given that his current knowledge is inconsistent. Given the chance, the agent will choose the situation that is “most likely to occur”
by choosing the model of the world with the minimal number of assumptions based on the occurrence of unusual events. Sources who contribute inaccurate information are referred to as being *problematic* with respect to their incorrect statements. Let us now consider a few scenarios from the store example to illustrate the informal reasoning used by the agent.

### 1.3.3 Simple Store Scenario

Suppose the agent observes nice weather on a Monday afternoon. The agent knows that normally the store is open, but he also knows that the store may be closed if today is a holiday. The agent asks his neighbors for information regarding this matter. If both neighbors say it is a holiday, the agent will conclude that the store is closed. If both neighbors say that it is not a holiday or simply state that the store is open, the agent will conclude that the store is open. In either case no outliers exist because neither source contradicted the other nor were their statements outside the realm of plausibility within the environment.

### 1.3.4 Complex Store Scenario

The agent is considering taking a trip to the store and would like to make sure it is open. Suppose the agent knows that today is not a holiday but lacks some other relevant information. The agent asks the neighbors John and Bill for information regarding this matter. Bill states that the store is open, but John says
the store is closed. Neither person states additional information to support their beliefs. Because John and Bill’s statements contradict each other, the agent is forced to make a decision. Believing John is problematic and assuming John’s statement is an outlier will stop the application of the default “normally neighbors tell the truth” and consistency will be restored to the knowledge base. Intuitively this means that the agent does not believe that the weather is bad or that it is Sunday. If the agent believes Bill is problematic and Bill’s statement is an outlier, the agent will also need to assume that it is either Sunday or that bad weather conditions exist in order to justify John’s statement. Since the first instance assumes the occurrence of one unusual event, John being problematic, and the second instance assumes two, Bill being problematic and either Sunday or bad weather, the agent will choose to believe the first instance because it assumes a minimal amount of unusual events. The agent concludes that John is problematic: John’s statement is an outlier, and the store is open.

After arriving at this conclusion the agent visits the store only to find it is closed. The agent also observes during his trip that the weather is nice. These two facts are then added to the knowledge base, and the agent concludes that John’s statement is accurate and Bill’s statement is an outlier, reversing his previous conclusion. The agent also concludes that today is Sunday. Although the neighbors did not initially help the agent learn if the store was open, the agent was able to learn about the reliability of his neighbors.
1.4 Statement of the Problem

In this research we present programming methods that allow intelligent systems the ability to detect outliers in input information. As the system learns about the soundness of this information, it also learns about the accuracy of the sources.
CHAPTER II

BACKGROUND

2.1 Description of a Knowledge Base

A knowledge base describes a collection of objects in a domain and the set of relationships between those objects. The knowledge base serves as the basic component of an agent by encoding the agent’s knowledge about the environment. There are several different ways to represent a knowledge base. One method, which we will use throughout the rest of this work, is to encode the knowledge in the form of a logic program. CR-Prolog is the knowledge representation language that we will use to design knowledge bases.

2.2 Description of CR-Prolog

In this work we present the representation of knowledge bases as CR-Prolog programs. CR-Prolog (Balduccini and Gelfond 2003; Balduccini and Mellarkod 2004a; Balduccini and Mellarkod 2004b) is an extension of A-Prolog (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991) that uses consistency restoring rules (cr-rules) with preferences which are capable of gracefully performing the reasoning needed for certain types of conflict resolution. The cr-rules can be used to encode different types of common-sense knowledge which have no natural formalization in A-Prolog.
A knowledge base should accurately model the environment but become inconsistent in the event of an unobserved occurrence without the use of cr-rules. Applying cr-rules allows the inconsistent knowledge base to restore consistency by assuming that events happened unobserved. If the knowledge base does not accurately model the environment, we would like the program to remain inconsistent even after all cr-rules have been applied.

2.3 Designing Knowledge Bases in CR-Prolog

CR-Prolog offers a variety of functionality for the purpose of encoding knowledge bases (Balduccini and Mellarkod 2004b). In the following sections we will examine some of the features that are available in CR-Prolog but first we will give an informal introduction to the language.

2.3.1 Informal Introduction to CR-Prolog

CR-Prolog programs consist of regular rules and cr-rules. A regular rule is a statement:

\[ r : h_1 \text{ or } h_2 \text{ or } \ldots \text{ or } h_k : - \ l_1, \ldots, l_m, \not l_{m+1}, \ldots, \not l_n \]  

(2.1)

where \( r \) is the name of the rule, \( h_i \)'s and \( l_i \)'s are literals, \( h_1 \text{ or } \ldots \text{ or } h_k \) is the head, and \( l_1, \ldots, l_m, \not l_{m+1}, \ldots, \not l_n \) is the body. The intuitive reading of (2.1), in terms of the beliefs that a rational agent complying with the rule should have, is: “if
the agent believes $l_1, \ldots, l_m$ and does not believe $l_{m+1}, \ldots, l_n$, then it must believe one element of the head of the rule.” It is important to note that not believing $l$ does not imply believing $\neg l$.

In order to increase the readability of the programs, we allow regular rules with *choice atoms* in the head (Niemela, Simons, and Soininen 2002):

$$r : L\{p(X) : q(X)\}U : - l_1, \ldots, l_m,$$
$$\quad \text{not } l_{m+1}, \ldots, \text{not } l_n$$

Intuitively, the head of this rule defines subset $p \subseteq q$, such that $L \leq |p| \leq U$.

Although this form can be translated in rules of type (2.1), it allows for more concise programs, in particular when writing planners.

A *cr-rule* is a statement of the form:

$$r : h_1 \text{ or } h_2 \text{ or } \ldots \text{ or } h_k + - l_1, \ldots, l_m,$$
$$\quad \text{not } l_{m+1}, \ldots, \text{not } l_n$$

(2.3)

The cr-rule intuitively says that, if the agent believes $l_1, \ldots, l_m$ and does not believe $l_{m+1}, \ldots, l_n$, then it “may possibly” believe one element of the head. This possibility is used only if there is no way to obtain a consistent set of beliefs using regular rules only.
2.3.2 Minimal Set Inclusion

CR-Prolog uses minimal set inclusion on the candidate answer sets of the program to compute answer sets. To illustrate how minimal set inclusion works, we will look at the following example program:

\textit{\%Regular Rules}
\begin{align*}
c & : - a. & c & : - b. \\
& : - \text{not} \ c.
\end{align*}

\textit{\%Cr-rules}
\begin{align*}
r1: & a + - . & r2: & b + - .
\end{align*}

When this program is run in a CR-Prolog inference engine such as CRModels (Kolvekal 2004), two models are returned, \{a,c\} and \{b,c\}. The set \{a,b,c\} is a candidate answer set of the program but it is not returned by the inference engine because \{a,c\} and \{b,c\} are smaller subsets of \{a,b,c\}. If the following two constraints were added to the program
\begin{align*}
& : - \text{not} \ a. & & : - \text{not} \ b.
\end{align*}
the candidate answer set \{a,b,c\} would be returned by the program since it would be the only minimal answer set.

2.3.3 Preferences in CR-Prolog

CR-Prolog allows the user the ability to specify preferences (Balduccini and Gelfond 2003) between consistency, restoring rules within the program. Preference
rules are written in the form $\text{prefer}(\text{rule1}, \text{rule2})$ which is read as “do not apply rule2 unless there are no solutions obtained by applying rule1.” Let us add the following preference rule to the first five rules of the previous program and observe the results.

\[
\% \text{Prefer rule } r1 \text{ over rule } r2
\]

\[
\text{prefer}(r1, r2).
\]

When the program is run in a CR-Prolog inference engine, only one answer set \{a, c\} is returned because the candidate answer set \{a, c\} is preferred over the candidate answer set \{b, c\} since a is preferred over b. However if the constraint

\[
:- \ a.
\]

were to be added to the program, \{b, c\} would be the only answer set returned under the semantics of CR-Prolog.


## 2.3.4 Minimum Cardinality

A CR-Prolog inference engine has the ability to return only those candidate answer sets that apply the minimum amount of cr-rules to an inconsistent program. Consider the following example program:

\[
\% \text{Regular rules}
\]

\[
d :~ a. \quad d :~ b, c.
\]

\[
:- \text{not} \ d.
\]

\[
\% \text{Cr-rules}
\]

\[
r1: a +-. \quad r2: b +-. \quad r3: c +-.\]
Running this program in a CR-Prolog inference engine using minimal set inclusion returns two answer sets, \{a,d\} and \{b,c,d\}. Notice that the first answer set requires rule \( r1 \) to fire, but the second answer set requires rules \( r2 \) and \( r3 \) to fire. When minimum cardinality is applied and specified in the command line, only one answer set is returned, \{a,d\}, since \{b,c,d\} is obtained by applying two cr-rules.

Minimum cardinality is applied as a preference that says “prefer candidate answer sets constructed with a minimal number of cr-rules over all other candidate answer sets.” This preference is applied to the candidate answer sets after the preferences of the program have been applied.

2.3.5 Preferring Different Types of Exceptions

The previous examples demonstrate how one cr-rule can be preferred over another, as well as show how answer sets may be limited to those candidate answer sets which apply the minimal amount of cr-rules. Expanding on this knowledge one might ask, “How do we prefer one type of default exceptions over other types?” This type of reasoning becomes important when we begin to monitor agent reliability. Let’s assume that within a domain there exist two classes of exceptions which may be used to restore consistency to the program. These classes consist of naturally occurring exceptions and rare exceptions. Naturally occurring exceptions happen quite often within the environment, but rare exceptions are out of the ordinary. When forced to believe an unobserved event has occurred, the agent should believe a naturally
occurring exception or a combination of naturally occurring exceptions may have
happened before believing any rare exceptions have taken place.

To illustrate this concept we will give an informal example. Suppose a reliable
student arrives at school late on a cold, icy day and reports to his professor that he is
tardy because an accident caused a traffic jam. Given the current weather conditions
the professor likely would not question the students tardiness. But if a student came
to school late on a warm, sunny day when the roads were empty and reported that he
waited in traffic for some time, the professor might find his story to be questionable.
In both instances the student makes the same claim; however, the circumstances of
the environment make his story more or less believable. The default in this story
is that “normally reliable students arrive at school on time.” Naturally occurring
exceptions to this default include bad weather, traffic or car trouble. Rare exceptions
for reliable students would consist of oversleeping, laziness or irresponsibility. When
the default is broken by a student’s tardiness, the professor must decide whether or
not to believe the student’s excuse. The professor also limits the number of times
he will accept naturally occurring exceptions. If a student showed up late everyday
saying traffic was the problem, the professor would label the student as irresponsible.

As long as a student is on time the majority of class days and the student limits
the number of natural occurring exceptions of tardiness to an acceptable amount, the
professor will believe he is a responsible student. The rational professor strives to
believe in the student’s favor until he is forced to believe otherwise. This example
illustrates an intuitive scenario when naturally occurring exceptions are preferred over rare exceptions within the environment. Now that we have an intuitive understanding of naturally occurring and rare exceptions, we will look at a programming method that captures this reasoning.

2.3.5.1 Encoding Rare and Naturally Occurring Exceptions

Suppose bad weather conditions are exceptions to defaults of the knowledge base when they occur within an environment. These conditions consist of rain and snow which happen frequently, as well as tornadoes and earthquakes which are rare. It tends to rain more often than it snows. In this environment rain and snow are naturally occurring exceptions while tornadoes and earthquakes serve as rare exceptions. The following cr-rules capture this information.

% Naturally Occurring Exceptions

%Rain and snow may possibly occur

r1nat: occur(rain) +-. 

r2nat: occur(snow) +-. 

%Rain occurs more often than snow

prefer(r1nat,r2nat).

% Rare Exceptions

%K is used to count the number of rare exceptions

const k=2.
\[\text{count}(0..k).\]

\[\#\text{domain count}(K).\]

\% Tornadoes and earthquakes are rare exceptions
\n\text{rare_exc(tornado;earthquake).}

\% K amount of rare exceptions may possibly occur
\n\text{r1rare}(K): \text{num\_rare\_exc}(K) \text{ + - .}

\% Candidate answer sets with cardinality of rare exceptions are preferred over others
\n\text{prefer(r1rare}(K),\text{r1rare}(K+1)) \text{ :- } K < k.

\% Generate K rare exceptions
\n\text{K\{occur}(X)\text{:rare\_exc}(X)\text{\}}K :- \text{num\_rare\_exc}(K).

Rules \textit{r1nat} and \textit{r2nat} state that “rain and snow may possibly occur.” The following preference rule states that “rain occurs more often than snow.” These two rules model the naturally occurring exceptions of the environment. The next three facts state that integer \( K \) ranges from 0 to \( k \). In this scenario, \( k \) has been set equal to 2, but \( k \) may also be used as a constraint to limit the number of rare exceptions that can occur at one time. The next two facts state that “tornadoes and earthquakes are rare exceptions.” The cr-rule \textit{r1rare}(K) states that “\( K \) rare exceptions may possibly occur.” The following preference rule states that “answer sets containing fewer rare exceptions are preferred over other answer sets.” The choice rule generates \( K \) amount of rare exceptions as determined by \text{num\_rare\_exc}(K).
It is important to note that this encoding will produce the minimal answer sets with respect to cardinality on the literal, \textit{occur}(EXC), where \textit{EXC} is a rare exception of the form \textit{rare_exc}(EXC). Rewriting the preference rule as
\begin{verbatim}
prefer(r1rare(0),r1rare(K)) :- K != 0
\end{verbatim}
will return all possible occurrences of rare exceptions if they are needed to restore consistency to the regular rules of the program. This is done by preferring 0 rare exceptions over all others, which means that if a rare exception occurs then any amount of rare exceptions may possibly occur. In this instance candidate answer sets with one rare exception will not be preferred over candidate answer sets with two rare exceptions.

2.4 Example of a Knowledge Base in CR-Prolog

Recall the Store Example from section 1.3.1 of the Introduction. The following CR-Prolog program models the store environment.

2.4.1 Modeling the Store Environment

\begin{verbatim}
%Time steps of the program
const n = 1.
step(0..n).
#domain step(T).

% Naturally occurring exceptions in the environment
exception(sunday; holiday; weather).
\end{verbatim}
#domain exception(E).

% Normally the store is open

\[ h(\text{store\_open},T) :- \text{not} \ ab(T). \]

\[ ab(T) :- h(E,T). \]

% CWA on store being open

\[ -h(\text{store\_open},T) :- ab(T). \]

% Consistency Restoring Rule

% Naturally occurring exceptions may possibly occur

\[ r1(E,T): h(E,T) +\ T < n. \]

The rule \( r1(E,T) \) says, “If the regular rules of the program are inconsistent, then apply an exception at step \( T \) in an attempt to restore consistency to the program.” If the agent knows the store is closed but is not sure why, he will believe that one of the exceptions in the domain has occurred.

Let’s suppose that we learn the following information: there are approximately fifty Sundays, thirty government holidays and seven bad weather days during which businesses close during any given year. How can this new information be captured into our knowledge base? One method is to replace the cr-rule, \( r1(E,T) \), with the following rules:

\[ r1(T): h(\text{sunday},T) +. \]

\[ r2(T): h(\text{holiday},T) +. \]

\[ r3(T): h(\text{weather},T) +. \]
prefer(r1(T),r2(T)).

prefer(r2(T),r3(T)).

These rules state that Sundays, holidays and bad weather conditions may possibly occur, however, Sundays are more likely to occur than holidays, and holidays are more likely to occur than bad weather. It is important to note that since Sundays are preferred over holidays, and holidays are preferred over bad weather conditions, then Sundays are also indirectly preferred over bad weather.

2.4.2 Store Example Scenarios

We will now look at a few scenarios and run the program to test the store knowledge base. Each scenario is run independently of the other scenarios.

2.4.2.1 Scenario 1

If no additional information is added to the knowledge base, the agent will believe the store is open at time step 1 since “normally the store is open.”

2.4.2.2 Scenario 2

Suppose the agent learns that today is a holiday. The fact \( h(holiday,1) \) is added to the knowledge base to represent this information. When the program is run the fluent \( \neg h(store\_open,1) \) is returned, which says the store is closed at time step 1.
2.4.2.3 Scenario 3

The agent learns that the store is not open at time step 1. The fact 
\(-h(store\_open,1)\) is added to the knowledge base. This causes the regular rules of the 
program to become inconsistent and the cr-rule \(r1(T)\) is applied to restore consistency. 
The program concludes \(h(sunday,1)\), meaning that today is a Sunday. If the agent 
learned that it was not Sunday, which is represented by adding the fact \(-h(sunday,1)\), 
the program would conclude \(h(holiday,1)\), meaning that it is a holiday.

2.5 Related Work

In (Angiulli, Greco, and Palopoli 2004), the authors formally state the concept 
of outliers in the context of logic programming. Their work focuses on the task of 
singling out anomalous individuals from a given population to both detect rare events 
in a time-series analysis settings and to identify objects whose behavior is deviant with 
respect to a codified standard set of “social” rules.

A major contribution of their work lies in the exploitation of a minimality cri-
teria in the outlier detection, as well as, determining the computational complexity 
of detecting outliers. In the final section of their work, they propose a rewriting algo-
thesis that transforms any outlier problem into an equivalent answer set computation 
problem.
2.5.1 Method of Detecting Outliers

The definition of outlier detection as introduced by Angiulli, Greco and Palopoli takes as input an arbitrary logic program $P^{rls}$ and a set of observations $P^{obs}$. They define $P = <P^{rls}, P^{obs}>$, relating the general knowledge encoded in $P^{rls}$ with the evidence about the world contained in $P^{obs}$ as the input for the outlier detection process. Given $P$, they are interested in identifying a set $O \subseteq P^{obs}$ that are anomalous according to the general theory $P^{rls}$ and the other facts in $P^{obs} \setminus O$. The idea underlying the identification of $O$ is to discover a witness set $W \subseteq P^{obs}$ that is a set of facts which would be explained in the theory if and only if all the facts in $O$ were not observed. The activity of identifying the witness sets constitutes the main source of computational complexity in outlier detection problems and is also the distinguishing characteristic. If a witness set cannot be found then according to their definition, no outliers exist.

2.5.2 Algorithm

Angiulli, Greco and Palopoli also exhibit a sound and complete algorithm that transforms any rule-observation pair $P$ into a suitable logic program $L(P)$, such that its stable models are in a one-to-one correspondence with outliers in $P$. The rewriting algorithm, $OutlierDetectionToASP$ as it is named, takes as input a pair $P = <P^{rls}, P^{obs}>$ and outputs a logic program $L(P)$ that is capable of detecting outliers.
2.5.3 Evaluation

The work of Angiulli, Greco and Palopoli produced in the research paper are the first attempts to detect outliers in input information. The complexity computations of their work are helpful in determining the complexity of detecting outliers in a wide range of programs. While the definition given for an outlier detects most outliers in many given scenarios, we have discovered that the definition does not detect every intuitive outlier in all scenarios. Later in this work we will compare and contrast their definition of an outlier with the definition we introduce in the next chapter. We did not attempt to evaluate their algorithm due to these differing definitions. The complexity of their algorithm hinges on the computation of a witness set, which we do not use in our definition of an outlier.
Now that we have an intuitive understanding of an outlier, we will present a more precise definition.

3.1 Framework

Observations can be made by various sources within an environment which are modeled by an agent’s knowledge base. These observations are stored as a set of literals, $Obs$, and are combined with the agent’s knowledge base to aid the agent in the decision-making process. The agent uses these observations in the reasoning process, so long as the union of the knowledge base and the observation set remain consistent. The following definition can be used to determine outlier sets of a given CR-Prolog program and a related set of observations.

3.2 Definition of an Outlier Set

**Definition (Outlier Set).**

Given a consistent logic program $P$ of CR-Prolog and a set of observations $Obs$, a set $O \subseteq Obs$ is called an outlier set of $P$ with respect to $Obs$ if:

1) $P \cup Obs$ is inconsistent and

2) $P \cup Obs \setminus O$ is consistent.
The elements of an outlier set are simply called outliers. For any given program $P$ of CR-Prolog, and an observation set $Obs$, there may exist multiple outlier sets of $P$ with respect to $Obs$. In most instances we will be interested in determining minimal outlier sets, meaning the outliers containing the least amount of observations. If the union of program $P$, and observations $Obs$, is consistent, then no outliers exist. If program $P$ is inconsistent, then we can conclude that the program does not accurately model the environment.

3.3 Outlier Examples

To gain an understanding of what is an outlier according to the definition we will start by showing a very simple program. Then we will give a few more complex examples.

3.3.1 Simple Example

Consider the program $P_S$ which consists of the following rules:

\[
\begin{align*}
a. & \quad \neg b. \\
c. &
\end{align*}
\]

The following observations have been made in the environment and are contained in the observation set such that $Obs_S = \{a,b,q\}$. Intuitively we can see that the observation, $b$, is an outlier, but let’s test it according to the definition.
Program $P_S$ is the above program.

Let $\text{Obs}_S = \{a,b,q\}$ and $O_S = \{b\}$.

1. $P_S \cup \text{Obs}_S$ is inconsistent

   \[
   \{a, \neg b, c\} \cup \{a,b,q\} \text{ has no answer sets}
   \]

2. $P_S \cup \text{Obs}_S \setminus O_S$ is consistent

   \[
   \{a, \neg b, c\} \cup \{a,q\} \text{ is consistent and entails } \{a,\neg b, c, q\}
   \]

Conclusion: the set $O_S = \{b\}$ is an outlier set according to the definition.

It is important to note that although $\{b\}$ is an outlier set of program $P_S$ with respect to $\text{Obs}_S$, every subset of $\{a,b,q\}$ containing $b$ is also an outlier set. The outlier set, $\{b\}$, is the minimal outlier set for this example.

3.3.2 Light Bulb Example

A room contains a light switch and a light bulb. If the switch is turned on, then the light bulb will be lit unless the bulb is broken. The light switch is either on or off at all times, represented by the notation $\text{switch}_\text{on}$ and $\neg\text{switch}_\text{on}$; the bulb is either on or off at all times, represented by the notation $\text{bulb}_\text{lit}$ and $\neg\text{bulb}_\text{lit}$. A bulb may be broken if the bulb has burnt out ($\text{burnt}_\text{bulb}$) or if there is a $\text{power}_\text{outage}$. Bulb burnouts happen more frequently than power outages.
The following CR-Prolog program, $P_L$, models the Light Bulb environment:

{%The light switch is either on or off at all times

\begin{verbatim}
switch_on :- not \neg switch_on.
\neg switch_on :- not switch_on.
\end{verbatim}

%Normally when the light switch is on the bulb is lit

\begin{verbatim}
bulb_lit :- switch_on,
not broken.
\end{verbatim}

%The bulb is broken if it is burnt out or there is a power outage

\begin{verbatim}
broken :- burnt_bulb.
broken :- power_outage.
\end{verbatim}

%It is impossible for the bulb to be lit when the switch is off

\begin{verbatim}
:- bulb_lit, \neg switch_on.
\end{verbatim}

%It is not possible for the switch to be on and the bulb to be %off but not be broken

\begin{verbatim}
:- switch_on, \neg bulb_lit, not broken.
\end{verbatim}

%Cr-rules to restore consistency

%Burnt bulbs and power outages may possibly occur

\begin{verbatim}
r1: burnt_bulb +-. 
r2: power_outage +-. 
\end{verbatim}

%Bulbs burn out more often then power outages occur

\begin{verbatim}
prefer(r1,r2).
\end{verbatim}

25
3.3.2.1 Scenario 1

It is observed in the domain that the light switch is on. This observation is added to the knowledge base, and the program concludes switch_on and bulb_lit. No cr-rules were used in the construction of this answer set and no outliers exist.

3.3.2.2 Scenario 2

It is observed in the domain that the switch is on, and the light bulb is not lit. These observations are added to the observation set. Because the regular rules of the program are inconsistent, the cr-rule r1 fires and causes the program P_L to conclude that the bulb is burnt out since burnt bulbs are preferred over power outages. Because the knowledge base remains consistent, there are no outliers in this scenario and the answer set, \{switch_on, ¬bulb_lit, broken, burnt_bulb, ...\}, is returned by the program. If it were observed that the bulb was not burnt, then the program would conclude that a power outage had occurred.

3.3.2.3 Scenario 3

It is reported to the agent that the light switch is off, but the bulb is on. These observations are recorded in the observation set. When the observations of Obs_L are combined with P_L, the program becomes inconsistent because the union contains either bulb_lit and ¬bulb_lit or switch_on and ¬switch_on. Removing the observation ¬switch_on from Obs_L, because it is an outlier, and combining the observation set
with $P_L$, causes the program to entail $\text{switch}_\text{on}$ and $\text{bulb}_\text{lit}$. Consistency may also be restored to the program by instead labeling $\text{bulb}_\text{lit}$ as an outlier and removing it from the observation set which causes the program to entail $\neg \text{switch}_\text{on}$.

It is important to note that according to the definition, the set \{¬$\text{switch}_\text{on}$, $\text{bulb}_\text{lit}$\} is also an outlier set because removing both observations from $\text{Obs}_L$ will restore consistency to the program. Although the set containing both observations is an outlier set, the agent will prefer in most instances to believe minimal outlier sets.

3.3.3 Network Example

The following example is taken from (Angiulli, Greco, and Palopoli 2004), as described in section 2.5. Let us consider the network of computers (Figure 3.1), which is monitored by an intelligent agent. All computers in the diagram are either on or off at all times. The arrows in the diagram represent the wiring of the computers. A computer is connected if it is wired to a machine that is on. Computer $s$ is always on which is designated by the notation $\text{on}(s)$. A computer different from $s$ can only
be on if it is connected to another computer that is on. Even though a computer is
connected, it may still be off. Since s is on and wired to a, denoted by the notation
\textit{wired}(s,a), computer a is connected which is denoted by \textit{connected}(a). Computer a
will normally be on since it is connected, but it may still be off.

The agent’s knowledge consists of an A-Prolog program, $P_N$, and a set of
observations, $\textit{Obs}_N$. Program $P_N$ is shown below.

3.3.3.1 Network Program

\texttt{\%Objects}

\begin{verbatim}
r1 : computer(s). computer(a). ... computer(t).
\end{verbatim}

\texttt{\%Wiring of the Network.}

\begin{verbatim}
r2 : wired(s,a). ... wired(g,t).
\end{verbatim}

\texttt{\%Rules}

\texttt{\%Normally a network computer is on.}

\begin{verbatim}
r3 : on(X) :- computer(X),
    not \neg on(X).
\end{verbatim}

\texttt{\%Normally a network computer is off if it is not connected.}

\begin{verbatim}
r4 : \neg on(X) :- computer(X),
    not connected(X).
\end{verbatim}
%A computer is connected if wired to a machine that is on.

\[ r5 : \text{connected}(Y) :- \text{wired}(X, Y), \]
\[ \text{on}(X). \]

%Computer s is always on and connected.

\[ r6 : \text{on}(s). \]
\[ r7 : \text{connected}(s). \]

3.3.3.2 Scenario

Assume that the following facts were observed: computers s, h, b, d, e, f, g, t are on and computers a and c are off. In Figure 3.1, the computers marked in bold were observed to be off. This information is recorded in the observation set such that \( \text{Obs}_N = \{\text{on}(s), \neg\text{on}(a), \text{on}(h), \text{on}(b), \neg\text{on}(c), \text{on}(d), \text{on}(e), \text{on}(f), \text{on}(g), \text{on}(t)\} \).

Notice that if the computers d, e, f, g and t had not been observed to be on, then the agent would have concluded exactly the opposite by exploiting his knowledge of the world (program \( P_N \)), since the failure of c suffices for breaking the connectivity between computer s and the other machines. In this scenario the literal \( \neg\text{on}(c) \) is intuitively considered to be an outlier.

Let us show that \( \neg\text{on}(c) \) is an outlier with respect to \( P_N \) according to the definition. When the observations of \( \text{Obs}_N \) are combined with program \( P_N \), the program becomes inconsistent. When the observation, \( \neg\text{on}(c) \), is removed and \( \text{Obs}_N \) is combined with \( P_N \), the program remains consistent. According to the definition, the set, \( \{\neg\text{on}(c)\} \), is an outlier, and it is also happens to be the minimal outlier set.
in this scenario. Every subset of \(\{\neg \text{on}(a), \text{on}(b), \text{on}(d), \text{on}(e), \text{on}(f), \text{on}(g), \text{on}(h), \text{on}(t), \neg \text{on}(c)\}\) containing \(\neg \text{on}(c)\) is also an outlier set.

Often the minimal outlier set is the model of the environment believed by the agent since it is the most likely to occur. However, the minimal outlier set may not be the actual outlier set of a scenario. When the program is inconsistent and outliers are generated, the agent may be forced to choose one of the outlier sets. At a later time step, the agent may receive additional information that no longer validates the original minimal outlier set, thus causing him to choose a different outlier set.
CHAPTER IV
MONITORING AGENT RELIABILITY

Now that we have a clear and precise understanding of an outlier, we can address the question of detecting problematic sources submitting information into an intelligent system. Let’s look at the following example.

4.1 Military Example

Consider a military analyst whose job is to monitor the reliability of secret agents by reviewing the information they report. The analyst’s country is at war with an opposing country, and the analyst’s country launches long range missile attacks against military targets. Normally the attacks succeed, but they may occasionally fail. The analyst has a precise understanding of which targets have been attacked, as well as, a general understanding of the typical frequency of failed attacks. However, the analyst relies on the reports of the secret agents to determine the outcome of the attacks. Secret agent reports are normally true, but occasionally may be incorrect. Failed attacks occur quite frequently, but incorrect secret agent reports are rare.

Our goal is to create an intelligent system that can be used as a tool by the analyst to detect false reports or outliers in this environment and to track the reliability of the intelligence gathering secret agents. To create a system that accurately models this environment, two knowledge bases are needed. The first knowledge base,
which we will call the *Battlefield Knowledge Base*, models the environment of attacks
and destroyed targets. The second knowledge base models the domain of gathering
outside information. We will call this second knowledge base the *Secret Agent Knowl-
dge Base* because it contains rules concerning secret agents and their reports. When
the Battlefield Knowledge Base and the Secret Agent Knowledge Base are combined,
they build the framework needed to detect outliers in the form of problematic reports
and monitor the reliability of the intelligence gathering agents.

### 4.1.1 Battlefield Knowledge Base

The Battlefield Knowledge Base is a CR-Prolog program that dynamically
models the battlefield environment of the Military Example. The meaning of the
relations used in the program are as follows:

- *T* - Time steps of the program
- *target(TAR)* - TAR is a target
- *destroyed(TAR)* - Target TAR has been destroyed
- *o(attack(TAR),T)* - Target TAR was attacked at step T
- *failed(attack(TAR),T)* - The attack against target TAR at step T failed

The following CR-Prolog program models the Battlefield domain:

```prolog
% Dynamic Causal Law
h(destroyed(TAR), T+1) :- o(attack(TAR), T),
    -failed(attack(TAR), T).
```

32
% Consistency Restoring Rules
\[ r(\text{TAR}, T) : \text{failed}(\text{attack}(	ext{TAR}), T) \leftarrow \text{o}(\text{attack}(	ext{TAR}), T). \]
\[ \leftarrow (m+1)\{\text{failed}(\text{attack}(	ext{Tar}), S) : \text{target}(	ext{Tar}) : \text{step}(S)\}. \]

% Inertia Axiom
\[ h(F, T+1) \leftarrow T < n, \]
\[ h(F, T), \]
\[ \neg h(F, T+1). \]
\[ -h(F, T+1) \leftarrow T < n, \]
\[ -h(F, T), \]
\[ \neg h(F, T+1). \]

% Non-Inertial Fluents
\[ -\text{failed}(\text{attack}(	ext{TAR}), T) \leftarrow \neg \text{failed}(\text{attack}(	ext{TAR}), T). \]

The action description consists of the rules in the first two sections modeling the knowledge base of the Battlefield domain. Statement \( h(f, t) \) states that “fluent \( f \) holds at time \( t \)” and \( o(a, t) \) represents that “action \( a \) occurred at time \( t \).”

The first section of the knowledge base contains dynamic causal laws (McCain and Turner 1995; Gelfond and Lifschitz 1993; Gelfond and Lifschitz 1998) of the environment. The Dynamic Causal Laws state that “a successful attack destroys the target.” Exceptions to a destroyed target are failed attacks.

The second section contains cr-rules that model the occurrence of acceptable natural failures within the environment. Acceptable natural failures consist of equip-
ment failures, human errors and other events that may lead to failed attacks during the firing process. The analyst is capable of specifying the number of acceptable natural failures $m$, within the environment. The rule $r(TAR, T)$ says, “It may be possible for an attack to fail if the target has been attacked, but such events are rare.” The next rule is a constraint that says, “It is impossible for more than $m$ failures to occur.”

The rules of the third section formalize the Inertia Axiom (Hayes and McCarthy 1969) which states that “things tend to stay as they are.” The final section contains the rules of the non-inertial fluents within the domain. The non-inertial fluent states, “An attack did not fail if there is no reason to believe the attack did fail”.

The history of actions that are known to have occurred in the domain are also added to the Battlefield Knowledge Base. In this example, the history of known actions includes a list of targets that have been attacked. Other relevant and verified facts may also be added to the Battlefield Knowledge Base.

Now that we have modeled the Battlefield domain, we will introduce a CR-Prolog program that models the domain of the intelligence gathering secret agents called the Secret Agent Knowledge Base. The Secret Agent Knowledge Base represents the secret agent environment and includes the reports submitted by the intelligence gathering secret agents. The Secret Agent Knowledge Base is intended to be combined with the Battlefield Knowledge Base to enhance the system’s reasoning
ability in the environment. Shown below is a CR-Prolog program that models the domain of the intelligence gathering secret agents.

### 4.1.2 Secret Agent Knowledge Base

The meaning of the relations used in the Secret Agent Knowledge Base are as follows:

- `agent(A)` - A is an intelligence gathering secret agent
- `id(R)` - R is the unique identifier of a report submitted by an intelligence gathering secret agent
- `report(R,T), author(R,A), content(R,F,Boolean)` - At time step T, report R was submitted with content F (if `Boolean = “t”`) or ¬F (if `Boolean = “f”`) by agent A

The following program models the domain of information gathering secret agents.

```prolog
% Secret Agent Default

h(F,T) :- report(R1,T),
         content(R1,F,t),
         not problematic(R1).

-h(F,T) :- report(R1,T),
          content(R1,F,f),
          not problematic(R1).
```

All reports submitted by the agents are formatted into the form `report(R,T), author(R,A)` and `content(R,F,Boolean)` as described and entered into the Secret Agent
Knowledge Base to be evaluated by the system.

The two rules of the Secret Agent Knowledge Base state: “normally agents tell the truth.” The system believes the agents’ reports to be true if there is no reason to believe that their reports are problematic.

The Secret Agent Knowledge Base may also contain rules about observations and their sources. Suppose, for instance, that a certain secret agent is always correct, or perhaps when two agent’s reports are contradictory, the one secret agent is preferred over the other. This type of knowledge may be captured by adding additional rules to the Secret Agent Knowledge Base.

4.2 Determining Problematic Reports and Problematic Agents

Given the Battlefield Knowledge Base and the Secret Agent Knowledge Base, we would like to determine outliers in the form of problematic reports that are submitted by agents into the system. A problematic report is described as an observation that causes the program to become inconsistent when added to the Secret Agent Knowledge Base and combined with the Battlefield Knowledge Base. Consistency is restored to the program when the report is labeled as problematic. If a secret agent submits a problematic report, then the agent is considered to be problematic with respect to that problematic report. A precise definition of problematic reports and problematic agents are as follows.
4.2.1 Definition of Problematic Reports

Definition(Problematic Reports).

Given the Battlefield Knowledge Base $B$ and the Secret Agent Knowledge Base $SA$, let $R \subseteq SA$ consist of the set of secret agent reports. A set $PR \subseteq R$, is problematic if and only if $B \cup SA$ is inconsistent and $B \cup (SA \setminus \{PR\})$ is consistent.

4.2.2 Definition of a Problematic Agent

Definition(Problematic Agent).

Given the Battlefield Knowledge Base $B$, the Secret Agent Knowledge Base $SA$, and the set of problematic reports $PR$, a secret agent $a \in SA$ is problematic if and only if $a$ has submitted a report $r$ such that $r \in PR$.

4.3 Problematic Report Solver

The definitions for problematic reports and problematic agents can be modeled into the system by including a CR-Prolog program which we will call the Problematic Report Solver. The Problematic Report Solver is used to detect problematic reports and problematic agents within the Secret Agent Knowledge Base when combined with the Battlefield Knowledge Base.
Listed below are the rules contained in the Problematic Report Solver.

\[
\text{% Problematic Reports}
\]
\[
\text{const } k=3.
\]
\[
\text{num_br(0..k)}.
\]
\[
\text{#domain num_br(K)}.
\]
\[
rr(K) : \text{bad_report(K)} +-. \]
\[
\text{prefer(rr(K),rr(K+1)) :- K < k.}
\]
\[
K\{\text{problematic(Id) : id(Id)}\}K :- \text{bad_report(K)}.
\]
\[
\text{problematic_agent(A) :- problematic(R),}
\]
\[
\text{author(R,A).}
\]

The analyst is capable of specifying the number of bad reports the system may generate by setting the constant \( k \) equal to an acceptable amount. If the system is making unreasonable assumptions, the analyst would like the program to remain inconsistent. The cr-rule \( rr(K) \) states that “\( K \) amount of bad reports may possibly occur.” The preference rule states that “\( K \) bad reports are preferred over \( K+1 \) bad reports.” This preference rule causes the program to return minimal outlier sets based on the cardinality of problematic reports. Rewriting the preference rule as \( \text{prefer(rr(0),rr(K)) :- K!=0} \) would cause the program to return all outlier sets of the program. The next rule is a choice rule that generates \( K \) problematic reports. The final rule states: “An agent is problematic if he submits a problematic report.”
4.4 Failed Attacks vs. Bad Reports

When the system becomes inconsistent two cr-rules may be applied to restore consistency. The rule $r(TAR, T)$ from the Battlefield Knowledge Base generates failed attacks in an effort to restore consistency. The rule $rr(K)$ in the Problematic Report Solver attempts to restore consistency to the program by labeling reports as problematic. Given that failed attacks occur frequently and false reports are rare, the analyst would like the system to believe an acceptable number of failed attacks have occurred before labeling an agent’s report as problematic. Because models containing zero problematic reports are preferred over other models by the preference rule in the Problematic Report Solver, the program will generate naturally occurring exceptions before generating problematic reports. This programming method allows the system to return the maximum number of acceptable failed attacks before labeling any agent’s report as problematic.

4.5 Testing the System

The following rules are added to the respective programs to test the scenarios.

4.5.1 Objects of the Battlefield Knowledge Base

```plaintext
% Time steps of the program
const n = 1.
step(0..n).
```
4.5.2 Objects of the Secret Agent Knowledge Base

% Intelligence Gathering Agents

agent(a1;a2;a3).

#domain agent(A).

4.5.3 Objects of the Problematic Report Solver

% Unique report identifiers.

id(r1;r2;r3;r4;r5).

#domain id(R,R1).

br(0..k). #domain br(K).
When the Battlefield Knowledge Base, $B$, is combined with the Secret Agent Knowledge Base, $SA$, and the Problematic Report Solver, $PRS$, the system is capable of detecting outliers in the form of problematic reports and problematic agents within the environment. To illustrate the reliability of the system we will give some scenarios and test the system’s response.

4.5.4 Military Scenarios

In the following scenarios, when we refer to running the system or running the program, we are referring to running $B \cup SA \cup PRS$ under the semantics of CR-Prolog. Each scenario is run independently of the other scenarios. Minimal outlier sets are returned in the following examples with respect to the cardinality of problematic reports.

4.5.4.1 Scenario 1

At time step 0 an attack was launched against target $t1$. This information is captured in the history section of the Battlefield Knowledge Base by adding the fact $o(attack(t1),0)$. The system returns the literal, $h(destroyed(t1),1)$, which represents that target $t1$ was destroyed at time step 1.
4.5.4.2 Scenario 2

At time step 0 attacks were launched against targets $t1$ and $t2$. This information is captured in the history section of the Battlefield Knowledge Base by adding the facts $o(attack(t1),0)$ and $o(attack(t2),0)$. Agent $a1$ reports that target $t1$ was destroyed at time step 1. The facts $report(r1,1)$, $author(r1,a1)$ and $content(r1,destroyed(t1),t)$ are added to the Secret Agent Knowledge Base to represent agent $a1$’s report.

When the system is run, one answer set is returned which contains the literals $h(destroyed(t1),1)$ and $h(destroyed(t2),1)$ representing that targets $t1$ and $t2$ were destroyed at time step 1.

4.5.4.3 Scenario 3

At time step 0 an attack was launched against target $t1$. This information is represented in the history section of the Battlefield Knowledge Base by adding the rule $o(attack(t1),0)$. Agents $a1$ and $a3$ report that $t1$ was not destroyed, and agent $a2$ states that $t1$ was destroyed. The following rules are added to the Secret Agent Knowledge Base to represent this information: $report(r1,1)$, $author(r1,a1)$, $content(r1,destroyed(t1),f)$, $report(r2,1)$, $author(r2,a2)$, $content(r2,destroyed(t1),t)$, $report(r3,1)$, $author(r3,a3)$ and $content(r3,destroyed(t1),f)$.

Because the secret agents’ reports are in direct contradiction, the system must decide which statements to believe. The system produces one answer set
for this scenario, \{problematic(r2), failed(attack(t1),0), -h(destroyed(t1),1), problematic_agent(a2), ... \}, which says the attack against \textit{t1} failed and agent \textit{a2} is problematic with respect to report \textit{r2}. The system chooses to believe agents \textit{a1} and \textit{a3}'s statements over \textit{a2}'s because believing \textit{a1} and \textit{a3} requires the occurrence of one problematic report and believing \textit{a2} requires the occurrence of two. If a fourth agent submitted a report with the same statement as \textit{a2}, then the program would produce two answer sets, as both models would have a minimum number of problematic reports.

4.5.4.4 Scenario 4

At time step 0 attacks were launched against targets \textit{t1} and \textit{t2}. This information is captured in the history section of the Battlefield Knowledge Base by adding the facts \textit{o(attack(t1),0)} and \textit{o(attack(t2),0)}. Agent \textit{a1} reports that target \textit{t1} was not destroyed. The facts \textit{report(r1,1)}, \textit{author(r1,a1)} and \textit{content(r1,destroyed(t1),f)} are added to the Secret Agent Knowledge Base to represent agent \textit{a1}'s report. The number of acceptable failed attacks \textit{m}, is set to 1 by the analyst.

Running this program returns the answer set, \{failed(attack(t1),0), -h(destroyed(t1),1), h(destroyed(t2),1), ... \}, which states that the attack on target \textit{t1} failed and the target was not destroyed, but target \textit{t2} was destroyed. If the analyst were to change the number of acceptable failed attacks to 0 meaning failed attacks are not possible the answer set, \{problematic(r1), h(destroyed(t1),1), h(destroyed(t2),1), \
... }, would be returned stating that agent $a1$’s report $r1$ is problematic, and target $t1$ and $t2$ have been destroyed.

4.5.4.5 Scenario 5

At time step 0 attacks were launched against targets $t1$, $t2$ and $t3$. This information is captured in the history section of the Battlefield Knowledge Base by adding the facts $o(attack(t1),0)$, $o(attack(t2),0)$ and $o(attack(t3),0)$. Agent $a1$ reports that target $t1$ was not destroyed, agent $a2$ reports that target $t2$ was not destroyed and agent $a3$ reports that target $t3$ was not destroyed. These three statements are translated into the following facts and added to the Secret Agent Knowledge Base:

\begin{align*}
report(r1,1), & \quad author(r1,a1), \quad content(r1,destroyed(t1),f), \quad report(r2,1), \quad author(r2,a2), \\
& \quad content(r2,destroyed(t2),f), \quad report(r3,1), \quad author(r3,a3) \quad \text{and} \quad content(r3,destroyed(t3),f).
\end{align*}

The analyst sets the maximum number of failed attacks to two by setting $m=2$. Three answer sets are returned when the system is run.

The first answer set, \{ $h(destroyed(t1),1)$, $-h(destroyed(t2),1)$, $-h(destroyed(t3),1)$, failed(attack(t2),0), failed(attack(t3),0), problematic(r1), problematic_agent(a1), ...
\}, says that the attack against target $t1$ succeeded but the attacks against targets $t2$ and $t3$ failed. Report $r1$ is believed to be problematic; secret agent $r1$ is believed to be problematic with respect to report $r1$.

The second answer set, \{-$h(destroyed(t1),1)$, $h(destroyed(t2),1)$, $-h(destroyed(t3),1)$, failed(attack(t1),0), failed(attack(t3),0), problematic(r2), prob-

44
lematic_agent(a2), ... }, states that the attack against target $t_2$ succeeded, the attacks against $t_1$ and $t_3$ failed, and report $r_2$ was problematic. Secret agent $a_2$ is listed as problematic for submitting report $r_2$.

The third answer set, $\{-h(\text{destroyed}(t_1),1), -h(\text{destroyed}(t_2),1), h(\text{destroyed}(t_3),1), \text{failed}(\text{attack}(t_1),0), \text{failed}(\text{attack}(t_2),0), \text{problematic}(r_3), \text{lematic_agent}(a_3), ... \}$, states that the attack against target $t_3$ succeeded, the attacks against $t_1$ and $t_2$ failed, and report $r_3$ was problematic. Secret agent $a_3$ is listed as problematic for submitting report $r_3$.

4.5.4.6 Scenario 6

At time step 0 attacks were launched against targets $t_1$, $t_2$, $t_3$, $t_4$ and $t_5$. This information is captured in the history section of the Battlefield Knowledge Base by adding the facts: $o(\text{attack}(t_1),0)$, $o(\text{attack}(t_2),0)$, $o(\text{attack}(t_3),0)$, $o(\text{attack}(t_4),0)$ and $o(\text{attack}(t_5),0)$. Agent $a_1$ reports that targets $t_1$ and $t_2$ were not destroyed. Agent $a_2$ reports that targets $t_3$, $t_4$ and $t_5$ were not destroyed. The following reports are added to the Secret Agent Knowledge Base: $\text{report}(r_1,1)$, $\text{author}(r_1,a_1)$, $\text{content}(r_1,\text{destroyed}(t_1),f)$, $\text{report}(r_2,1)$, $\text{author}(r_2,a_1)$, $\text{content}(r_2,\text{destroyed}(t_2),f)$, $\text{report}(r_3,1)$, $\text{author}(r_3,a_2)$, $\text{content}(r_3,\text{destroyed}(t_3),f)$, $\text{report}(r_4,1)$, $\text{author}(r_4,a_2)$, $\text{content}(r_4,\text{destroyed}(t_4),f)$, $\text{report}(r_5,1)$, $\text{author}(r_5,a_2)$ and $\text{content}(r_5,\text{destroyed}(t_5),f)$.

The analyst sets the maximum number of failed attacks to two by setting $m=2$. 
Ten answer sets are returned when the program is run. For readability we will write $\text{prob}(R)$ instead of $\text{problematic}(R)$ and $\text{failed}(\text{TAR})$ instead of $\text{failed}(\text{attack}(\text{TAR}),0)$. The answer sets returned by the program are as follows:

\[
\begin{align*}
\{ & \text{prob}(r1), \text{failed}(t2), \text{prob}(r3), \text{failed}(t4), \text{prob}(r5), \ldots \} \\
\{ & \text{prob}(r1), \text{prob}(r2), \text{failed}(t3), \text{failed}(t4), \text{prob}(r5), \ldots \} \\
\{ & \text{prob}(r1), \text{prob}(r2), \text{prob}(r3), \text{failed}(t4), \text{failed}(t5), \ldots \} \\
\{ & \text{prob}(r1), \text{prob}(r2), \text{failed}(t3), \text{prob}(r4), \text{failed}(t5), \ldots \} \\
\{ & \text{prob}(r1), \text{failed}(t2), \text{failed}(t3), \text{prob}(r4), \text{prob}(r5), \ldots \} \\
\{ & \text{prob}(r1), \text{failed}(t2), \text{prob}(r3), \text{prob}(r4), \text{failed}(t5), \ldots \} \\
\{ & \text{failed}(t1), \text{prob}(r2), \text{failed}(t3), \text{prob}(r4), \text{prob}(r5), \ldots \} \\
\{ & \text{failed}(t1), \text{prob}(r2), \text{prob}(r3), \text{prob}(r4), \text{failed}(t5), \ldots \} \\
\{ & \text{failed}(t1), \text{failed}(t2), \text{prob}(r3), \text{prob}(r4), \text{prob}(r5), \ldots \} \\
\{ & \text{failed}(t1), \text{prob}(r2), \text{prob}(r3), \text{failed}(t4), \text{prob}(r5), \ldots \} \\
\{ & \text{failed}(t1), \text{prob}(r2), \text{prob}(r3), \text{failed}(t4), \text{prob}(r5), \ldots \} \\
\end{align*}
\]

These answer sets represent every possible state incorporating two failed attacks and three problematic reports. The answer sets that are returned by this program all use the minimal amount of problematic reports. It is possible that more than three reports could be problematic, but the system returns the minimal outlier sets because they are the models which are most likely to occur.
4.5.4.7 Scenario 7

Suppose five targets were attacked, and secret agents reported that all five targets were not destroyed. If the analyst decided that two failed attacks were an acceptable amount \( m=2 \) and two problematic reports were acceptable \( k=2 \), the program would return no answer sets. This is because the maximum number of failed attacks plus the maximum number of problematic reports would not restore consistency to the program. Anytime the number of reports claiming attacks failed is greater than \( m+k \), the program will remain inconsistent. Specifying \( m \) and \( k \) allows the analyst to indicate reasonable and unreasonable explanations of a scenario and causes the system to return no answer sets as long as reasonable explanations cannot be obtained.

4.6 Summary

The Military Example shows how multiple knowledge bases can be used to represent the domains of intelligence gathering agents and the environments in which they function. This example also shows how CR-Prolog programs can be used to detect outliers in the form of problematic reports and to discover the problematic sources of these reports. By discovering these problematic sources, the system is able to monitor the reliability of the outside intelligence gathering agents and maintain soundness in the decision-making process.
CHAPTER V

ALGORITHMS

In this section we present two algorithms, DetectOutlierSets and DetectMinimalOutlierSets, that output outlier sets and minimal outlier sets respectively, given an agent’s knowledge and a related set of observations.

The algorithm DetectOutlierSets is sound and complete when given a consistent CR-Prolog knowledge base $KB$, and a set of observations $OBS$. The algorithm DetectMinimalOutlierSets is sound but may not be complete if the knowledge base contains preference rules. If $KB$ does not use preferences, then answer sets of the algorithm are sound and complete. The more complex the cr-rules of the knowledge base are, the more likely they will conflict with the cr-rules used in the minimal outlier detection. In the event that cr-rules do conflict, answer sets of the program will be lost.

5.1 Action Language Description

Let $\Sigma$ be a signature containing three special classes of symbols, $F$, $A$ and $S$ called fluents, actions and time steps respectively. Two relations are used in the signature, $h(f, t)$ and $o(a, t)$ where $f \in F$, $a \in A$ and $t \in S$. 

48
5.2 Framework

Let $KB$ be a CR-Prolog program over $\Sigma$ that models an environment, and let $OBS$ be a set containing observations of the form $\text{obs}(\text{ObsNumber}, \text{Time})$ and $\text{content}(\text{ObsNumber}, \text{Fluent}, \text{Boolean})$ where $\text{Fluent} \in F$, $\text{ObsNumber}$ is a unique identifier for the observation and $\text{Boolean}$ consists of “t” if the $\text{Fluent}$ is true and “f” if $\text{Fluent}$ is not true (classical negation).

5.2.1 Observation Module

Given the observation set $OBS$, a set of rules are needed to translate the observations into the form $h(f,t)$ so they can be joined with the knowledge base. The Observation Module ($OM$) is the program that performs this task. The rules of the Observation Module state that “normally observations are correct.” The exception to this default is when an observation is an outlier.

Let program $OM$ contain the following rules:

% Normally observations are correct

\[
\begin{align*}
 h(F,T) &::= \text{obs}(ID,T), \\
 &\quad \text{content}(ID,F,t), \\
 &\quad \text{not outlier}(ID). \\
-h(F,T) &::= \text{obs}(ID,T), \\
 &\quad \text{content}(ID,F,f), \\
 &\quad \text{not outlier}(ID).
\end{align*}
\]
5.2.2 Outlier Detection

Given $KB$, $OBS$ and $OM$, a set of rules are needed to instruct the system on how to detect outliers. The program Outlier Detector ($OD$) contains rules that are used to detect outliers in the observation set. The constant $k$ is a positive integer used as a counter that specifies the number of outliers the choice rule may generate. The cr-rule $rOD(K)$ states: “$K$ amount of bad observations may possibly occur.” The choice rule generates $K$ amount of outliers, as long as $K$ is not equal to zero. The declarations $hide$ and $show$ are used to display only the literal $outlier()$ in the answer sets of the algorithm.

Let program $OD$ contain the following rules:

```
%Number of reasonable outliers
const k=3.
num_outliers(0..k).
#domain num_outliers(K).

%K amount of bad observations may possibly occur
rOD(K): bad_obs(K) +-.

%Generate K outliers
K{outlier(Obs):obs(Obs,Time):step(Time)}K :-
    bad_obs(K).

%Only show the relation outlier() in the answer sets
hide. show outlier(X).
```
5.3 Algorithm *DetectOutlierSets*

*KB, OBS, OM* and *OD* present the framework needed to detect outliers of *OBS* with respect to *P = KB ∪ OM* using the algorithm *DetectOutlierSets* shown in Figure 5.1. The algorithm we believe is sound and complete with respect to the definition of an outlier.

| INPUT: A consistent knowledge base *KB*, and a set of observations *OBS* |
| OUTPUT: The algorithm will either return “no outliers”, all outlier sets of *P = KB ∪ OM* and *OBS* or “inaccurate model” |
| METHOD: Perform the following steps: |
| 1. If *π* is consistent then the algorithm returns “no outliers”. |
| 2. If *π* is inconsistent and *π ∪ OD* is consistent let *A_1, A_2, . . . , A_K* be the collection of answer sets of *π ∪ OD*. Let |
|  
| \[ O_i = \{\text{outlier(ObsNumber)} : \text{outlier(ObsNumber)} \subseteq A_i\} \] |
| The algorithm returns |
| \[ O = \{O_1, \ldots, O_i\} \] |
| Note that for every *i*, *O_i* is not empty. |
| 3. Otherwise the algorithm returns “inaccurate model” meaning |
| *KB* does not accurately model the environment. |

**Figure 5.1: Algorithm DetectOutlierSets**

5.4 Outlier Detection Examples

We will now demonstrate the algorithm *DetectOutlierSets* by detecting outliers in the following scenarios from the Store Example.
5.4.1 Detecting Outlier Sets in the Store Example

Recall the Store Example from section 1.3.1. We will use the algorithm DetectOutlierSets and the knowledge base from the Store Example to detect outliers in the following scenarios. The scenarios are run independently of each other.

Let $KB_S$ be the CR-Prolog program in section 2.4.1 that models the store domain, let $OBS_S$ be the set of observations observed in the environment, and let $\pi_s = KB_S \cup OBS_S \cup OM$.

5.4.1.1 Scenario 1

It is observed in the domain that the store is open at time step 1. This information is captured in the observation set $OBS_S$, by adding the facts $obs(r1,1)$ and $content(r1,store\_open,t)$. The algorithm returns "no outliers" because $\pi_s$ is consistent, and no outliers exist in this scenario.

5.4.1.2 Scenario 2

At time step 1, it is observed in the domain that the day is Sunday. This information is captured in $OBS_S$ by adding the facts $obs(r1,1)$ and $content(r1,sunday,t)$. The algorithm returns "no outliers" for this scenario because $\pi_s$ is consistent.
5.4.1.3 Scenario 3

Two conflicting observations were made at time step 1. One observation states that the day is a holiday; the other states that it is not a holiday. This information is captured in the observation set by adding the facts: \( \text{obs}(r_1,1), \text{content}(r_1,\text{holiday},t), \text{obs}(r_2,1) \) and \( \text{content}(r_2,\text{holiday},\neg t) \). Three outlier sets are returned by \( \pi_s \cup OD \), \{outlier(r_1)\}, \{outlier(r_2)\} and \{outlier(r_1), outlier(r_2)\}.

5.4.1.4 Scenario 4

Observations were made in the domain stating that today is Sunday, and the store is open. Both observations were made at time step 1. These statements are captured in the observation set by adding the facts: \( \text{obs}(r_1,1), \text{content}(r_1,\text{sunday},t), \text{obs}(r_2,1) \) and \( \text{content}(r_2,\text{store\_open},t) \). When the algorithm is run, three outlier sets are returned for this scenario, \{outlier(r_1)\}, \{outlier(r_2)\} and \{outlier(r_1), outlier(r_2)\}.

5.4.1.5 Scenario 5

Observations are made in the domain stating that the day is not a Sunday, not a holiday and the store is open. Two other observations state that the weather is bad, and the store is not open. These observations are captured in \( OBS_S \) and represented by the following rules: \( \text{obs}(r_1,1), \text{content}(r_1,\text{sunday},\neg t), \text{obs}(r_2,1), \text{content}(r_2,\text{holiday},\neg t), \text{obs}(r_3,1), \text{content}(r_3,\text{store\_open},t), \text{obs}(r_4,1), \text{content}(r_4,\text{weather},t) \),
obs(r5,1) and content(r5,store_open,f). Eleven outlier sets are returned when the algorithm is run.

\{\text{outlier}(r3)\}  
\{\text{outlier}(r4), \text{outlier}(r5)\}  
\{\text{outlier}(r1), \text{outlier}(r3)\}  
\{\text{outlier}(r2), \text{outlier}(r3)\}  
\{\text{outlier}(r3), \text{outlier}(r5)\}  
\{\text{outlier}(r3), \text{outlier}(r4), \text{outlier}(r5)\}  
\{\text{outlier}(r1), \text{outlier}(r4), \text{outlier}(r5)\}  
\{\text{outlier}(r2), \text{outlier}(r4), \text{outlier}(r5)\}  
\{\text{outlier}(r2), \text{outlier}(r3), \text{outlier}(r5)\}  
\{\text{outlier}(r1), \text{outlier}(r2), \text{outlier}(r3)\}  
\{\text{outlier}(r1), \text{outlier}(r3), \text{outlier}(r5)\}

These eleven sets represent all outlier sets for this scenario.

5.4.2 Explanation of Outlier Sets

The algorithm DetectOutlierSets detects all outlier sets in each scenario. These outlier sets represent every possible set that may occur. We explained earlier that an intelligent agent should believe the outlier set that is most likely to occur, meaning the set with cardinality on the number of outliers. In the next section we will introduce
an algorithm, *DetectMinimalOutlierSets*, that returns only the minimal outlier sets in each of the scenarios.

### 5.5 Detecting Minimal Outlier Sets

In order to detect minimal outlier sets of a scenario, we will replace the program *OD* with the program Minimal Outlier Detector (*MOD*). The program *MOD* contains the same rules as *OD* but a preference rule is added to *MOD* that states: “Prefer answer sets containing minimal outlier sets.” Because *DetectMinimalOutlierSets* contains this preference rule, only the minimal outlier sets of each scenario are returned, unlike *DetectOutlierSets* which returns all outlier sets. This is the key difference between the two algorithms.

Let program *MOD* contain the following rules:

\[
\text{const } k=3. \\
\text{counter}(0..k). \\
\text{#domain counter}(K). \\
\%\text{Bad observations may possibly occur} \\
rOD(K): \text{bad}_\text{obs}(K) \land -. \\
\%\text{Prefer } K \text{ bad observations over } K+1 \\
p\text{refer}(rOD(K),rOD(K+1)) \land - K < k. \\
\%\text{Generate } K \text{ outliers} \\
K\{\text{outlier(Obs):obs(Obs,Time):step(Time)}\}K \land - \text{bad}_\text{obs}(K).
\]
5.6 Algorithm DetectMinimalOutlierSets

KB, OBS, OM and MOD present the framework needed to detect minimal outlier sets of OBS with respect to \( P = KB \cup OM \) using the algorithm DetectMinimalOutlierSets shown in Figure 5.2. If KB does not use preferences, the algorithm is sound and complete with respect to the definition of an outlier. If KB contains preferences, the cr-rules of KB may possibly interfere with the cr-rules of MOD, causing the algorithm to return incomplete answer sets.

INPUT: A consistent knowledge base KB, and a set of observations OBS
OUTPUT: The algorithm will either return “no outliers”, minimal outlier sets of \( P = KB \cup OM \) and OBS or “inaccurate model”
METHOD: Perform the following steps:

Let \( \pi = KB \cup OBS \cup OM \)
1. If \( \pi \) is consistent then the algorithm returns “no outliers”.
2. If \( \pi \) is inconsistent and \( \pi \cup MOD \) is consistent let \( A_1, A_2, ..., A_K \) be the collection of answer sets of \( \pi \cup MOD \). Let \( O_i = \{ outlier(ObsNumber) : outlier(ObsNumber) \subseteq A_i \} \).
   The algorithm returns \( O = \{ O_1, \ldots, O_i \} \).
   Note that for every \( i \), \( O_i \) is not empty.
3. Otherwise the algorithm returns “inaccurate model” meaning \( KB \) does not accurately model the environment.

END

Figure 5.2: Algorithm DetectMinimalOutlierSets
5.7 Minimal Outlier Detection Examples

We will use the algorithm \textit{DetectMinimalOutlierSets} to detect outliers in the Store Example.

5.7.1 Detecting Minimal Outlier Sets in the Store Example

Recall the Store Example from section 1.3.1. We will use the algorithm \textit{DetectMinimalOutlierSets} and the knowledge base from the Store Example to detect outliers in the following scenarios. Each scenario is run independently of the others.

Let $KB_S$ be the CR-Prolog program in section 2.4.1 that models the store domain, let $OBS_S$ be the set of observations observed in the environment, and let $\pi_s = KB_S \cup OBS_S \cup OM$.

5.7.1.1 Scenario 1

It is observed in the domain that the store is open at time step 1. This information is captured in the observation set $OBS_S$, by adding the facts $\text{obs}(r1,1)$ and $\text{content}(r1,\text{store\_open},t)$. When the algorithm is run, “no outliers” is returned since $\pi_s$ is consistent and no outliers exist in this scenario.
5.7.1.2 Scenario 2

It is observed in the domain that the day is Sunday at time step 1. This information is captured in $OBS_S$ by adding the facts $obs(r1,1)$ and $content(r1,sunday,t)$. The algorithm returns “no outliers” for this scenario because no outliers were needed to restore consistency.

5.7.1.3 Scenario 3

Two conflicting observations were made at time step 1. One observation states that the day is a holiday; the other states that it is not a holiday. This information is captured in the observation set by adding the facts: $obs(r1,1)$, $content(r1,holiday,t)$, $obs(r2,1)$ and $content(r2,holiday,f)$. The algorithm $DetectMinimalOutlierSets$ returns two minimal outlier sets for this scenario, \{outlier(r1)\} and \{outlier(r2)\}, stating that either observation $r1$ is an outlier, or observation $r2$ is an outlier.

5.7.1.4 Scenario 4

Observations were made in the domain stating that today is Sunday and the store is open. Both observations were made at time step 1. These statements are captured in the observation set by adding the facts: $obs(r1,1)$, $content(r1,sunday,t)$, $obs(r2,1)$ and $content(r2,store\_open,t)$. The algorithm $DetectMinimalOutlierSets$ computes two minimal outlier sets, \{outlier(r1)\} and \{outlier(r2)\}, stating that either observation $r1$ is an outlier or observation $r2$ is an outlier. Although the set,
\{\text{outlier}(r1), \text{outlier}(r2)\}, is an outlier set according to the definition, it is not returned by the algorithm because it is a not minimal outlier set of the scenario.

5.7.1.5 Scenario 5

Observations are made in the domain stating that today is not a Sunday, not a holiday, and the store is open. Two other observations state that the weather is bad, and the store is not open. These observations are captured in $OBS_S$ in the following rules: $\text{obs}(r1,1)$, $\text{content}(r1,\text{Sunday},f)$, $\text{obs}(r2,1)$, $\text{content}(r2,\text{Holiday},f)$, $\text{obs}(r3,1)$, $\text{content}(r3,\text{store\_open},t)$, $\text{obs}(r4,1)$, $\text{content}(r4,\text{weather},t)$, $\text{obs}(r5,1)$ and $\text{content}(r5,\text{store\_open},f)$. When the algorithm is run only one outlier set is returned, $\{\text{outlier}(r3)\}$, because it is the minimal outlier set for this scenario.

5.7.2 An Example of Incomplete Answer Sets

We mentioned earlier that $DetectOutlierSets$ is sound and complete with respect to the definition of an outlier, but $DetectMinimalOutlierSets$ is sound and complete only if $KB$ does not use preferences. We will now show an example when $DetectMinimalOutlierSets$ is unable to return complete outlier sets of $P = KB \cup OM$ and $OBS$ when $KB$ uses preferences.

Consider the following program $KB_B$ which is a more complex model of the Battlefield environment described in section 4.1 of the Military Example.
% Knowledge Base - KB_B

target(t1).

#domain target(TAR).

step(0..1).

#domain step(T).

poss_exc(missile_malfunction; human_error).

#domain poss_exc(E).

fluent(destroyed(TAR)).

#domain fluent(F).

%Normally when a target is attacked it is destroyed

\[ h(destroyed(TAR), T+1) :- o(attack(TAR), T), 
\]

\[ \text{not exception}(T). \]

except(T) :- failure(E, T).

%Exceptions may occur

\[ \text{count}(0..3). \quad \#\text{domain count}(C). \]

\[ r1(C) : \text{num}_{-}e_{x}_{c}(C) +-. \]

\[ \text{prefer}(r1(C), r1(C+1)). \]

\[ C\{\text{failure}(EXC, Time) : \text{poss}_{-}e_{x}_{c}(EXC) : \text{step}(Time)\} C :- \]

\[ \text{num}_{-}e_{x}_{c}(C). \]

%History

\[ o(attack(t1), 0). \]
The default of the knowledge base states that “normally when a target is attacked it is destroyed unless a failure occurs.” The cr-rule \( r1(C) \) states: “Failures may possibly occur, but they are rare.” The preference rule states that “answer sets with fewer exceptions are preferred over answer sets with more exceptions.” The history section records that target \( t1 \) was attacked at time step 0.

Notice that the encoding of the cr-rules is slightly different from the example in the previous section. This example is a more complex representation of the Battlefield environment since it states that attacks may fail for multiple reasons; whereas the program representing this environment in the previous chapter simply stated that “attacks may possibly fail.”

In this scenario the observation set \( OBS_B \) contains three reports, two of which say the target \( t1 \) was not destroyed, and one contradictory report which says that \( t1 \) was destroyed at time step 1. These reports are contained in the observation set \( OBS_B \).

\[
% Observations - OBS_B
\]
\[
obs(r1,1). \ content(r1,destroyed(t1),f).
\]
\[
obs(r2,1). \ content(r2,destroyed(t1),t).
\]
\[
obs(r3,1). \ content(r3,destroyed(t1),f).
\]

According to the definition of an outlier, five outlier sets exist in this scenario:

\{outlier(r2)\}, \{outlier(r1), outlier(r3)\}, \{outlier(r1), outlier(r2)\}, \{outlier(r2),
outlier(r3)} and \{outlier(r1), outlier(r2), outlier(r3)\}. Let’s see what happens when \(KB_B\) and \(OBS_B\) are input into the algorithm \textit{DetectMinimalOutlierSets}.

Under the semantics of CR-Prolog, \(KB_B \cup OBS_B \cup OM\) is inconsistent and \(KB_B \cup OBS_B \cup OM \cup MOD\) has the following views:

\begin{align*}
\text{v1} &= \{\text{bad}_\text{obs}(1), \text{outlier}(r2), \text{num}_\text{exc}(1), -h(\text{destroyed}(t1),1), \ldots \} \\
\text{v2} &= \{\text{bad}_\text{obs}(1), \text{outlier}(r2), \text{num}_\text{exc}(2), -h(\text{destroyed}(t1),1), \ldots \} \\
\text{v3} &= \{\text{bad}_\text{obs}(2), \text{outlier}(r1), \text{outlier}(r3), h(\text{destroyed}(t1),1), \ldots \} \\
\text{v4} &= \{\text{bad}_\text{obs}(2), \text{outlier}(r1), \text{outlier}(r3), \text{num}_\text{exc}(0), h(\text{destroyed}(t1),1), \ldots \} \\
\text{v5} &= \{\text{bad}_\text{obs}(2), \text{outlier}(r1), \text{outlier}(r2), \text{num}_\text{exc}(1), -h(\text{destroyed}(t1),1), \ldots \} \\
\text{v6} &= \{\text{bad}_\text{obs}(2), \text{outlier}(r1), \text{outlier}(r2), \text{num}_\text{exc}(2), -h(\text{destroyed}(t1),1), \ldots \} \\
\text{v7} &= \{\text{bad}_\text{obs}(2), \text{outlier}(r2), \text{outlier}(r3), \text{num}_\text{exc}(1), -h(\text{destroyed}(t1),1), \ldots \} \\
\text{v8} &= \{\text{bad}_\text{obs}(2), \text{outlier}(r2), \text{outlier}(r3), \text{num}_\text{exc}(2), -h(\text{destroyed}(t1),1), \ldots \} \\
\text{v9} &= \{\text{bad}_\text{obs}(3), \text{outlier}(r1), \text{outlier}(r2), \text{outlier}(r3), h(\text{destroyed}(t1),1), \ldots \} \\
\text{v10} &= \{\text{bad}_\text{obs}(3), \text{outlier}(r1), \text{outlier}(r2), \text{outlier}(r3), \text{num}_\text{exc}(0), h(\text{destroyed}(t1),1), \ldots \} \\
\text{v11} &= \{\text{bad}_\text{obs}(3), \text{outlier}(r1), \text{outlier}(r2), \text{outlier}(r3), \text{num}_\text{exc}(1), -h(\text{destroyed}(t1),1), \ldots \} \\
\text{v12} &= \{\text{bad}_\text{obs}(3), \text{outlier}(r1), \text{outlier}(r2), \text{outlier}(r3), \text{num}_\text{exc}(2), -h(\text{destroyed}(t1),1), \ldots \} 
\end{align*}

Views \text{v1} and \text{v2} are preferred over views \text{v3} through \text{v12} because \text{bad}_\text{obs}(1) is preferred over \text{bad}_\text{obs}(2) and \text{bad}_\text{obs}(3). However, views \text{v4} and \text{v10} are pre-
ferred over views $v_1$ and $v_2$ because $num_{\text{exc}}(0)$ is preferred over $num_{\text{exc}}(1)$ and $num_{\text{exc}}(2)$. Because all of the views are in some way preferred over each other, no answer sets are returned under the semantics of CR-Prolog. This example shows a scenario when the algorithm $DetectMinimalOutlierSets$ does not return complete outlier sets according to the definition of an outlier.
In this section, we will compare the definition of an outlier presented in this research in section 3.2 with the definition of an outlier presented in (Angiulli, Greco, and Palopoli 2004). We will start by reviewing both definitions, and then apply the definitions to a few examples to compare the results.

6.1 Definitions

Let us start by reviewing the definition of an outlier presented earlier in this work. For comparison we will refer to the definition from section 3.2 as Definition 1. 

**Definition 1 (Outlier Set) from Section 3.2.**

Given a consistent logic program $P$ of CR-Prolog, and a set of observations $\text{Obs}$, a set $O \subseteq \text{Obs}$ is called an outlier set of $P$ with respect to $\text{Obs}$ if:

1) $P \cup \text{Obs}$ is inconsistent and

2) $P \cup \text{Obs} \setminus O$ is consistent.

Elements of the outlier set are called outliers.

Now let us look at the definition of an outlier as presented by Angiulli, Greco and Palopoli which we will refer to as Definition 2.
Definition 2 (Outlier) (Angiulli, Greco, and Palopoli 2004).

Let \( P = \langle P_{rls}, P_{obs} \rangle \) be a rule-observation pair relating the general knowledge encoded in \( P_{rls} \) with the evidence about the world encoded in \( P_{obs} \). \( O \subseteq P_{obs} \) is called an outlier, under cautious (resp. brave) semantics, in \( P \) if there exists a non empty set \( W \subseteq P_{obs} \), called outlier witness for \( O \) in \( P \), such that:

1. \( P(P)_W \models_c \neg W \) (resp. \( P(P)_W \models_b \neg W \)), and
2. \( P(P)_{W,O} \not\models_c \neg W \) (resp. \( P(P)_{W,O} \not\models_b \neg W \)).

where \( P(P) = P_{rls} \cup P_{obs} \), \( P(P)_W = P(P) \setminus W \) and \( P(P)_{W,O} = P(P)_W \setminus O \).

In these two definitions, input is given as an arbitrary logic program, \( P \) for Definition 1 and \( P_{rls} \) for Definition 2 and observations are recorded in the observation set, \( Obs \) for Definition 1 and \( P_{obs} \) for Definition 2. The key difference between the two definitions is that Definition 2 uses a witness set \( W \), to define outliers while Definition 1 does not. Having reviewed both definitions of an outlier, we will now look at the following examples.

6.2 Simple Example

Let’s look at a very simple example program in which we will compare outliers according to both definitions. An arbitrary logic program, \( P \) for Definition 1 and \( P_{rls} \) for Definition 2, consists of the following rules:
6.2.1 Equivalent Outlier Sets

Suppose observations \( \neg b \) and \( c \) were observed in the domain and are contained in the observation set. When the program and the observation set are combined, the program becomes inconsistent. We will test to see if the observation \( c \) is an outlier according to both definitions starting with Definition 1.

Let \( \text{Obs} = \{\neg b,c\} \) and \( O = \{c\} \).

1. \( P \cup \text{Obs} \) is inconsistent.

\( P \cup \{\neg b,c\} \) has no answer sets.

The first condition holds.

2. \( P \cup \text{Obs} \setminus O \) is consistent.

\( P \cup \{\neg b\} \models \{a,\neg b\} \).

The second condition holds and we conclude that \( c \) is an outlier according to Definition 1. Now we will test to see if \( c \) is an outlier according to Definition 2.

Let \( \text{Obs} = \{\neg b,c\} \), \( O = \{c\} \) and \( W = \{\neg b\} \).

1. \( P^{rls} \cup P^{obs} \setminus W \models \neg W \) holds because

\( P^{rls} \cup \{c\} \models \{a,b,c\} \).

2. \( P^{rls} \cup P^{obs} \setminus W \setminus O \neq \neg W \) holds because

\( P^{rls} \cup \{\} \models \{a\} \).
Because both conditions hold, we conclude that \{c\} is an outlier according to Definition 2 and has a witness set of \{¬b\}.

6.2.2 Differing Outlier Sets

Suppose in a different scenario, the fact \(¬a\) was observed and stored in the observation set such that \(Obs = \{¬a\}\) and \(P^{obs} = \{¬a\}\) in this simple example. Intuitively one can see that the observation \(¬a\) is an outlier with respect to the program. We will apply both definitions to the combination of the program and the observation set to determine outliers in this example.

Let us show that \(\{¬a\}\) is an outlier according to Definition 1.

1. \(P \cup Obs\) is inconsistent.
   
   \(P \cup \{¬a\}\) has no answer sets

2. \(P \cup Obs \setminus O\) is consistent.
   
   \(P \cup \{\} \models \{a,b\}\).

We conclude that \(\{¬a\}\) is an outlier according to Definition 1.

Let us test if the observation \(¬a\) is an outlier according to Definition 2. The witness set \(W \subseteq P^{obs}\) cannot be empty according to the definition so we will attempt to find an outlier by setting \(W = \{¬a\}\), the only literal in the observation set.

1. \(P_{rls} \cup P^{obs} \setminus W \models ¬W\) holds because

   \(P_{rls} \cup \{\} \models \{a\}\).

2. \(P_{rls} \cup P^{obs} \setminus W \setminus O \not\models ¬W\).
\[ P^{rls} \cup \{ \} \models \{a\}. \]

Condition 2 fails because it entails \( \neg W \).

No other witness sets can be established for this example. Thus we can conclude that no outliers exist according to Definition 2, despite our intuitive belief that the observation \( \{\neg a\} \) is an outlier. From this example we can conclude the following observation.

6.2.3 Observation

*Definition 1 and Definition 2 are not equivalent.*

We will now present a more complex example showing the similarities and differences of both definitions by returning to the Network Example.

6.3 Network Example Revisited

In the following scenarios of the Network Example we will examine occurrences in which a literal is an outlier according to one definition but is not according to the other.

6.3.1 Network Scenario

To gain a deeper understanding of Definition 2, we will recall the Network Example from section 3.3.3 in which we have already shown that \( \neg on(c) \) is an outlier
according to Definition 1. Now we will show that \( \neg on(c) \) is an outlier according to Definition 2.

Let \( P^{rls} \) be the program shown in section 3.3.3.1.

\[
P^{obs} = \{\neg on(a), on(b), \neg on(c), on(d), on(e), on(f), on(g), on(h), on(t)\},
\]

\( O = \{\neg on(c)\} \) and the witness set \( W = \{on(d), on(e), on(g), on(f), on(t)\}. \)

1. \( P^{rls} \cup P^{obs} \setminus W \models \neg W \) holds because

\[
P \cup \{\neg on(a), on(b), on(h), \neg on(c)\} \models \{\neg on(d), \neg on(e), \neg on(g), \neg on(f), \neg on(t)\}
\]

2. \( P^{rls} \cup P^{obs} \setminus W \setminus O \not\models \neg W \) holds because

\[
P^{rls} \cup \{\neg on(a), on(h), on(b)\} \models \{on(d), on(e), on(g), on(f), on(t)\}
\]

We can conclude that \( \neg on(c) \) is an outlier and has a witness set of \( W = \{on(d), on(e), on(g), on(f), on(t)\} \) according to Definition 2.

As shown in the above example and in section 3.3.3.2, the literal, \( \neg on(c) \), is an outlier according to both definitions. Notice again that Definition 1 was able to detect this outlier without requiring a witness set \( W \).

### 6.3.2 Multiple Outlier Sets

In the Network Example the observation \( \neg on(c) \) is the minimal outlier set. As stated in the paper (Angiulli, Greco, and Palopoli 2004), according to Definition 2 every subset of \( \{\neg on(a), on(h), on(b), \neg on(c)\} \) containing \( \neg on(c) \) is also an outlier of the Network Example. According to Definition 1, every subset of \( \{\neg on(a), on(h), \)
on(b), ¬on(c), on(d), on(e), on(f), on(g), on(h)} containing ¬on(c) is an outlier set. Notice that the outlier sets of Definition 1 may include parts of the witness set from Definition 2.

6.3.3 Another Scenario

Suppose it is observed in the network (Fig 3.1) that computers s and t are on but all of the other computers are off. The following information is recorded in the observation set: \{¬on(a), ¬on(b), ¬on(c), ¬on(d), ¬on(e), ¬on(f), ¬on(g), ¬on(h), on(t)\}. Intuitively, in this scenario, the computer t is acting outside of the social “norm” by being on and is an outlier.

Let us show that on(t) is an outlier according to Definition 1:

Let O = \{on(t)\}.

1. \(P \cup \text{Obs}\) is inconsistent

\(P \cup \{¬\text{on(a)}, ¬\text{on(b)}, ¬\text{on(c)}, ¬\text{on(e)}\}\) has no answer sets

2. \(P \cup \text{Obs} \setminus O\) is consistent

\(P \cup \{¬\text{on(a)}, ¬\text{on(b)}, ¬\text{on(h)}, ¬\text{on(c)}\}\) is consistent and entails \{¬on(t)\}

From this example we can conclude that the observation literal on(t) is an outlier according to Definition 1. Because computer t is at the end of the network (Fig 3.1) and no other computers are affected by it being on or off, no witness set can be established. Definition 2 is unable to produce any outliers for this scenario.
6.4 Comparison Summary

As shown in these examples, both definitions are capable of determining outliers in most scenarios given a consistent logic program and a pertaining set of observations. We feel, however, that Definition 1 better models our intuition of an outlier than does Definition 2 as shown through these examples.
CONCLUSIONS AND FUTURE WORK

7.1 Summary

In this work we introduced methods for programming intelligent systems capable of handling potentially unsound information in a reasonable manner. We started by describing, in general, what an outlier is, then gave some intuitive examples. In the second chapter we provided an overview of CR-Prolog, which included background information, syntax and functionality of the language. We also introduced the previous work of (Angiulli, Greco, and Palopoli 2004). In chapter three we presented the formal definition of an outlier set and explained a few simple examples.

The Military Example was introduced in chapter four in which we showed how to detect outliers within the environment and determine the reliability of the information gathering outside sources. In chapter five we introduced two algorithms, DetectOutlierSets and DetectMinimalOutlierSets, which detect all outlier sets and minimal outliers sets respectively. We made comparisons in the last section between definitions presented in this work and definitions from previous research. Finally, we compared and contrasted these definitions with a few examples.

7.2 Future Work

The work of this thesis generated many new questions and problems that will need to be solved to expand on this research. We will briefly look at a few of these
areas. The biggest limitation I discovered in this work was the inability to prefer one preference rule over another preference rule. Solving this problem would greatly increase the programming power of CR-Prolog and would also allow the algorithm DetectMinimalOutlierSets to detect complete minimal outlier sets in all CR-Prolog knowledge bases regardless of whether or not they use preferences.

Another major area of expansion lies in creating more elaborate systems that are capable of preferring reliable sources over unreliable sources. This type of reasoning would help maintain the soundness of the decision-making process. An even larger project would involve developing a system to detect people who deliberately lie. The most dangerous sources of information come from those who purposely wish to sabotage the system. These deceptive sources submit correct information as often as possible to gain trust, but will then intentionally give false information during crucial times in the decision-making process. To detect these deceptive sources, an intelligent system would need to monitor agent reports specifically during crucial decision-making times within the environment. The system could thus detect agents who were problematic at crucial times in the system but were reliable in non-critical situations.

Finally, we need to explore the full functionality of CR-Prolog to discover all of its capabilities as well as its limitations. When the full functionality of CR-Prolog is documented, we will gain a clearer view of which problems and tasks we may be able to solve using CR-Prolog.
REFERENCES


Detecting Suspicious Input in Intelligent Systems Using Answer Set Programming

A Thesis Defense by Nicholas Giancuntoos

December 8, 2004
PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master’s degree at Texas Tech University or Texas Tech University Health Sciences Center, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Agree (Permission is granted.)

Nicholas Gianoutsos          April 12, 2005

Disagree (Permission is not granted.)

__________________  ____________