A DECLARATIVE FRAMEWORK FOR MODELING MULTI-AGENT SYSTEMS

by

GREGORY GELFOND

A MASTER 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

Dr. Richard Watson
Committee Chairman

Dr. Daniel Cooke

Dr. Nelson Rushton

John Borrelli
Dean of the Graduate School

May, 2007

ACKNOWLEDGMENTS

I read once that one of the keys to happiness is having something that you are grateful for. During the course of my work I have found much to be thankful for, and am grateful to very many people.

I would like to begin by thanking my advisor, Dr. Richard Watson, for his guidance, friendship, and for giving me the freedom to explore many diverse avenues of research in the field. I would also like to thank my committee members, Dr. Daniel Cooke and Dr. Nelson Rushton, for their generous feedback, time, and many conversations on a wide range of topics. I have learned a great deal from all of them, and working with them has been a pleasure.

I would like to thank the members of the KRLab for their friendship and support. Marcello and Veena for always being there to listen, answer questions, and cheer me on. Jarred and Justin for many pleasant afternoon discussions and study sessions. There are many others, members both past and present that have my thanks.

I would like to thank my parents. They have been a great source of inspiration to me and a great help. They have taught me much about the world and have opened my eyes to a great deal of its beauty and wonder. I would also like to thank my sister Yulia and her husband Patrick. In addition to being a constant source of encouragement they have brought into this world my niece Alexandra and my nephew Johnathan which are a great source of joy in my life.

I also wish to thank the many people whom I consider to be friends and teachers, Dr. Lifschitz and his wife Elena, Dima and Taya Lenkov, Victor Rivkin, and many others. All of them have my life richer in their own ways, and them I am deeply grateful.

I would also like to thank Kim for being one of the greatest joys of my life, and

for her love, friendship, and so much more.

There are many others whom I would like to thank. Friends and family, some of whom are no longer with us, all of whom have helped shape me in some way. Thank you all.

CONTENTS

LIST OF FIGURES

ABSTRACT

Current work in answer-set programming with regards to its application in the development of reasoning agents has centered around single-agent systems. A well established body of research showing its applicability towards such domains has been developed, describing a thorough methodology for their development upon a theoretical foundation. This work hopes to expand the applicability of this field to the realm of multi-agent domains.

In this work we present a general framework for reasoning about cooperative multi-agent systems. We begin with an overview of the current framework for representing single-agent systems as well as the syntax and semantics of the logic programming language CR-Prolog. Once this baseline has been established, we extend the fundamental notion of an agent to facilitate communication via the introduction of special named sets of fluents known as requests. We then define the notions of an agent's local and global perspectives and their respective diagrams which serve as the theoretical foundation of this work.

Once the general framework has been discussed, a motivating example of a simple multi-agent domain is presented. This example is used to develop a methodology for representing agents capable of reasoning in such domains using the logic programming language of CR-Prolog, together with an axiomatization of multi-agent communication.

Finally a series of results detailing some fundamental properties of the framework are presented.

CHAPTER 1

INTRODUCTION

1.1 RESEARCH GOALS

Current work in answer-set programming with regards to its application in the development of reasoning agents has centered around single-agent systems. A well established body of research showing its applicability towards such domains has been developed describing a thorough methodology for their development upon a theoretical foundation.

The goal of this research is to expand the current declarative frameworks used for reasoning about domains containing a single agent to domains containing multiple agents in an attempt to answer the following question: "How do we develop programs that may intelligently ask other programs to perform various tasks?" A general framework for reasoning about multi-agent systems is presented, as well as its application in the design and operation of a collaborative multi-agent domain.

We begin by extending our notion of what constitutes an agent. Previous work on the development of single-agent systems largely abstracted out the means by which an agent might communicate with the outside world. In order to accommodate the ability of agents to communicate between themselves, the definition of an agent has been extended via the addition of named sets of fluents known as *requests*. Requests provide a *language for communication* between the agents that may be present in a given system. Depending on the nature of the particular system being represented, these requests may be used in the formation of *messages* which the agents may pass between themselves.

Having established this basic building block, we then formalize the notion of a multi-agent system as a set of agents which satisfies various properties. In general

there are many types of multi-agent systems. Such systems may contain agents which work independently, collaborate to achieve their goals, compete against each other, or even contain a mixture of all three. This work will largely focus on formalizing an agent's reasoning within a system in which the agents cooperate in order to achieve their various goals. Such a system is called a *collaborative multi-agent system*.

Once these basic notions have been introduced, a formalization of a class of actions known as *message passing* actions is described. Depending on the particular task at hand, an agent reasons about the effects of such actions from either a *local perspective*, or a *global perspective*. These modes of reasoning are described by an artifact called the *communication module*, which uses a combination of action languages and logic programming under the answers-set semantics as its foundation. The particular languages used are the action language $\mathcal{AL}$, and the answer-set programming language CR-Prolog.

The remainder of the introduction is organized as follows: a brief overview of the action language $\mathcal{AL}$ will be presented, followed by a description of the syntax and semantics of the relevant portion of CR-Prolog. Once this has been done, a description of the current framework for modeling single-agent systems will be given.

1.2 THE ACTION LANGUAGE $\mathcal{AL}$

Briefly stated, action languages are a class of declarative languages used for describing the effects of actions. They have a simple syntax and semantics, and yet remain powerful enough to represent many complex reasoning domains [1]. Collections of statements in an action language are termed *action descriptions*, and define transition diagrams whose nodes correspond to possible *states of the*

world, and whose arcs are labeled by actions. Intuitively, an arc $\langle \sigma, a, \acute{\sigma} \rangle$ states that if action a occurs in state σ, the resulting state will be $\acute{\sigma}$. In addition to specifying the effects of actions, it is necessary to specify what is left *unchanged* by the occurrences of actions. This is known as the *frame problem* [13], and its solution lies in an elegant and precise representation of the *inertia axiom*. The beauty and utility of action languages stems from their ability to provide both a concise representation of huge transition systems, and in their ability to gracefully solve the frame problem.

Action descriptions of $\mathcal{AL}$ are comprised of collections of *dynamic causal laws*, *static causal laws*, and *impossibility conditions*. Dynamic causal laws are statements of the form:

$$a \textbf{ causes } f \textbf{ if } l_0, \ldots, l_n$$

where a is an action and f and $l_0, \ldots, l_n$ are fluent literals. Laws of this form are read as "action a causes f to be true if $l_0, \ldots, l_n$." The corresponding arc in the transition diagram is as follows:

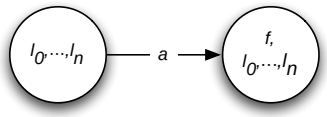

Figure 1.1: transition defined by a simple causal law

Static causal laws have the form:

$$\textbf{caused } f \textbf{ if } l_0, \ldots, l_n$$

where f and $l_0, \ldots, l_n$ are fluent literals. Static causal laws (also known as *state constraints*) are read as: "f is true whenever $l_0, \ldots, l_n$ are true." Unlike dynamic causal laws, state constraints define properties of states, rather than the direct

effects of an action. They may be used however to specify the indirect effects of actions as in the following example:

Example 1.2.1. Consider the following action description, AD_1:

$$AD_1 = \left\{ \begin{array}{l} a \textbf{ causes } f. \\ \\ \textbf{caused } g \textbf{ if } f. \end{array} \right\}$$

and state $\sigma = \{\neg f, \neg g\}$. If a is executed in σ, we have the following transition:

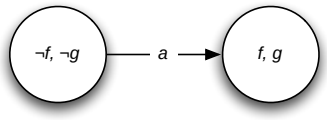

Figure 1.2: transition containing *direct* and *indirect* effects

Note that the value of f changes as a result of the dynamic causal law, but the value of g changes due to the presence of the state constraint.

Impossibility conditions (also known as *executability conditions*) have the form:

$$a \textbf{ impossible if } l_0, \ldots, l_n$$

where as before, a is an action and $l_0, \ldots, l_n$ are fluent literals. Rules such as this are used to state that "action a may not occur if $l_0, \ldots, l_n$ are true." From the standpoint of the transition system, such rules specify that an outgoing arc labeled by a may not originate in a state that satisfies $l_0, \ldots, l_n$.

The semantics of an action description of $\mathcal{AL}$ is given by its transition diagram. For a detailed description of the semantics of $\mathcal{AL}$, the reader is referred to [1].

1.3 CR-PROLOG

CR-Prolog is an extension of the logic programming language A-Prolog developed by Michael Gelfond and Vladimir Lifschitz in [11] which introduces the notion

of *consistency-restoring rules* and the ability to assign *preferences* over them. When taken together these new constructs allow for a more graceful handling of planning and diagnosis. For a complete description of the syntax and semantics of CR-Prolog the reader is referred to [2]. What follows is a description (taken in part from the one presented in [10] with permission of the author) of those constructs which are relevant to the work presented in this thesis. A program of this subset of CR-Prolog is a collection of rules of the following form:

$$l_0 \text{ or } \ldots \text{ or } l_k \leftarrow l_{k+1}, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$$

and

$$r: \ l_0 \text{ or } \ldots \text{ or } l_k \stackrel{+}{\leftarrow} l_{k+1}, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$$

where $l_0 \ldots l_n$ are literals, not is *negation-as-failure* (also known as *default negation*), and r is the name of a rule. Rules of the first form are termed *regular rules*, and are read as: "if one has reason to believe in $l_{k+1}, \ldots, l_m$, and no reason to believe in $l_{m+1}, \ldots, l_n$, then one must believe in one element of $l_0 \ldots l_k$." Rules of the second form are called *consistency restoring rules*, (also known as *cr-rules*), and are read as "if one has reason to believe in $l_{k+1}, \ldots, l_m$, and no reason to believe in $l_{m+1}, \ldots, l_n$, then one may possibly believe in one element of $l_0 \ldots l_k$." In addition, there is an underlying assumption that such rules are used as little as possible.

Having given an overview of the syntax of the language, a brief description of its semantics is given. It should be noted that this discussion assumes that the reader is familiar with the semantics of A-Prolog. Given a CR-Prolog program Π, we denote the set of regular rules of Π by Π^r. Similarly, the set of cr-rules of Π is denoted by Π^{cr}. In addition, let $\alpha(r)$ denote the regular rule obtain from the cr-rule by replacing the symbol $\stackrel{+}{\leftarrow}$ with $\leftarrow$. α is extended in a similar vein to apply to sets of cr-rules.

Definition 1.3.1 (abductive support)**.** Given a CR-Prolog program Π, a minimal (with respect to set theoretic inclusion) set R of cr-rules of Π, such that $\Pi^r \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an *abductive support* of Π.

Definition 1.3.2 (answer set of a CR-Prolog program)**.** Given a CR-Prolog program Π, a set of literals, A, is called an *answer set* of Π, if it is an answer set of a regular program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.

Example 1.3.1. Consider the following CR-Prolog program:

$$\Pi_1 = \left\{ \begin{array}{l} p(X) \leftarrow not\ ab(X). \\ s(X) \leftarrow p(X),\ q(X). \\ r(X) : ab(X) \stackrel{+}{\leftarrow}. \\ ab(1). \\ q(2). \end{array} \right\}$$

Π_1 includes a default with an exception, 1, a partial definition of s in terms of p and q, and a consistency restoring rule which states that there may possibly be unknown exceptions to the default. As the applicability of cr-rules is governed by the assumption that such rules are used as little as possible, the cr-rule is not applied, and Π_1 has only a single answer set, namely $\{p(2), s(2), q(2), ab(1)\}$.

Now let us consider an example where the cr-rule is applicable.

Example 1.3.2. Consider the program $\Pi_2 = \Pi_1 \cup \{\neg s(2)\}$.

$$\Pi_2 = \left\{ \begin{array}{l} p(X) \leftarrow not\ ab(X). \\ s(X) \leftarrow p(X),\ q(X). \\ r(X) : ab(X) \stackrel{+}{\leftarrow}. \\ ab(1). \\ q(2). \\ \neg s(2). \end{array} \right\}$$

The addition of the fact $\neg s(2)$ causes the cr-rule to become applicable, and consequently Π_2 has the answer set $\{\neg s(2), q(2), ab(1), ab(2)\}$.

Before proceeding it must be noted that in order for Π_1 and Π_2 to be executable under current inference systems such as *crmodels* [3], they must be suitably encoded in an extension of the language of *LParse* [3, 14]. For an in-depth description of the LParse system the reader is referred to [14]. The differences are largely cosmetic as the following representation of Π_2 demonstrates:

```
#const n = 2.
object(1..n).
#domain object(X).
p(X) :- not ab(X).
s(X) :- p(X), q(X).
r(X) : ab(X) +-.
ab(1).
q(2).
-s(2).
```

The first three lines are used to specify the values over which the variable X ranges, in this case the set $\{1, \ldots, n\}$ where $n = 2$. This information is necessary in order for the system to automatically ground the rules and generate answer sets. All subsequent program examples will be presented as runnable programs, i.e. in the extended language of LParse.

1.4 MODELING SINGLE-AGENT SYSTEMS

Now that we have established some of the action language $\mathcal{AL}$ and CR-Prolog we now proceed with a brief overview of the current approach towards modeling single-agent systems. For a detailed account of how such systems are modeled the reader is referred to [1, 7].

In general, the following assumptions are made when developing single-agent systems [1]:

- ▷ The agent's environment can be represented as a transition diagram whose states are sets of fluents and whose arcs are labeled by actions.
- ▷ The agent is capable of making correct observations, performing actions, and remembering the history of the domain.

These assumptions define an agent as being comprised of three distinct components: an action description, a set of observations, and a domain history. In order to perform actual reasoning tasks, these components are often described as logic programs written in A-Prolog or one of its many variants. Each of these components is used in various stages of what has been termed the *agent-loop*.

An agent begins by observing the state of the world around it. If these observations fail to match up with its expectations, the agent performs a step known as *diagnosis*, in which it attempts to update its domain history in order to explain

the discrepancies through the occurrence of *exogenous actions* (actions that are not executed by the agent himself). Afterwards the agent selects a goal, and by *planning*, constructs a sequence of actions to achieve its goal. Having done so, the agent executes the first element of this sequence, and repeats the process.

Such systems have been used in many domains ranging from query answering [6], to modeling complex flight control systems of the Space Shuttle [12]. These systems do not model domains in which multiple agents may be operating however. Such domains are quite common, and the extension of the current methodology for developing single-agent systems to multi-agent domains is the focus of this work.

CHAPTER 2

THE MULTI-AGENT FRAMEWORK

2.1 SYSTEMS OF AGENTS

When reasoning about systems which may contain many agents, there is generally an underlying assumption that agents may communicate amongst themselves. Such communication is typically used by agents to ask others in the domain to perform various tasks. In order to accommodate the ability of agents to communicate, the definition of an agent must be extended. This definition is extended by the introduction of *requests*, which may be thought of as a *language for communication* between the agents that may be present in a given system. Depending on the nature of the domain being represented, these requests may be used in the formation of *messages* which the agents may pass between themselves.

Definition 2.1.1 (agent)**.** An *agent*, α, is defined as a 4-tuple $\langle F, A, R, D\rangle$ where:

- $\triangleright$ F is a set of *fluents*.
- $\triangleright$ A is a set of *elementary actions*.
- $\triangleright$ R is a collection of named sets of fluents from F known as *requests*.
- $\triangleright$ D is an action description in the language of $\mathcal{AL}$ with signature $\Sigma = F \cup A$.

Given an agent α, F_α denotes the set of α's fluents, A_α denotes the set of α's actions, etc.

Now that the notion of an agent has been defined, the notion of a *multi-agent system* may be introduced.

Definition 2.1.2 (multi-agent system)**.** A *multi-agent system*, M, is defined as a finite, non-empty set of agents such that:

$$\forall \alpha, \beta \in M, F_\alpha \cap F_\beta = \emptyset \wedge A_\alpha \cap A_\beta = \emptyset.$$

On its surface, the condition that the agents of a multi-agent system have disjoint sets of actions seems restrictive. This restriction can be overcome however by associating an actor with each action. The restriction upon fluents is in place for a myriad of reasons, among them being a desire to have the agents be as independent of each other as possible in order to preserve as much of the existing single-agent framework as possible. While for some domains this restriction may be overly restrictive, lifting this restriction raises a number of theoretical issues which are outside the scope of this work. This is a syntactic restriction however, and it is still possible to construct systems of homogenous agents.

In general there are many types of multi-agent systems. Such systems may contain agents which work together to achieve their goals, compete against each other, or even contain a mixture of collaborative and competitive agents. This work will focus on collaborative multi-agent systems which are defined below:

Definition 2.1.3 (collaborative multi-agent system)**.** A *collaborative multi-agent system*, M, is defined as a multi-agent system, combined with the 3-tuple $\langle \mathcal{C}, \mathcal{S}, \mathcal{M} \rangle$ where:

- $\mathcal{C}$ is a function which given an agent α and a request $r \in R_\alpha$, returns a set of agents known as the *clients* of r.
- $\mathcal{S}$ is a function which given an agent α and a request $r \in R_\alpha$, returns a set of agents known as the *servers* of r.
- $\mathcal{M}$ is a set of ordered pairs of the form $\langle r, \beta \rangle$ known as *messages* such that $\beta \in M$, $r \in R_\alpha$ for some $\alpha \neq \beta \in M$, and $\beta \in \mathcal{S}(\alpha, r)$

In addition to the above, $\mathcal{C}$, $\mathcal{S}$, and $\mathcal{M}$ must satisfy the following properties:

▷ If $\beta \in \mathcal{C}(\alpha, r)$ then $\forall \gamma \in M,\ \beta \notin \mathcal{S}(\gamma, r)$.

▷ If $\beta \in \mathcal{C}(\alpha, r)$ then $\alpha \in \mathcal{S}(\beta, r)$.

▷ If $\beta \in \mathcal{S}(\alpha, r)$ then $\alpha \in \mathcal{C}(\beta, r)$.

2.2 AGENT COMMUNICATION

An agent's knowledge defines a pair of transition diagrams known as its *local* and *global* diagrams. These diagrams are used by the agent to reason about its actions from what are termed its *local* and *global* perspectives. These perspectives differ on how actions which involve message passing are handled. The communication module will be discussed in detail in subsequent chapters, for the moment, suffice it to say that it is comprised of two sub-modules, the *local module*, C_{local}, and *global module*, C_{global}. C_{local} is defined by an action description in the language $\mathcal{AL}$, while C_{global} is defined by a logic program in CR-Prolog.

2.2.1 *THE LOCAL PERSPECTIVE*

Consider the following simple scenario: "John wants to prepare for a trip. In order to do so he must pack his bags and obtain a ticket. John has a secretary that is able to obtain a ticket for him." How would John reason about preparing for his trip? Assuming that John has a competent secretary it is reasonable to assume that John's reasoning can be summarized as follows:

▷ John is ready when he has his tickets and is packed.

▷ Packing bags causes them to be packed.

▷ Having his secretary purchase tickets for him causes him to have tickets.

The first two statements can be captured by the following static and dynamic causal laws of $\mathcal{AL}$:

$$\textbf{caused } ready \textbf{ if } packed,\ ticket.$$

$$pack \textbf{ causes } packed.$$

What about the last statement? Recall that when we are representing agents we only represent their knowledge of the domain. John is not concerned with how his secretary obtains the tickets. He only knows that when he has her purchase them, he will have them. This line of reasoning can be captured by the following dynamic causal law:

$$ask \textbf{ causes } ticket.$$

This type of reasoning is termed *reasoning from a local perspective*. When reasoning in this perspective an agent abstracts out all of the details concerning the actions that other agents may take. The sole focus is how his requests affect his own model of the world. This approach is generalized through the use of *requests*. Recall that requests are named sets of fluents. To make this law more general, and therefore applicable to other commands that John may at some point issue to his secretary, we introduce into our representation the request $buy(ticket, john)$ which contains the fluent $ticket$. With the request in place, we can now represent the final statement via the following dynamic causal law:

$$send(\mathrm{Request}) \textbf{ causes } \mathrm{Fluent} \textbf{ if } \mathrm{Fluent} \in \mathrm{Request}.$$

This is precisely what the local communication module, C_{local} accomplishes. In this framework, the representation of John would be as follows:

- $F_{john} = \{ticket, packed, ready\}$.
- $A_{john} = \{pack\}$.
- $R_{john} = \{buy(ticket, john) = \{ticket\}\}$.

$$\triangleright\ D_{john} = \left\{ \begin{array}{l} \textbf{caused}\ ready\ \textbf{if}\ packed,\ ticket. \\ pack\ \textbf{causes}\ packed. \end{array} \right\}$$

and the causal law:

$$send(Request)\ \textbf{causes}\ Fluent\ \textbf{if}\ Fluent \in Request.$$

is part of C_{local}.

When taken together D_{john} and C_{local} define what is known as John's *local diagram*.

Definition 2.2.1 (local diagram)**.** Given an agent α, α's *local diagram*, $T_{local}(\alpha)$, is the diagram defined by the action description $D_\alpha \cup C_{local}$.

Example 2.2.1. Using our previous description of John, $D_{john} \cup C_{local}$ is as follows:

$$D_{john} \cup C_{local} = \left\{ \begin{array}{l} \textbf{caused}\ ready\ \textbf{if}\ packed,\ ticket. \\ pack\ \textbf{causes}\ packed. \\ send(Request)\ \textbf{causes}\ Fluent\ \textbf{if}\ Fluent \in Request. \end{array} \right\}$$

Which gives the following local diagram for John:

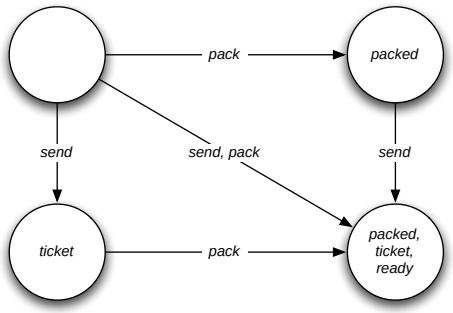

Figure 2.1: local diagram for agent John

2.2.2 *THE GLOBAL PERSPECTIVE*

As was mentioned earlier, an agent's local perspective is used by the agent to reason about his own actions. The central idea is that when an agent reasons about actions of type $send$, he operates under the assumption that issuing a request causes the effect specified by the request to materialize. Recall our characterization of John's mental model:

- ▷ John is ready when he has his tickets and is packed.
- ▷ Packing bags causes them to be packed.
- ▷ Having his secretary purchase tickets for him causes him to have tickets.

This model is sufficient if John simply wants to reason about the effects of his own actions. Suppose however that John wants to reason about how his actions may *actually play out*. It is doubtful that John's secretary is capable of instantaneously obtaining his tickets. Consequently John knows that he must wait for some unspecified period of time, depending on how many other tasks his secretary has to perform. His new model is described as follows:

- ▷ John is ready when he has his tickets and is packed.
- ▷ Packing bags causes them to be packed.
- ▷ Issuing a request causes that request to become pending.
- ▷ If his request is satisfied, its corresponding effect is satisfied.
- ▷ Until his request is satisfied John has to wait.

Notice that his models only differ in how communication actions are handled. Building off of the previous representation we represent the new knowledge as follows:

$$send(Request) \textbf{ causes } pending(Request).$$
$$pending(Request) \textbf{ triggers } wait(Request).$$
$$wait(Request) \textbf{ causes } satisfied(Request) \textbf{ or } \neg satisfied(Request).$$
$$\textbf{caused } Fluent \textbf{ if } satisfied(Request),\ Fluent \in Request.$$

As with his local perspective, John is not concerned with the actual actions that his secretary may perform to obtain his tickets. He does however need to understand that he has to wait for her to get them. This type of reasoning is termed *reasoning from a global perspective*. With the exception of the first and last statements, the above rules are not expressible in $\mathcal{AL}$. They are however easily represented in A-Prolog (and hence in CR-Prolog), and their representation comprises the module C_{global}.

When taken together D_{john} and C_{global} define what is known as John's *global diagram*.

Definition 2.2.2 (global diagram)**.** Given an agent α, α's *global diagram*, $T_{global}(\alpha)$, is the transition diagram defined by the following logic program:

$$\Pi_{global}(\alpha) = \Pi_{\alpha} \cup \Pi_{inertia} \cup \Pi_{effects} \cup \Pi_{default} \cup C_{global}.$$

Where Π_{α} is a logic program representation of the agent; $\Pi_{inertia}$ is a logic program describing inertia; $\Pi_{effects}$ is a logic program describing the general effects of actions; and $\Pi_{default}$ is a logic program characterizing the behavior of default fluents. These extra modules will be presented in greater detail in subsequent chapters.

Example 2.2.2. Using our previous description of John, a high level presentation

of $D_{john} \cup C_{global}$ is given below:

$$D_{john \cup C_{global}} = \left\{ \begin{array}{l} \textbf{caused } ready \textbf{ if } packed,\ ticket. \\ pack \textbf{ causes } packed. \\ send(Request) \textbf{ causes } pending(Request). \\ pending(Request) \textbf{ triggers } wait(Request). \\ wait(Request) \textbf{ causes } satisfied(Request) \textbf{ or } \neg satisfied(Request). \\ \textbf{caused } Fluent \textbf{ if } satisfied(Request),\ Fluent \in Request. \end{array} \right\}$$

Which describes the global diagram presented in figure 2.2.2 (note that the notation has been simplified somewhat).

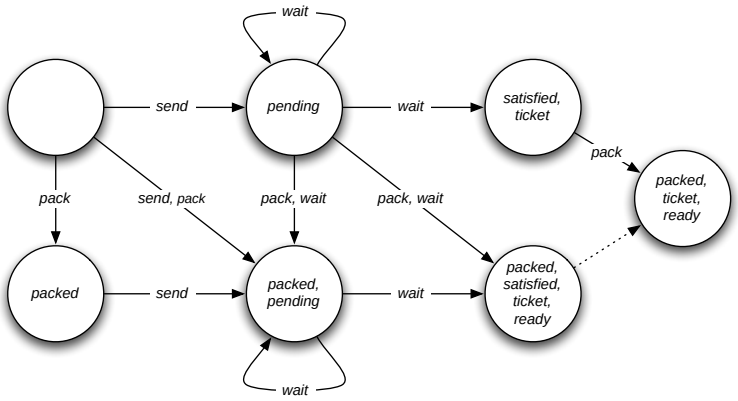

Figure 2.2: global diagram for agent John

2.2.3 *THE SYSTEM PERSPECTIVE*

Every agent of a multi-agent system has its own local and global perspectives of the domain that it uses to reason about the effects of its actions. If we take a step back however, and look at the system of agents as a whole, the global perspectives of the agents may be combined to form what is termed the *system perspective*, characterized by the *system diagram* defined as follows:

Definition 2.2.3 (system diagram of a collaborative multi-agent system)**.** Given a collaborative multi-agent system, M, M's *system diagram*, $T_{system}(M)$, is the transition diagram defined by the following logic program:

$$\Pi_{system}(M) = (\bigcup_{\alpha \in M} \Pi_\alpha) \cup \Pi_{inertia} \cup \Pi_{effects} \cup \Pi_{default} \cup C_{global}.$$

The system diagram is used to describe the behavior of a multi-agent system as a whole, showing the interactions of all the agents of the system. This diagram is used primarily in diagnosis (where it plays the same role as the actual transition diagram mentioned in [4]), and in analyzing the properties of a given multi-agent system. These topics will be revisited in subsequent chapters.

2.3 THE MULTI-AGENT LOOP

Recall that in the current research on single-agent systems, an agent's behavior is characterized by what has been termed the agent-loop [1] (also known as the observe-think-act loop). To recap, the agent loops through the following general steps:

- ▷ observe the state of the world
- ▷ if these observations fail to match up with the agent's expectations, identify the reason for the discrepancy
- ▷ select a goal
- ▷ generate a sequence of actions to achieve the goal
- ▷ execute the first element of the sequence

In our framework for multi-agent systems the structure of the loop remains unchanged. What does change is how the agent performs the steps of *planning* and *diagnosis*. This is where local and global perspectives play a vital role.

In the single-agent version of the loop, planning is reduced to finding a path from the agent's current/initial state to a goal state in the agent's transition diagram. This basic template is observed in our framework as well, except that path generated is from the agent's local diagram, rather than the diagram described by the agent's action description.

Example 2.3.1. Recall that our previous description of John, yielded the local diagram described in figure 2.2.1. Suppose that John has the goal of becoming $ready$. During the planning phase of the multi-agent loop John could generate the following trajectory, π as a possible plan:

Figure 2.3: possible plan, π, for achieving $ready$

Once the agent has generated a plan, the agent uses its global diagram to generate a sequence of actions describing how execution of its plan may play out in the presence of other agents. This sequence forms the agent's expectations concerning the results of its actions, and is captured by the notion of an *expansion of a path*.

Definition 2.3.1 (expansion of a transition)**.** Let $\tau = \langle \sigma, a, \acute{\sigma} \rangle$ be a transition in $T_{local}(\alpha)$ for some agent α. The *expansion* of τ in $T_{global}(\alpha)$, denoted by $\acute{\tau}$, is defined as follows:

- ▷ If $a \cap A_\alpha = a$, then $\acute{\tau}$ is a transition $\langle \delta, \acute{a}, \acute{\delta} \rangle \in T_{global}(\alpha)$ where $\sigma \subseteq \delta$, $\acute{\sigma} \subseteq \acute{\delta}$, and $a \subseteq \acute{a}$, where $\acute{a} \setminus a = \emptyset$, or $\acute{a} \setminus a$ only contains actions of type $wait$.
- ▷ If a contains an action of type $send$ whose corresponding message is m, $\acute{\tau}$ is a path in $T_{global}(\alpha)$ of the form:

$$\langle \delta, \acute{a}, \delta_0, a_0, \ldots, a_n, \acute{\delta} \rangle$$

where $a_0, \ldots, a_n$ are actions of type $wait$ whose corresponding message is m, and $\sigma \subseteq \delta$, $\acute{\sigma} \subseteq \acute{\delta}$, and $a \subseteq \acute{a}$.

Definition 2.3.2 (expansion of a path). Let $\pi = \langle \sigma_1, a_1, \sigma_2, a_2, \ldots, a_n, \sigma_n \rangle$ be a path in $T_{local}(\alpha)$ for some agent α. The *expansion* of π in $T_{global}(\alpha)$, is any path $\acute{\pi} = \beta \circ \gamma \in T_{global}(\alpha)$ such that:

- β is an expansion of $\langle \sigma_1, a_1, \sigma_2 \rangle$ whose final state is δ_2
- γ is an expansion of $\langle \sigma_2, a_2, \ldots, a_n, \sigma_n \rangle$ whose initial state is δ_2

Example 2.3.2. Given the possible plan from example 2.3.1, John could use the following expansion to form his expectations of how the plan might actually be realized:

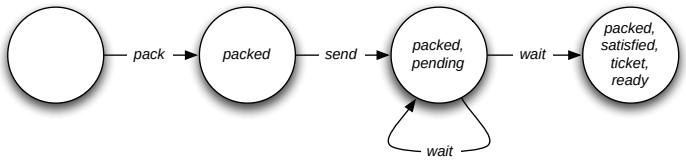

Figure 2.4: an expansion of π from example 2.3.1

The relationship between plans and their expansions will be discussed in subsequent chapters.

CHAPTER 3

REPRESENTING SYSTEMS OF AGENTS

Having established the basic framework for reasoning about multi-agent systems, we can focus our attention on the process of representing such a system in a logic programming language. As a guiding example we consider a simple multi-agent system consisting of two agents: John and his secretary. John is able to pack his bags, and have his secretary purchase tickets for him. To get ready for a trip he needs to have packed and to have obtained a ticket. John's secretary purchases tickets online by logging onto a computer and performing a transaction. We will begin our discussion by modeling John. As we are not using a formal action language in our framework, the translations presented in the subsequent sections are necessarily informal.

3.1 PURCHASING TICKETS: MODELING JOHN

From the above description we know the following about John:

1. he is able to pack his bags
2. he is able to have his secretary buy tickets for him
3. if he is both packed and has a ticket he is ready for a trip

The first item tells us that John has a single non-communication action (packing his bags), while the second states that he is able to send a message to a secretary with the associated request of purchasing a ticket. Furthermore, we know that John is aware of several fluents, namely: being packed, having a ticket, and being ready. Consequently, we describe the agent John as follows:

▷ $\mathrm{F}_{\mathrm{john}} = \{\mathit{ticket}, \mathit{packed}, \mathit{ready}\}$.

▷ $A_{john} = \{pack\}$.

▷ $R_{john} = \{buy(ticket, john) = \{ticket\}\}$.

▷ D_{john} defines the following local diagram:

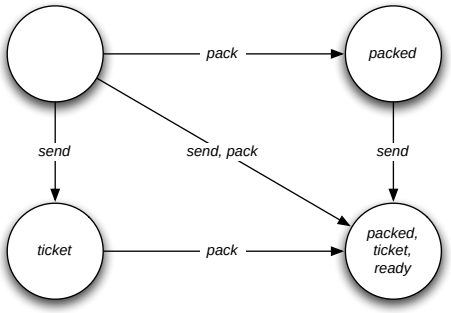

Figure 3.1: local diagram for agent $john$

In the following sections we describe in detail a representation of John's knowledge in CR-Prolog.

3.1.1 *DOMAIN SPECIFIC KNOWLEDGE*

In this section we represent the portion of John's knowledge of the world that is domain-specific, yet independent of other agents, i.e. his knowledge of fluents, and actions which do not involve message passing.

3.1.1.1 OBJECTS OF THE DOMAIN

When defining an agent that operates in a multi-agent system, we begin by introducing the various objects of the domain. John is aware of himself (as an agent of the domain), time, and tickets. In addition John is aware of the boolean values *true* and *false*. This information is encoded below:

```
#const maxTime = 3.
```

```
agent(john).
time(0..maxTime).
object(ticket).
boolean(true).
boolean(false).
#domain time(T).
#domain agent(Agent).
#domain object(Object).
#domain boolean(Value).
#domain boolean(Value1).
#domain boolean(Value2).
```

In addition, John is aware of the following properties:

```
property(packed).
property(ready).
property(possessionOf(Object)).
#domain property(Property).
```

Properties denote attributes that an agent may have. Specifically, John may have the property of being $packed$, $ready$, or having possession of an object ($possessionOf$).

3.1.1.2 FLUENTS AND ACTIONS

Having introduced the objects of the domain it is now possible to describe the fluents that John is aware of, and the actions that he may perform. Note that here we are only describing actions which do not involve communication between agents.

John is aware of only one type of fluent: $has(Agent, Property)$. Fluents of this form are used to specify that an agent has some specific property. For example,

$has(john, possessionOf(ticket))$ is read as: "John has possession of a ticket," and $has(john, packed)$ has the informal reading of "John has the property of being packed." Fluents of this form are considered to be inertial. This information is encoded in CR-Prolog below:

```
inertialFluent(has(Agent,Property)).
#domain inertialFluent(InertialFluent).
fluent(InertialFluent).
#domain fluent(Fluent).
```

Actions are defined as named records consisting of the following fields:

- ▷ $action(Name)$ - denotes the name of the action
- ▷ $type(Name, Type)$ - denotes the type of the action
- ▷ $actor(Name, Agent)$ - denotes the agent performing the action
- ▷ $effect(Name, Fluent, Value)$ - denotes the effect of the action in terms of assigning a value to a fluent.

If an action has more than one effect, then the record contains multiple entries of the form $effect(Name, Fluent, Value)$. The names of actions are parameterized by the actor. This ensures that if the names of the agents in a particular domain are unique, the names of their actions will be unique as well. As an additional note, facts of the form $h(Fluent, Time)$ are used to specify that the given fluent is true at the a particular time. Similarly, facts of the form $-h(Fluent, Time)$ state that the given fluent does not hold at the given time. As an example, the fact $h(has(john, packed), 0)$ is read as: "John is initially packed."

As John has a single action that does not involve message passing (packing his bags), only one action is specified below:

```
action(pack(john)).
type(pack(john),pack).
actor(pack(john),Agent).
effect(pack(john),has(john,packed),true).
#domain action(Action).
```

In addition, we introduce the following executability condition: "agents do not repack their bags." This rule is presented below:

```
-o(Action,T) :-
    type(Action,pack),
    actor(Action,Agent),
    h(has(Agent,packed),T).
```

The rules which govern when an action's effects take hold are discussed in subsequent sections as they are domain independent and are part of a separate module.

3.1.2 *DEPENDENT KNOWLEDGE*

Having described the portion of an agent's domain-specific knowledge which is independent of other agents in the domain, we now describe the portion of his knowledge that deals with communication. This is done by expanding his knowledge of the objects in the domain to include other agents, and by introducing two new classes of objects known as *requests* and *messages*.

3.1.2.1 OBJECTS OF THE DOMAIN

We first expand John's knowledge of the agents in the domain. This is accomplished by simply adding the following fact to John's knowledge base:

```
agent(secretary).
```

```
#domain agent(Agent1).
#domain agent(Agent2).
```

As was mentioned previously, requests are named sets of fluents that are used to define an interface for communication between agents. These correspond to the names of tasks that other agents may perform. In this particular example, John only has a single type of request that he may issue namely, $buy(Object, Agent)$, read as "buy the given object for the specified agent." Requests are declared as follows:

```
request(buy(Object,Agent)).
#domain request(Request).
mapsTo(john,buy(Object,Agent),has(Agent,possessionOf(Object)),true).
```

The relation $mapsTo$ is used to specify the set of fluents literals that the request represents. In this case, the request $buy(Object, Agent)$ in John's knowledge base is mapped onto the fluent $has(Agent, possessionOf(Object))$. The third parameter specifies the value that is to be assigned to the fluent when the request is satisfied (more on this later).

Once an agent's requests have been declared, we can define the relationships between agents in terms of these requests. Specifically we need to specify which agents can make requests, and which agents satisfy these requests. This is done by specifying the client/server relationships between agents and the various requests as shown below:

```
clientOf(john,buy(Object,Agent)).
serverOf(secretary,buy(Object,Agent)).
```

The first rule states that John is able to issue requests of the given form, while the second states that the secretary is able to satisfy such requests.

Having defined John's requests, we now turn our attention to the messages he may send. Messages, like actions, are represented as named records consisting of the following fields:

▷ $message(Name)$ - denotes the name of the message.

▷ $task(Name, Request)$ - denotes the request passed by the message

▷ $receiver(Name, Agent)$ - denotes the agent who is a recipient of the message

The names of message are parameterized by the sender. As with actions, this will ensure that if the names of agents in a particular domain are unique, the names names of their messages will be unique as well.

John's message to his secretary requesting that she purchase a ticket for him is encoded as follows:

```
message(msg1(john)).
task(msg1(john),buy(ticket,john)).
receiver(msg1(john),secretary).
#domain message(Message).
```

3.1.2.2 SPECIFYING COMMUNICATION ACTIONS

Once the basic objects of the domain have been extended by requests and messages, we now specify the agent's communication actions. As with his regular actions, such actions are represented as named records having the following fields:

▷ $action(Name)$ - denotes the name of the action

▷ $type(Name, Type)$ - denotes the type of the action

▷ $actor(Name, Agent)$ - denotes the agent performing the action

- ▷ $sent(Name, Message)$ - denotes the message sent by an occurrence of the action

- ▷ $effect(Name, Fluent, Value)$ - denotes the effect of the action in terms of assigning a value to a fluent

Unlike regular actions however, communication actions must have the type $send$, and are only partly defined in the module pertaining to an agent. The effects of such actions are specified in the logic program representations of C_{local} and C_{global}. The names of such actions are parameterized by both the actor and the message sent in order to ensure that their names are unique.

John's action of sending a message is given below:

```
action(send(john,Message)).
type(send(john,Message),send).
actor(send(john,Message),john).
sent(send(john,Message),Message).
#domain action(Action).
```

3.2 PURCHASING TICKETS: MODELING THE SECRETARY

We continue our discussion by modeling the secretary. Let us begin by recalling the scenario: John is able to pack his bags, and have his secretary purchase tickets for him. To get ready for a trip he needs to have packed and to have obtained a ticket. John's secretary purchases tickets online by logging onto a computer and performing a transaction. Given this description, we know the following about John's secretary:

- ▷ she is able to login to a computer

- ▷ once she is logged on to a computer she is able to purchase tickets

▷ she purchases tickets by completing a transaction

From this description we can see that all of the secretary's actions do not involve communication. In addition, we know that she is capable of satisfying John's request to purchase a ticket. As a result, we know that she is aware of several fluents, namely: being online (logged on), and having successfully purchased the tickets. Consequently, we describe the secretary as follows:

▷ $F_{secretary} = \{online, ticket\}$.

▷ $A_{secretary} = \{login, buy\}$.

▷ $R_{secretary} = \{buy(ticket, john) = \{ticket\}\}$.

▷ $D_{secretary}$ defines the following local diagram:

Figure 3.2: local diagram for agent $secretary$

In the following sections we describe in detail a representation of the secretary's knowledge in CR-Prolog.

3.2.1 *DOMAIN SPECIFIC KNOWLEDGE*

In this section we represent the portion of the secretary's knowledge of the world that is domain-specific, yet independent of other agents, i.e. her knowledge of fluents, and actions which do not involve message passing.

3.2.1.1 OBJECTS OF THE DOMAIN

As with our representation of John, we being by introducing the various objects of the domain. The secretary is aware of herself (as an agent of the domain), time, and tickets. In addition she is aware of the boolean values *true* and *false*. This information is encoded below:

```
#const maxTime = 3.
agent(john).
time(0..maxTime).
object(ticket).
boolean(true).
boolean(false).
#domain time(T).
#domain agent(Agent).
#domain object(Object).
#domain boolean(Value).
#domain boolean(Value1).
#domain boolean(Value2).
```

In addition, the secretary is aware of the following properties:

```
property(online).
property(received(Object)).
#domain property(Property).
```

As was mentioned previously, properties denote attributes that an agent may have. Specifically, the secretary may have the property of being $online$, or having received of an object ($received$). Ideally we would use different variables when describing John's secretary, however due to programmatic necessity we use the same variables

in order to combine the modules together smoothly. When more modular answer set solver appear however this will no longer be necessary.

3.2.1.2 FLUENTS AND ACTIONS

Having introduced the objects of the domain it is now possible to describe the fluents that the secretary is aware of, and the actions that she may perform. Note that here we are only describing actions which do not involve communication between agents.

John's secretary is aware of only one type of fluent, namely $has(Agent, Property)$. As with John, fluents of this form are used to specify that an agent has some specific property. For example, $has(secretary, online)$ is read as: "the secretary is online," and $has(secretary, recieved(ticket))$ has the informal reading of "the secretary has received a ticket." Fluents of this form are considered to be inertial and are encoded in CR-Prolog below:

```
inertialFluent(has(Agent,Property)).
#domain inertialFluent(InertialFluent).
fluent(InertialFluent).
#domain fluent(Fluent).
```

Notice that because the secretary is aware of a different set of properties from John, the restriction that the agents of our system have disjoint sets of fluents is preserved.

When representing the secretary's actions we again use named records as the basis of our representation. John's secretary is specified as having two types of actions, that of logging onto a computer, and purchasing an object for some agent online. These are represented as follows:

```
action(login(secretary)).
```

```
type(loginlogin(secretary),login).
actor(login(secretary),secretary).
effect(login(secretary),has(secretary,online),true).

action(buy(secretary,Object,Agent)).
type(buy(secretary,Object,Agent),buy).
actor(buy(secretary,Object,Agent),secretary).
effect(buy(secretary,Object,Agent),has(Agent,received(Object)),true).
#domain action(Action).
```

Along with these actions come a pair of executability conditions restriction their execution:

```
-o(Action,T) :-
    type(Action,login),
    actor(Action,Agent),
    -h(has(Agent,online),T).

-o(Action,T) :-
    type(Action,buy),
    actor(Action,Agent),
    -h(has(Agent,online),T).
```

The first rule states that an agent may not login to a computer if that agent is already online, and the second specifies that an agent may not purchase an object if that agent is not online.

3.2.2 *DEPENDENT KNOWLEDGE*

Having described the portion of the secretary's domain-specific knowledge which is independent of other agents in the domain, we now describe the portion of her knowledge that deals with communication. As before, this is accomplished by expanding her knowledge of the objects in the domain to include other agents, and by introducing any and all requests and messages.

3.2.2.1 OBJECTS OF THE DOMAIN

We first expand the secretary's knowledge of the agents in the domain. This is accomplished by simply adding the following fact (and associated variable declarations) to her knowledge base:

```
agent(john).
#domain agent(Agent1).
#domain agent(Agent2).
```

As was mentioned previously, requests are named sets of fluents that are used to define an interface for communication between agents. John's secretary is aware of a single type of request, $\mathrm{buy}(\mathrm{Object}, \mathrm{Agent})$, for which she happens to be a server. As with John, the same general representation is used to describe her knowledge of requests:

```
request(buy(Object,Agent)).
#domain request(Request).
clientOf(john,buy(Object,Agent)).
serverOf(secretary,buy(Object,Agent)).
mapsTo(secretary,buy(Object,Agent),has(Agent,received(Object)),true).
```

Notice that the secretary maps the request $\mathrm{buy}(\mathrm{Object}, \mathrm{Agent})$ onto the fluent $\mathrm{has}(\mathrm{Agent}, \mathrm{received}(\mathrm{Object}))$ whereas John maps the the request onto the fluent

$has(Agent, possessionOf(Object))$. This is by design, as each agent in the domain is intended to have his own view of the world. Through this clean partitioning of the world-views of agents we can develop the agents in isolation, and simply combine them to form a multi-agent domain.

The secretary herself has no communication actions, and consequently no messages, and no actions of the type $send$ need to be specified. For convenience however, the following message is added from the logic program representing John, to enable the secretary to be run independently of the full representation of John:

```
message(msg1(john)).
task(msg1(john),buy(ticket,john)).
receiver(msg1(john),secretary).
#domain message(Message).
```

The remaining sections will present the representations of C_{local} and C_{global}, as well as other domain independent modules, namely those dealing with default fluents and inertia.

3.3 GENERAL KNOWLEDGE

Now that we have represented John's domain-specific knowledge, we present those portions of his knowledge which are domain-independent. This knowledge is presented in the form of various logic programs, which are to be thought of as part of a general library of knowledge. These modules include both the local and global communication modules, as well as modules describing inertia, the effects of actions, and the behaviors of default fluents, etc.

3.3.1 *THE COMMUNICATION MODULE:* C_{local}

Recall that the *local communication module*, C_{local}, is used to define an agent's local diagram. As was mentioned previously, when an agent reasons in his local perspective, his model of the effects of message passing actions is governed by the following dynamic causal law:

$$send(\text{Request}) \textbf{ causes } \text{Fluent} \textbf{ if } \text{Fluent} \in \text{Request}.$$

The effect of such actions in an agent's local perspective as described by the causal law above is represented in CR-Prolog by the following rule:

```
effect(Action,Fluent,Value) :-
    type(Action,send),
    actor(Action,Agent),
    sent(Action,Message),
    task(Message,Request),
    mapsTo(Agent,Request,Fluent,Value).
```

In addition, we introduce a pair of executability conditions governing when such actions may be executed. Taken together, these rules may be read as: "an agent does not issue a request, if the desired effect is already in place."

```
-o(Action,T) :-
    type(Action,send),
    h(Fluent,T),
    effect(Action,Fluent,true).


-o(Action,T) :-
    type(Action,send),
```

```
    -h(Fluent,T),
    effect(Action,Fluent,false).
```

These conditions are added to prevent an agent from making redundant requests, such as John asking his secretary to purchase tickets if he already has them.

3.3.2 *THE COMMUNICATION MODULE:* C_{global}

Having described C_{local}, we now turn our attention to the *global communication module*, C_{global} which defines an agent's global diagram. The causal statements presented in this module are defined in terms of the fluents $pending$, and $satisfied$, whose behaviors vary depending on whether the agent in question is a client or a server of a particular request. We begin our representation by introducing these fluents:

```
defaultFluent
(
    pending(perspective(Client),Client,Server,Request);
    satisfied(perspective(Client),Client,Server,Request)
) :-
    clientOf(Client,Request),
    serverOf(Server,Request).
#domain defaultFluent(DefaultFluent).
defaultsTo(DefaultFluent,false).
fluent(DefaultFluent).
```

```
inertialFluent
(
    pending(perspective(Server),Client,Server,Request);
    satisfied(perspective(Server),Client,Server,Request)
) :-
    clientOf(Client,Request),
    serverOf(Server,Request).
```

The use of the semi-colon in the above rules is a convenient notation for specifying a list of elements. The first rule specifies that both $pending$ and $satisfied$ are considered to be default fluents when the body of the rule is satisfied. Similarly, the second specifies that both $pending$ and $satisfied$ are considered to be inertial when the body of the rule is specified. Notice that from the perspective of a client these fluents are *default fluents*, whose value defaults to $false$. This stems from our intuition that such fluents are only "active" (have a value of $true$) from the point at which an agent issues the request, until the request becomes fulfilled. Another way to view this is that if no requests are ever issued, they may never be pending, and similarly never become satisfied. However, once a client's request becomes pending it remains so until the request becomes satisfied, at which point it returns to it's default value. The client himself however does not affect the value of these fluents. Such fluents are only changed indirectly through the actions of some other agent in the domain. From the perspective of a server however, these fluents are considered to be inertial, due to the fact that the actions of the server affect these fluents directly. It is also worth noting that the fluents have been expanded by additional parameters. A fluent literal of the form $pending(perspective(Client), Client, Server, Request)$ has the informal reading of "from the perspective of the client, the client's request to the given server is

pending." Similarly for the fluent *satisfied*.

The behavior of these fluents is captured in part by a collection of state constraints. The first pair of state constraints governs the behavior of these fluents from the perspective of a client agent. Taken together, they represent the idea that if a request is pending, it remains so until it becomes satisfied.

```
h(pending(perspective(Client),Client,Server,Request),T + 1) :-
    -h(satisfied(perspective(Client),Client,Server,Request),T + 1),
    h(pending(perspective(Client),Client,Server,Request),T).


-h(pending(perspective(Client),Client,Server,Request),T) :-
    h(satisfied(perspective(Client),Client,Server,Request), T).
```

The next pair of rules rule links together the world-views of the client and server by stating that if a request is pending from the client's perspective, it is also pending in the world-view of its respective server. Similarly, if a server views a request as satisfied, then the client views the request as being satisfied as well.

```
h(pending(perspective(Server),Client,Server,Request),T) :-
    h(pending(perspective(Client),Client,Server,Request),T).


h(satisfied(perspective(Client),Client,Server,Request),T) :-
    h(satisfied(perspective(Server),Client,Server,Request),T).
```

The next state constraint specifies that if a server has satisfied a request, that request is no longer pending:

```
-h(pending(perspective(Server),Client,Server,Request),T) :-
    h(satisfied(perspective(Server),Client,Server,Request),T).
```

The next grouping of rules define when a request is viewed as satisfied by a server. A request is said to be satisfied if it was previously pending, and the fluent that the request specifies is satisfied. In addition, while a particular request is pending, it is not considered to be satisfied.

```
h(satisfied(perspective(Server),Client,Server,Request),T + 1) :-
    h(pending(perspective(Server),Client,Server,Request),T),
    h(Fluent,T + 1),
    mapsTo(Server,Request,Fluent,true).


h(satisfied(perspective(Server),Client,Server,Request),T + 1) :-
    h(pending(perspective(Server),Client,Server,Request),T),
    -h(Fluent,T + 1),
    mapsTo(Server,Request,Fluent,false).


-h(satisfied(perspective(Server),Client,Server,Request),T) :-
    h(pending(perspective(Server),Client,Server,Request),T).
```

Now that we have discussed the behaviors of the fluents $pending$ and $satisfied$, we can now define how client agents reason about communication actions. When reasoning from his global perspective, a client's model of the effects of message passing actions is governed by the following causal statements:

$$send(\text{Request}) \textbf{ causes } pending(\text{Request}).$$
$$pending(\text{Request}) \textbf{ triggers } wait(\text{Request}).$$
$$wait(\text{Request}) \textbf{ causes } satisfied(\text{Request}) \textbf{ or } \neg satisfied(\text{Request}).$$
$$\textbf{caused } \text{Fluent} \textbf{ if } satisfied(\text{Request}),\ \text{Fluent} \in \text{Request}.$$

Due to our choice of using named records as the basis of our representation of an agent's action, the causal statement:

$$send(Request) \textbf{ causes } pending(Request).$$

is represented as follows:

```
effect(Action,pending(perspective(Client),
    Client,Server,Request),true) :-
        type(Action,send),
        actor(Action,Client),
        sent(Action,Message),
        receiver(Message,Server),
        task(Message,Request).
```

Before we represent the second statement, we must first introduce actions of type $wait$. Such actions are again represented as named records consisting of the following fields:

- ▷ $action(Name)$ - denotes the name of the action
- ▷ $type(Name, Type)$ - denotes the type of the action
- ▷ $actor(Name, Agent)$ - denotes the agent performing the action
- ▷ $msg(Name, Message)$ - denotes the message for whose task the agent is waiting to become satisfied

The names of such actions are parameterized by both the actor, and the message in question. As before, this is to ensure that the names of such actions are unique provided that the names of agents and messages in the given domain are unique as well. Such actions are represented in a domain independent fashion as follows:

```
action(wait(Agent,Message)) :-
    task(Message,Request),
    clientOf(Agent,Request).
type(wait(Agent,Message),wait).
actor(wait(Agent,Message),Agent).
msg(wait(Agent,Message),Message).
```

In addition, we introduce the following executability conditions to help govern a clients reasoning regarding the applicability of actions of type *send* and *wait*:

```
-o(Action,T) :-
    h(pending(perspective(Client),Client,Server,Request),T),
    type(Action,send),
    actor(Action,Client).


-o(Action,T) :-
    h(Fluent,T),
    type(Action,send),
    actor(Action,Client),
    sent(Action,Message),
    task(Message,Request),
    mapsTo(Agent,Request,Fluent,true).


-o(Action,T) :-
    -h(Fluent,T)
    type(Action,send),
    actor(Action,Client),
    sent(Action,Message),
```

```
        task(Message,Request),
        mapsTo(Agent,Request,Fluent,false).

    -o(Action,T) :-
        -h(pending(perspective(Client),Client,Server,Request),T),
        type(Action,wait),
        actor(Action,Client),
        msg(Action,Message),
        receiver(Message,Server),
        task(Message,Request).
```

Taken together these constraints state that an agent does not issue any requests if the desired effect of the request is already satisfied, and that an agent does not wait for a request to become satisfied unless that request is currently pending.

Having introduced the required actions, we can now represent the following statements:

$$wait(\mathrm{Request})\ \mathbf{causes}\ satisfied(\mathrm{Request})\ \mathbf{or}\ \neg satisfied(\mathrm{Request}).$$
$$pending(\mathrm{Request})\ \mathbf{triggers}\ wait(\mathrm{Request}).$$

as follows (note that in the extended language of *LParse*, | takes the place of or):

```
    h(satisfied(perspective(Client),Client,Server,Request),T + 1) |
    -h(satisfied(perspective(Client),Client,Server,Request),T + 1) :-
        o(Action,T),
        type(Action,wait),
        actor(Action,Client),
        msg(Action,Message),
        receiver(Message,Server),
```

```
        task(Message,Request).


    o(Action,T) :-
        h(pending(perspective(Client),Client,Server,Request),T),
        type(Action,wait),
        actor(Action,Client),
        msg(Action,Message),
        receiver(Message,Server),
        task(Message,Request).
```

The final statement:

$$\textbf{caused } \mathit{Fluent} \textbf{ if } \mathit{satisfied}(\mathit{Request}),\ \mathit{Fluent} \in \mathit{Request}.$$

Is represented as by the following pair of rules:

```
    h(Fluent,T) :-
        h(satisfied(perspective(Client),Client,Server,Request),T),
        mapsTo(Client,Request,Fluent,true).


    -h(Fluent,T) :-
        h(satisfied(perspective(Client),Client,Server,Request),T),
        mapsTo(Client,Request,Fluent,false).
```

3.3.3 *EFFECTS OF ACTIONS*

Recall that actions were represented as named records consisting of a number of fields. Using such a representation for actions allows us to encode the rules which apply their effects in a general manner. Taken together, the following two rules may be read as: "if the effect of an action is to assign some value to a fluent, then an

occurrence of that action causes the fluent to have that value." As was mentioned earlier, this module is denoted by $\Pi_{effects}$.

```
h(Fluent,T + 1) :-
    o(Action,T),
    effect(Action,Fluent,true).

-h(Fluent,T + 1) :-
    o(Action,T),
    effect(Action,Fluent,false).
```

3.3.4 *THE INERTIA AXIOMS*

As was mentioned previously in the introduction, the inertia axioms present an elegant solution to the frame problem. Taken together, the following two rules may be read as: "the value of an inertial fluent remains unchanged unless we have reason to believe otherwise." From here on this module will be referred to as $\Pi_{inertia}$.

```
h(InertialFluent,T + 1) :-
    h(InertialFluent,T),
    not -h(InertialFluent,T + 1).

-h(InertialFluent,T + 1) :-
    -h(InertialFluent,T),
    not h(InertialFluent,T + 1).
```

3.3.5 *DEFAULT FLUENTS*

Finally we need to specify the behaviors of default fluents. The behavior of such fluents is given by a pair of rules which when taken together are read as: "if a fluent

defaults to some value, then it has that value unless otherwise specified." As was alluded to previously, this module is denoted by $\Pi_{default}$.

```
h(DefaultFluent,T) :-
    defaultsTo(DefaultFluent,true),
    not -h(DefaultFluent,T).


-h(DefaultFluent,T) :-
    defaultsTo(DefaultFluent,false),
    not h(DefaultFluent,T).
```

Now that we have seen how to describe multi-agent systems in CR-Prolog, we demonstrate how these various modules may be combined to define the various diagrams present in the system.

Example 3.3.1. Suppose that we know that John's bags are initially unpacked, that he does not have a ticket, and that he sets about packing his bags. We can determine the possible successor states in John's local diagram as follows:

1. represent what is known about his current state as a logic program $\Pi_{scenario}$
2. compute the answer sets of $\Pi_{\cup}C_{local} \cup \Pi_{effects} \cup \Pi_{inertia} \cup \Pi_{default} \cup \Pi_{scenario}$

The first step is accomplished by the following CR-Prolog representation of the scenario (the last two statements simply trim the output of the answer set solver to make it readable):

```
-h(has(john,packed),0).
-h(has(john, possessionOf(ticket)), 0).
o(pack(john),0).
#hide.
```

```
#show h(X,Y).
#show o(X,Y).
```

The second step is accomplished by using the CRModels, and MKatoms systems [3] as shown by the following command (note that the names of the modules would be replaced by the names of the corresponding files):

$$\texttt{crmodels -m 0}\ \ \Pi,\ C_{local},\ \Pi_{effects},\ \Pi_{inertia},\ \Pi_{default},\ \Pi_{scenario}$$

whose single answer set is given below:

```
h(has(john,packed),1)
-h(has(john,ready),1)
-h(has(john,possessionOf(ticket)),1)
o(pack(john),0)
```

and corresponds to the following transition in John's local diagram:

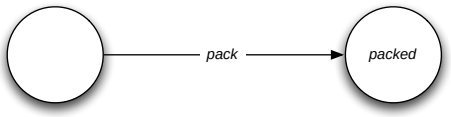

Figure 3.3: path π from $T_{local}(John)$

CHAPTER 4

PROPERTIES OF THE FRAMEWORK

The aforementioned logic programs are used in part to define an agent's local and global diagrams. In this chapter we discuss the properties of multi-agent systems that are reflected in various characteristics of these diagrams.

4.1 FEASIBILITY OF PLANS

As was mentioned previously, once an agent selects a goal he then uses his local perspective to determine a plan which will achieve it. Later he uses his global perspective to determine how his plan may play out in the domain by determining its possible expansions. Is this always possible however? Do paths in an agent's local diagram always have expansions in the global diagram? In this section we show that this is the case, that paths of an agent's local diagram are guaranteed to have at least one expansion in the agent's global diagram.

We begin by showing that states of an agent's local diagram may be extended to obtain states in the agent's global diagram. The logic programs referenced in the proofs that follow may be found in the appendix.

Lemma 4.1.1. Let α be an agent, and let σ be a state of $T_{local}(\alpha)$. Then there exists a state $\delta \in T_{global}(\alpha)$ such that $\sigma \subseteq \delta$.

Proof of Lemma 4.1.1: Let α be an agent, and let σ be a state of $T_{local}(\alpha)$. Let us show that $\delta = \sigma \cup \{\neg pending, \neg satisfied\}$ is a state of $T_{global}(\alpha)$. By definition of a global diagram [definition 2.2.2], we need to show that:

- $\Pi = \Pi_{global}(\alpha) \cup H(\delta, 0)$ has an answer set X.
- $\delta = \{f : h(f, 0) \in X\} \cup \{\neg f : \neg h(f, 0) \in X\}$.

Before we begin, let $H(s,i)$ denote $\{h(f,i) : f \in s\} \cup \{\neg h(f,i) : \neg f \in s\}$, and let $X = Statics(\Pi_{global}(\alpha)) \cup H(\delta,0) \cup H(\delta,1) \cup O_{base} \cup O_{comm}$ where:

- $Statics(\Pi_{global}(\alpha))$ is the set of all static rules of $\Pi_{global}(\alpha)$.
- $O_{base} = \{\neg o(A,T)\}$ such that $\exists$ a rule $r \in \Pi_\alpha$ for which $\neg o(A,T) = head(r)$ and $body(r) \subseteq Statics(\Pi_{global}(\alpha)) \cup H(\delta,0) \cup H(\delta,1)$.
- $O_{comm} = \{\neg o(A,T)\}$ such that $\exists$ a rule $r \in C_{global}$ for which $\neg o(A,T) = head(r)$ and $body(r) \subseteq Statics(\Pi_{global}(\alpha)) \cup H(\delta,0) \cup H(\delta,1)$.

First, let us show that X is an answer set of Π. Note that $\Pi^X = {\Pi_\alpha}^X \cup H(\delta,0) \cup H(\delta,1) \cup {\Pi_{inertia}}^X \cup {\Pi_{default}}^X \cup R_{unsatisfied}$ where:

- ${\Pi_{inertia}}^X$ has the form: $\{h(IF,1) \leftarrow h(IF,0)\} \cup \{\neg h(IF,1) \leftarrow \neg h(IF,0)\}$ where IF is an inertial fluent in X.
- ${\Pi_{default}}^X = \{\neg h(DF,T) \leftarrow defaultsTo(DF,false)\}$ if DF is a default fluent such that $h(DF,0) \notin X$.
- $R_{unsatisfied}$ is the remaining set of rules whose bodies are not satisfied by X.

To show that X is closed under the rules of Π^X, let us consider an answer set Y of $\Phi = \Pi_{local}(\alpha) \cup H(\sigma,0)$. From the definition of a local diagram [definition 2.2.1] and results in [15], we know that such a set Y exists, and that Y is equal to $Statics(\Pi_{local}(\alpha)) \cup H(\sigma,0) \cup H(\sigma,1) \cup O_{base}$. It is easy to see that:

1. $Statics(\Pi_{global}(\alpha)) = Statics(\Pi_{local}(\alpha)) \setminus$
   $\{effect(Action, Fluent, Value) : type(Action, send)\} \cup$
   $\{effect(Action, pending, true) : type(Action, send)\}$,
2. and that $X = Y \setminus \{effect(Action, Fluent, Value) : type(Action, send)\} \cup$
   $\{effect(Action, pending, true) : type(Action, send)\} \cup$

$\{\neg h(pending, 0), \neg h(satisfied, 0)\} \cup \{\neg h(pending, 1), \neg h(satisfied, 1)\} \cup O_{comm}$.

By the definition of an answer set, Y is closed under ${\Pi_\alpha}^Y \cup H(\sigma, 0)$. From 1 and 2, X is closed under ${\Pi_\alpha}^X \cup H(\delta, 0)$. Now consider any rule of the form:

$$h(IF, 1) \leftarrow h(IF, 0) \text{ where IF is an inertial fluent.}$$

By construction of X, if $h(IF, 0) \in X$, then $h(IF, 1) \in X$. Similarly for rules of the form $\neg h(IF, 1) \leftarrow \neg h(IF, 0)$. Now consider any rule of the form:

$$\neg h(DF, T) \leftarrow defaultsTo(DF, false) \text{ such that } h(DF, 0) \notin X.$$

As $\neg pending, \neg satisfied \in \delta$, we know that $\neg h(pending, T), \neg h(satisfied, T) \in X$ for $T \in \{0, 1\}$. Consequently, X is closed under such rules as well. Lastly, X is vacuously closed under the rules of $R_{unsatisfied}$.

The minimality of X and that $\delta = \{f : h(f, 0) \in X\} \cup \{\neg f : \neg h(f, 0) \in X\}$ are evident by construction. □

Now that we have established that states in an agent's local diagram may be extended to obtain states in the agent's global diagram, we show a similar pair of results with regards to the arcs in the agent's respective diagrams.

Lemma 4.1.2. Let α be an agent, ϵ be a single elementary action such that $type(\epsilon) \neq send$, and let $\tau = \langle \sigma, \epsilon, \acute{\sigma} \rangle$ be a transition in $T_{local}(\alpha)$. Then for any state $\delta \in T_{global}(\alpha)$ such that $\sigma \subseteq \delta$, there exists an expansion $\acute{\tau} = \langle \delta, \acute{\epsilon}, \acute{\delta} \rangle \in T_{global}(\alpha)$ of τ.

Proof of Lemma 4.1.2: For simplicity we assume that α belongs to a multi-agent system M which contains a single request R. In addition, let $H(s, i)$ denote $\{h(f, i) : f \in s\} \cup \{\neg h(f, i) : \neg f \in s\}$, and let $O(s, i)$ denote $\{o(a, i) : a \in s\} \cup \{\neg o(a, i) : a \notin s\}$. Furthermore, in the interest of readability, as there is only a single request the

parameters of the fluents $pending$, and $satisfied$ are omitted. When there is a need to differentiate between the client and server variants of these fluents, the notation $pending_c$, $satisfied_c$, and $pending_s$, $satisfied_s$ is used.

There are several cases which we must consider:

- $\delta = \sigma \cup \{pending, \neg satisfied\}$.
- $\delta = \sigma \cup \{\neg pending, satisfied\}$.
- $\delta = \sigma \cup \{\neg pending, \neg satisfied\}$.
- $\delta = \sigma \cup \{pending, satisfied\}$.

Case 1: Consider the case where $\delta = \sigma \cup \{pending, \neg satisfied\}$. Let $\acute{\tau} = \langle \delta, \acute{\epsilon}, \acute{\delta} \rangle$ where $\acute{\epsilon} = \{\epsilon, w\}$ where w is an action of type $wait$ associated with R, and $\acute{\delta} = \acute{\sigma} \cup \{pending, \neg satisfied\}$. By definition of an expansion [definitions 2.3.1, 2.3.2], $\acute{\tau}$ is an expansion of τ if $\acute{\tau} \in T_{global}(\alpha)$. By definition of a global diagram [definition 2.2.2], $\acute{\tau} \in T_{global}(\alpha)$ if there exists an answer set X of $\Pi = \Pi_{global}(\alpha) \cup H(\delta, 0) \cup \{o(\epsilon, 0)\} \cup \{o(w, 0)\}$.

Let $X = Statics(\Pi_{global}(\alpha)) \cup H(\delta, 0) \cup O(\{\epsilon, w\}, 0) \cup H(\acute{\delta}, 1) \cup O(\{w\}, 1)$. Let us show that X is an answer set of Π. Note that $\Pi^X = {\Pi_\alpha}^X \cup H(\delta, 0) \cup O(\{\epsilon, w\}, 0) \cup {\Pi_{inertia}}^X \cup {\Pi_{effects}}^X \cup {\Pi_{default}}^X \cup {C_{global}}^X$ where:

- ${\Pi_{inertia}}^X$ has the form: $\{h(IF, 1) \leftarrow h(IF, 0) : \neg h(IF, 1) \notin X\} \cup \{\neg h(IF, 1) \leftarrow \neg h(IF, 0) : h(IF, 1) \notin X\}$ where IF is an inertial fluent in X.
- ${\Pi_{effects}}^X$ has the form: $\{h(F, 1) \leftarrow o(A, 0), effect(A, F, true)\} \cup \{\neg h(F, 1) \leftarrow o(\epsilon, 0), effect(A, F, false)\}$, where F is a fluent in X.
- ${\Pi_{default}}^X = \{\neg h(DF, T) \leftarrow defaultsTo(DF, false)\}$ if DF is a default fluent such that $h(DF, 0) \notin X$.

▷ ${C_{global}}^X = \{GM1 - GM22\} \cup \{\leftarrow satisfied\}$

To show that X is closed under the rules of Π^X, let us consider an answer set Y of $\Phi = \Pi_{local}(\alpha) \cup H(\sigma, 0) \cup O(\{\epsilon\}, 0) \cup H(\acute{\sigma}, 1)$. From the definition of a local diagram and results in [15], we know that such a set Y exists, and that Y is equal to $Statics(\Pi_{local}(\alpha)) \cup H(\sigma, 0) \cup O(\{\epsilon\}, 0) \cup H(\acute{\sigma}, 1)$. It is easy to see that:

1. $Statics(\Pi_{global}(\alpha)) = Statics(\Pi_{local}(\alpha)) \setminus$
   $\{effect(Action, Fluent, Value) : type(Action, send)\} \cup Statics(C_{global})$

2. $X = Y \setminus \{effect(Action, Fluent, Value) : type(Action, send)\} \cup$
   $Statics(C_{global}) \cup \{h(pending, 0), \neg h(satisfied, 0)\} \cup$
   $\{o(w, 0)\} \cup \{h(pending, 1), \neg h(satisfied, 1)\} \cup \{o(w, 1)\}$

By the definition of an answer set, Y is closed under ${\Pi_{local}(\alpha)}^Y \cup H(\sigma, 0) \cup O(\{\epsilon\}, 0)$. From 1 and 2, X is closed under ${\Pi_\alpha}^X \cup H(\delta, 0) \cup O(\{\epsilon, w\}, 0)$. Consider any rule in ${\Pi_{inertia}}^X$ the form:

$$h(IF, 1) \leftarrow h(IF, 0) \text{ where IF is an inertial fluent.}$$

Such a rule belongs to ${\Pi_{inertia}}^X$ only if $\neg h(IF, 1) \notin X$. By construction, if $h(IF, 0) \in X$ and $\neg h(IF, 1) \notin X$, then $h(IF, 1) \in X$. Similarly for any rule in ${\Pi_{inertia}}^X$ the form:

$$\neg h(IF, 1) \leftarrow \neg h(IF, 0) \text{ where IF is an inertial fluent.}$$

Now let us consider any rule in ${\Pi_{effects}}^X$ of the form:

$$h(F, 1) \leftarrow o(A, 0), effect(A, F, true)$$

By construction, if $effect(\epsilon, F, true) \in X$, then $h(F, 1)$ must belong to X. Similarly for rules of ${\Pi_{effects}}^X$ of the form:

$$\neg h(F, 1) \leftarrow o(A, 0), effect(A, F, false)$$

Consequently, X is closed under ${\Pi_{effects}}^X$ as well. Furthermore, as $\neg satisfied \in \delta$ and $\neg satisfied \in \acute{\delta}$, X is closed under ${\Pi_{default}}^X$. Lastly, as for $T \in \{0,1\}$, $h(pending,T)$, $\neg h(satisfied,T)$, and $o(w,T)$ belong to X, it is evident that X is closed under ${C_{global}}^X$.

Having proved that X is closed under the rules of Π, we must now show that X is minimal. Suppose that there exists an answer set S of Π such that $S \subseteq X$. From the definition of an answer set [definition 1.3.2], we know that S must contain $Statics(\Pi_{global}(\alpha)) \cup H(\delta,0) \cup O(\{e,w\},0)$. As S contains $H(\delta,0)$, we know that S contains $h(pending,0)$ and $\neg h(satisfied,0)$. Consequently, from rule GM7, the fact that $S \subseteq X$, and that $h(satisfied,1) \notin X$, we know that $\neg h(satisfied,1) \in S$. As both $h(pending,0)$ and $\neg h(satisfied,1)$ belong to S, by application of rule GM13, S must also contain $h(pending,1)$. Finally by application of rule GM8, $o(w,1) \in S$.

Now let us consider any literal f where $f \in \acute{\sigma}$. By construction of X, $h(f,1) \in X$. By construction of δ, and the fact that $H(\delta,0) \subset S$, we know that $\{h(f,0) : f \in \sigma\} \subset S$. As $h(f,1) \in X$, we know that there exists a rule $r \in \Pi$ whose body is satisfied by X, and $head(r) = h(f,1)$. Suppose that r is a ground instance of IA1 (an inertia axiom). We know that $\neg h(f,1) \notin X$. As $S \subseteq X$, $\neg h(f,1) \notin S$. Consequently, $body(r)$ is satisfied, and hence $h(f,1) \in S$. As we have just shown, $\neg h(satisfied,1) \in S$. Consequently, the bodies of rules GM15 and GM16, are not satisfied, and therefore the only remaining rule r, where $head(r) = h(f,1)$ is rule AE1. As $h(f,1) \in X$, $body(r) \subset X$, and as we have previously shown, $body(r) \in S$, $h(f,1) \in S$ as well. Hence, $H(\acute{\delta},1) \in S$, $S = X$, and hence X must be minimal.

The proofs for the remaining cases are conducted in similar fashion. □

Lemma 4.1.3. Let α be an agent, a be a single elementary action such that $type(a) = send$, and let $\tau = \langle \sigma, a, \acute{\sigma} \rangle$ be a transition in $T_{local}(\alpha)$. There exists

an expansion $\acute{\tau} = \langle \delta, a, \delta_0, a_0, \ldots, a_n, \acute{\delta} \rangle \in T_{global}(\alpha)$ of τ, where $a_0, \ldots, a_n$ are actions of type $wait$ whose message corresponds to that of a, and $\sigma \subseteq \delta$, and $\acute{\sigma} \subseteq \acute{\delta}$.

Proof of Lemma 4.1.3: For simplicity we maintain the assumptions that were used in the previous proofs.

Let $\delta = \sigma \cup \{\neg pending, \neg satisfied\}$ and let $\acute{\tau} = \langle \delta, a, \delta_0, a_0, \acute{\delta} \rangle$ where $\delta_0 = \sigma \cup \{pending, \neg satisfied\}$, a_0 is the corresponding action of type $wait$, and $\acute{\delta} = \sigma \cup \{\neg pending, satisfied\}$. By construction of $T_{global}(\alpha)$ it is apparent that $\acute{\tau} \in T_{global}(\alpha)$ and hence $\acute{\tau}$ is an expansion of τ. □

Lastly, the relationship between the paths of an agent's local and global diagrams is given by the following theorem.

Theorem 4.1.1. Let α be an agent, and let π be a path whose arcs are labeled by elementary actions in $T_{local}(\alpha)$. Then there exists an expansion $\acute{\pi}$ of π in $T_{global}(\alpha)$.

Proof of Theorem 4.1.1: As before, we maintain the assumptions from the previous proofs. The proof is done by induction on the length of a path, denoted by n.

Let $n = 0$. In this case π consists of only a single state of $T_{local}(\alpha)$. From Lemma 4.1.1 we know that there exists a state $\delta \in T_{global}(\alpha)$ such that $\sigma \subseteq \delta$. Therefore, by definition π has an expansion $\acute{\pi} = \delta$ in $T_{global}(\alpha)$.

Let $n = 1$. In this case π has the form $\langle \sigma, a, \acute{\sigma} \rangle$. Here there are two possibilities to consider: $type(a) \neq send$, and $type(a) = send$. Suppose that $type(a) \neq send$. By Lemma 4.1.2, we know that there exists an expansion of π in $T_{global}(\alpha)$. Similarly, if $type(a) = send$, by Lemma 4.1.3, we know that there exists an expansion of π in $T_{global}(\alpha)$. Hence we have proven our base.

Let us assume that for any path π of length $k \leq n$ in $T_{local}(\alpha)$, π has an expansion in $T_{global}(\alpha)$.

Now let $k = n + 1$. In this case, π has the form $\langle \sigma_1, a_1, \sigma_2, a_2, \ldots, a_{n+1}, \sigma_{n+1} \rangle$. Consider the prefix $\beta = \langle \sigma_1, a_1, \sigma_2 \rangle$ of π. β has a length of 1, and hence has an expansion $\acute{\beta}$ in $T_{global}(\alpha)$. Now consider $\gamma = \langle \sigma_2, a_2, \ldots, a_{n+1}, \sigma_{n+1} \rangle$. By our inductive hypothesis, we know that γ has an expansion $\acute{\gamma}$ in $T_{global}(\alpha)$. Let $\acute{\pi} = \acute{\beta} \circ \acute{\gamma}$. By definition of an expansion, $\acute{\pi}$ is an expansion of π. Therefore by mathematical induction, if π is a path labeled by elementary actions in $T_{local}(\alpha)$, there exists an expansion $\acute{\pi}$ of π in $T_{global}(\alpha)$. $\square$

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

5.1 SUMMARY

In this work we introduced a general framework for reasoning about cooperative multi-agent systems. We began by extending the fundamental notion of an agent to facilitate communication via the introduction of requests. We then introduced the notions of an agent's local and global perspectives and their respective diagrams. These diagrams form the basis of our framework by providing the foundation upon which an agent perceives the world around him. Together these diagrams are used by an agent to construct plans, and reason about how they may play out in the domain. In addition we introduced the system diagram, which is used as a means of modeling the "actual state of the world" for the purposes of diagnosis and reasoning about the system at large.

Having introduced the basic framework we then presented an axiomatization of reasoning about agent communication. An example detailing a simple system was presented, and a methodology for representing agents to make use of the axiomatization was described in detail as well.

Finally, once the axiomatization was complete, we moved on to detailing some fundamental properties of the framework, specifically the relationship of paths in an agent's local diagram to those of it's global diagram. It was shown that given a cooperative multi-agent system, the paths of an agent's local diagram labeled by elementary actions have corresponding expansions in the agent's global diagram. Further properties will be built upon the results discussed here.

5.2 FUTURE WORK

This work presents only the foundation upon which a comprehensive logical theory of multi-agent systems may be built. Many open questions remain, chief among them are formalizations of many common phenomena (such as deadlocks, circular-wait conditions, starvation, etc.), as well as the expansion of the framework to deal with competitive multi-agent systems, and eventually systems containing both cooperating and competing agents. We examine each of these in turn in the following subsections.

5.2.1 *FORMALIZING PHENOMENA*

There is a large body of knowledge covering the topic of coordinating the efforts of, and the resources used by the agents of a given multi-agent domain. These topics first arose in the area of operating system research with the ability of systems to perform various forms of concurrent operations. By basing the framework on a set of transition diagrams, we believe that it will be possible to formalize the notions of deadlock, circular-wait, and the like in terms of properties of paths in the system diagram.

Consider the phenomenon of deadlock. A set of agents is said to be deadlocked if each agent of the set is waiting for some task that only another agent of that set may perform. A possible example of a deadlocked state in our framework is any state of the system diagram which contains a pair of fluents of the general form $pending(\alpha, \beta, r_1)$, $pending(\beta, \alpha, r_2)$.

Similarly with circular-wait. Such situations arise where the requests issued between the agents of the system for a loop. As an example, consider the scenario: "John asks his IT department to fix his computer. In order to do so the IT department asks a specialist to make an appointment. The specialist sends John an

email asking for a good time to stop by." A potential realization of such a scenario in our framework would be a state containing a set of fluents of the general form $\{pending(\alpha, \beta, r_1), pending(\beta, \gamma, r_2), pending(\gamma, \alpha, r_3)\}$.

It is our belief that a thorough exploration of these areas and a formalization of them as properties of paths in the system diagram could enable designers and implementers to make concrete assertions about the quality of plans that agents generate, in a similar fashion to the work done in [9]. In addition, comparing these formalizations with work done on representing these phenomena via Petri nets [8] may open new avenues of research.

5.2.2 *EXPANDING THE FRAMEWORK*

As was mentioned previously, our framework does not utilize a formal action language. As a result the translations making up our logic programs do not as of yet have formal proofs of soundness and completeness with regards to the various transition diagrams presented. In the future we plan to develop an action language suitable for representing multi-agent domains, together with a provenly correct translation of action description in this action language to logic programs.

The current definition of a multi-agent systems assumes that the agents in the domain do not change over time. Another possible avenue for expanding the framework is to enable agents to reason about the effects of agents entering or exiting the system. This could be coupled with work done on learning in [1], to have agents inquire about the capabilities of incoming agents.

We also hope to expand the applicability of our framework to other multi-agent domains. There are other forms of multi-agent systems aside from cooperative ones. The application of this work centers on the development of cooperative systems, and is not applicable to competitive or hybrid systems (systems containing both

competitive and cooperative agents). It is our hope that the methods described in this text may be applied towards expanding the framework both in the theoretic aspects, as well as expanding the communication modules to handle competitive domains. In combination with the development of more modular answer-set programming languages [10], these could then be bundled into a single, comprehensive library of axioms governing agent communication.

BIBLIOGRAPHY

[1] Marcello Balduccini. *Answer Set Based Design of Highly Autonomous, Rational Agents*. PhD thesis, Texas Tech University, December 2005.

[2] Marcello Balduccini. Computing Answer Sets of CR-Prolog Programs. Technical report, Computer Science Department, Texas Tech University, 2006.

[3] Marcello Balduccini. CR-MODELS: An inference engine for CR-Prolog. In *LPNMR 2007*, 2007.

[4] Marcello Balduccini and Michael Gelfond. Diagnostic Reasoning with A-Prolog. In *Theory and Practice of Logic Programming*, volume 3, pages 425 – 461, July 2003.

[5] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, January 2003.

[6] Chitta Baral, Gregory Gelfond, Michael Gelfond, and Richard Scherl. Textual inference by combining multiple logic programming paradigms. In *AAAI'05 Workshop on Inference for Textual Question Answering*, 2005.

[7] Chitta Baral and Michael Gelfond. Reasoning Agents in Dynamic Domains. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

[8] Daniel E. Cooke. Formal Specifications of Resource-Deadlock Prone Petri Nets. In *The Journal of Systems and Software*, volume 11, pages 53–69, January 1990.

[9] Jüergen Dix, Thomas Eiter, Michael Fink, Axel Polleres, and Yingqian Zhang. Monitoring Agents using Declarative Planning. Technical Report INFSYS RR-1843-03-10, Computer Science Department, Technical University of Vienna, November 2003.

[10] Michael Gelfond. Going places - notes on a modular development of knowledge about travel. In *AAAI Spring 2006 Symposium*, pages 56–66, 2006.

[11] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[12] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In Alessandro Provetti and Tran Cao Son, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, AAAI 2001 Spring Symposium Series, March 2001.

[13] Murray Shanahan. *Solving the frame problem: A mathematical investigation of the commonsense law of inertia*. MIT Press, 1997.

[14] Tommi Syrjänen. *Lparse 1.0 User's Manual*.

[15] Hudson Turner. Reprenting Actions in Logic Programs and Default Theories: A Situation Calculus Approach. *Journal of Logic Programming*, 31(1-3):245–298, June 1997.

APPENDIX: AGENT JOHN

OBJECTS OF THE DOMAIN

```
#domain time(T).


agent(john).
agent(secretary).
#domain agent(Agent).
#domain agent(Agent1).
#domain agent(Agent2).


object(ticket).
#domain object(Object).


property(packed).
property(ready).
property(possessionOf(Object)).
#domain property(Property).


boolean(true).
boolean(false).
#domain boolean(Value).
#domain boolean(Value1).
#domain boolean(Value2).
```

FLUENTS, REQUESTS, AND MESSAGES

```
inertialFluent(has(Agent,Property)).
#domain inertialFluent(InertialFluent).

fluent(InertialFluent).
#domain fluent(Fluent).

request(buy(Object,Agent)).
#domain request(Request).

clientOf(john,buy(Object,Agent)).
serverOf(secretary,buy(Object,Agent)).
mapsTo(john,buy(Object,Agent),has(Agent,possessionOf(Object)),true).

message(msg1(john)).
task(msg1(john),buy(ticket,john)).
receiver(msg1(john),secretary).
#domain message(Message).
```

STATE CONSTRAINTS

```
% An agent is ready if the agent is packed and has a ticket.
h(has(Agent,ready),T) :-
    h(has(Agent,packed),T),
    h(has(Agent,possessionOf(ticket)),T).
```

```
% An agent is not ready if the agent is not packed, or if the agent
% does not have a ticket.
-h(has(Agent,ready),T) :-
    -h(has(Agent,packed),T).

-h(has(Agent,ready),T) :-
    -h(has(Agent,possessionOf(ticket)),T).
```

ACTIONS AND EXECUTABILITY CONDITIONS

```
action(pack(john)).
type(pack(john),pack).
actor(pack(john),Agent).
effect(pack(john),has(john,packed),true).

action(send(john,Message)).
type(send(john,Message),send).
actor(send(john,Message),john).
sent(send(john,Message),Message).
#domain action(Action).

% Agents do not repack their bags.
-o(Action,T) :-
    type(Action,pack),
    actor(Action,Agent),
    h(has(Agent,packed),T).
```

APPENDIX: AGENT SECRETARY

OBJECTS OF THE DOMAIN

```
#domain time(T).


agent(john).
agent(secretary).
#domain agent(Agent).
#domain agent(Agent1).
#domain agent(Agent2).


object(ticket).
#domain object(Object).


property(online).
property(received(Object)).
#domain property(Property).


boolean(true).
boolean(false).
#domain boolean(Value).
#domain boolean(Value1).
#domain boolean(Value2).
```

FLUENTS, REQUESTS, AND MESSAGES

```
inertialFluent(has(Agent,Property)).
```

```
#domain inertialFluent(InertialFluent).


fluent(InertialFluent).
#domain fluent(Fluent).


request(buy(Object,Agent)).
#domain request(Request).


clientOf(john,buy(Object,Agent)).
serverOf(secretary,buy(Object,Agent)).
mapsTo(secretary,buy(Object,Agent),has(Agent,received(Object)),true).


message(msg1(john)).
task(msg1(john),buy(ticket,john)).
receiver(msg1(john),secretary).
#domain message(Message).
```

ACTIONS AND EXECUTABILITY CONDITIONS

```
action(logon(secretary)).
type(logon(secretary),logon).
actor(logon(secretary),secretary).
effect(logon(secretary),has(secretary,online),true).


action(buy(secretary,Object,Agent)).
type(buy(secretary,Object,Agent),buy).
actor(buy(secretary,Object,Agent),secretary).
```

```
effect(buy(secretary,Object,Agent),has(Agent,received(Object)),true).


#domain action(Action).


% The secretary cannot purchase an object if she is not online.
-o(Action,T) :-
    type(Action,buy),
    actor(Action,Agent),
    -h(has(Agent,online),T).
```

APPENDIX: COMMUNICATION MODULE C_{local}

EFFECTS OF COMMUNICATION ACTIONS

```
% rule LM1
effect(Action,Fluent,Value) :-
    type(Action,send),
    actor(Action,Agent),
    sent(Action,Message),
    task(Message,Request),
    mapsTo(Agent,Request,Fluent,Value).
```

EXECUTABILITY CONDITIONS

```
% rule LM2
-o(Action,T) :-
    type(Action,send),
    h(Fluent,T),
    effect(Action,Fluent,true).

% rule LM3
-o(Action,T) :-
    type(Action,send),
    -h(Fluent,T),
    effect(Action,Fluent,false).
```

APPENDIX: COMMUNICATION MODULE C_{global}

THE FLUENTS *pending* AND *satisfied*

```
% rule GM1
defaultFluent
    (
        pending(perspective(Client),Client,Server,Request);
        satisfied(perspective(Client),Client,Server,Request)
    ) :-
        clientOf(Client,Request),
        serverOf(Server,Request).

#domain defaultFluent(DefaultFluent).

% rule GM2
defaultsTo(DefaultFluent,false).

% rule GM3
fluent(DefaultFluent).

% rule GM4
inertialFluent
    (
        pending(perspective(Server),Client,Server,Request);
        satisfied(perspective(Server),Client,Server,Request)
    ) :-
```

```
        clientOf(Client,Request),
        serverOf(Server,Request).
```

COMMUNICATION ACTIONS

```
% rule GM5
effect(Action,pending(perspective(Client),
       Client,Server,Request),true) :-
    type(Action,send),
    actor(Action,Client),
    sent(Action,Message),
    receiver(Message,Server),
    task(Message,Request).

% rule GM6
action(wait(Agent,Message)) :-
    task(Message,Request),
    clientOf(Agent,Request).
type(wait(Agent,Message),wait).
actor(wait(Agent,Message),Agent).
msg(wait(Agent,Message),Message).

% rule GM7
h(satisfied(perspective(Client),Client,Server,Request),T + 1) |
-h(satisfied(perspective(Client),Client,Server,Request),T + 1) :-
    o(Action,T),
    type(Action,wait),
```

```
    actor(Action,Client),
    msg(Action,Message),
    receiver(Message,Server),
    task(Message,Request).

% rule GM9
-o(Action,T) :-
    h(pending(perspective(Client),Client,Server,Request),T),
    type(Action,send),
    actor(Action,Client).

% rule GM10
-o(Action,T) :-
    h(Fluent,T),
    type(Action,send),
    actor(Action,Client),
    sent(Action,Message),
    task(Message,Request),
    mapsTo(Agent,Request,Fluent,true).

% rule GM11
-o(Action,T) :-
    -h(Fluent,T)
    type(Action,send),
    actor(Action,Client),
    sent(Action,Message),
```

```
    task(Message,Request),
    mapsTo(Agent,Request,Fluent,false).


% rule GM12
-o(Action,T) :-
    -h(pending(perspective(Client),Client,Server,Request),T),
    type(Action,wait),
    actor(Action,Client),
    msg(Action,Message),
    receiver(Message,Server),
    task(Message,Request).
```

ACTION TRIGGERS

```
% rule GM8
o(Action,T) :-
    h(pending(perspective(Client),Client,Server,Request),T),
    type(Action,wait),
    actor(Action,Client),
    msg(Action,Message),
    receiver(Message,Server),
    task(Message,Request).
```

STATE CONSTRAINTS

```
% rule GM13
h(pending(perspective(Client),Client,Server,Request),T + 1) :-
    -h(satisfied(perspective(Client),Client,Server,Request),T + 1),
```

```
    h(pending(perspective(Client),Client,Server,Request),T).

% rule GM14
-h(pending(perspective(Client),Client,Server,Request),T) :-
    h(satisfied(perspective(Client),Client,Server,Request), T).

% rule GM15
h(Fluent,T) :-
    h(satisfied(perspective(Client),Client,Server,Request),T),
    mapsTo(Client,Request,Fluent,true).

% rule GM16
-h(Fluent,T) :-
    h(satisfied(perspective(Client),Client,Server,Request),T),
    mapsTo(Client,Request,Fluent,false).

% rule GM17
h(pending(perspective(Server),Client,Server,Request),T) :-
    h(pending(perspective(Client),Client,Server,Request),T).

% rule GM18
-h(pending(perspective(Server),Client,Server,Request),T) :-
    h(satisfied(perspective(Server),Client,Server,Request),T).

% rule GM19
h(satisfied(perspective(Server),Client,Server,Request),T + 1) :-
```

```
    h(pending(perspective(Server),Client,Server,Request),T),
    h(Fluent,T + 1),
    mapsTo(Server,Request,Fluent,true).

% rule GM20
h(satisfied(perspective(Server),Client,Server,Request),T + 1) :-
    h(pending(perspective(Server),Client,Server,Request),T),
    -h(Fluent,T + 1),
    mapsTo(Server,Request,Fluent,false).

% rule GM21
-h(satisfied(perspective(Server),Client,Server,Request),T) :-
    h(pending(perspective(Server),Client,Server,Request),T).

% rule GM22
h(satisfied(perspective(Client),Client,Server,Request),T) :-
    h(satisfied(perspective(Server),Client,Server,Request),T).

% rule GM23
:- h(satisfied(perspective(secretary),
     john,secretary,buy(ticket,john)),T),
   not h(satisfied(perspective(john),
         john,secretary,buy(ticket,john)),T).

% rule GM24
:- not h(satisfied(perspective(secretary),
```

```
      john,secretary,buy(ticket,john)),T),
h(satisfied(perspective(john),
  john,secretary,buy(ticket,john)),T).
```

APPENDIX: LIBRARY MODULES

DEFAULT FLUENTS

```
% rule DV1
h(DefaultFluent,T) :-
    defaultsTo(DefaultFluent,true),
    not -h(DefaultFluent,T).

% rule DV2
-h(DefaultFluent,T) :-
    defaultsTo(DefaultFluent,false),
    not h(DefaultFluent,T).
```

EFFECTS OF ACTIONS

```
% rule AF1
h(Fluent,T + 1) :-
    o(Action,T),
    effect(Action,Fluent,true).

% rule AF2
-h(Fluent,T + 1) :-
    o(Action,T),
    effect(Action,Fluent,false).
```

INERTIA AXIOMS

```
% rule IA1
h(InertialFluent,T + 1) :-
    h(InertialFluent,T),
    not -h(InertialFluent,T + 1).


% rule IA2
    -h(InertialFluent,T + 1) :-
    -h(InertialFluent,T),
    not h(InertialFluent,T + 1).
```