MCC Technical Report Number Carnot-017-92(Q)

Distributed Communicating Agents: Guide for Installation and Operation

Michael N. Huhns and Munindar P. Singh

January 1992

Abstract

This report describes a software tool for developing distributed, knowledge-based, communicating agents. The tool can be used to construct individual expert systems that communicate and cooperate with each other, and with human agents, in solving problems. The agents interact by using Rosette, an actor-based language, to manage their communications through TCP/IP and OSI. Thus, they can be located anywhere that is reachable through OSI. The agents also use Rosette to access databases. You can use this guide to construct, install, operate, and interact with teams of agents.

Microelectronics and Computer Technology Corporation
Carnot Project
Enterprise Integration Division
3500 West Balcones Center Drive
Austin, TX 78759-6509
(512) 338-3651
huhns@MCC.COM

Contents

Chapter 1

Introduction to Distributed Agents

Knowledge-based systems have become an important part of computing. After more than a decade of successful exploitation, there are now over 100,000 fielded systems [Feigenbaum 1988]. Most of these systems are small, involving small knowledge bases, single reasoning strategies, and specific domains of application. However, several trends have recently become apparent:

- Applications requiring an AI approach are increasingly large and diverse, yet existing expert-system tools work well only for small, specialized problems. Larger problems cannot be solved either because the necessary systems are too complex to build and maintain or because no single approach is adequate for all aspects of the problem. New tools are required that allow modular development of knowledge bases.
- Information resources abound throughout our environment, but there is no way for a single reasoning system to exploit the information available. Reasoning systems must be able to communicate and cooperate with one another and access multiple sources of information to solve a problem, just as human experts do.
- Once an expert system is developed with today's shells, it becomes an isolated island of knowledge. It cannot easily be reused in other, similar problem domains. Because of this, the cost of engineering and maintaining multiple expert systems is already becoming noticeably

high in the business community, where AI applications are increasingly commonplace.

- Knowledge based systems share a set of common needs, both for representing knowledge and for reasoning with it. The cost of building a new system in a new domain can be substantially reduced if it is written in a high-level language (often called a shell) that satisfies these common needs with built-in mechanisms.

These trends suggest the need for a new kind of system building tool, a tool that supports the construction of distributed, communicating, knowledge-based agents.

Distributed reasoning enables a set of knowledge-based systems, constructed quasi-independently, to act as a set of cooperating agents, working together to solve a problem. Developers of distributed reasoning systems can exploit a divide-and-conquer approach to development; they will be able to build smaller, more manageable knowledge-based agents. Smaller agents may represent alternative points of view on a problem. These smaller agents can also be *reused* in different combinations for solving additional problems as they arise. A further advantage of this approach is that it enables systems to be physically distributed in the world, just as the problems that they address are.

In this report we describe a tool, called *RAD*, that makes it possible to build distributed systems that solve large, complex reasoning problems. *RAD* is an extension of earlier MCC-proprietary expert system software. It provides high-performance forward and backward reasoning using Warren Abstract Machine (WAM) technology [Warren 1983], a frame system integrated into a typed unification algorithm [Aït-kaci, *et al.* 1985], a justification-based truth maintenance system [Russinoff 1985], and a contradiction resolution mechanism [Petrie 1987]. Recent extensions support OSI communications through Rosette [Tomlinson *et al.* 1991], access to multiple databases, and a substantially improved user interface.

RAD is a product of an ongoing research effort within the Carnot Project, whose goal is to provide tools for accessing and maintaining the consistency of data in distributed heterogeneous environments. This report describes how to install, construct, operate, and interact with multiple computational agents. Later reports will detail important applications and uses of these

agents for intelligent X.500 directory service, robust negotiation-based EDI, intelligent information retrieval, administration of multiple heterogeneous databases, and management of data consistency. We intend to exploit this version of *RAD* in building representative applications, in cooperation with our shareholders, and we intend to release a series of new *RAD* platforms that incorporate the results of those efforts.

A word of warning is appropriate here. Although *RAD* is intended to be a widely useful tool, the current release is only a research prototype: it is not a production-quality tool. Its only users so far are members of our research group. So if you decide to try to use it, please get in touch with us so that we can help. We are knowledgeable in several application areas, and will gladly give advice on the use of its novel features. We also value user input and guidance into the ongoing research effort.

1.1 Major Features of *RAD*

The other chapters of this report describe in detail each of *RAD*'s components. The rest of this section provides the context for those descriptions by quickly surveying the major components of the system and pointing out where the corresponding detailed descriptions can be found.

Knowledge Representation

Early artificial intelligence systems relied on first-order predicate logic as a language for representing domain knowledge. While this scheme is completely general and semantically clear, it has been found to be inadequate for organizing large knowledge bases and encoding complex objects. As an alternative, various frame-based languages have been employed. These languages are designed to support the natural representation of structured objects and taxonomies. They have proved to be well-suited for representing many useful relations, although they lack the general expressive power of the predicate calculus.

RAD combines the best features of these alternatives. It is frame-based, but allows knowledge to be expressed in terms of arbitrary predicates. It also represents relations as frames, so that a relation can have properties, any of its arguments can be typed (which *RAD* will then enforce), it can

have arbitrary arity, and it can inherit information from other frames. The representation of relations in *RAD* is described in Chapter 5.

In Chapter 4, we introduce frames, along with classes, attributes, metaclasses, and types. This chapter also describes simple data, including assertions of arbitrary arity that are attached to frames. Here we discuss the use of a justification-based truth maintenance system in connection with single-valued relations and inheritance.

Inference Mechanisms

Knowledge-based systems may also be classified according to inference methods. Most deductive systems may be characterized as either goal-directed (backward chaining) or data-directed (forward chaining). In a goal-directed system, logical implications are encoded as rules that are used by the system to reduce goals to simpler subgoals. This allows knowledge to be represented implicitly, without using space in the knowledge base, until it becomes relevant to a current problem. In this framework, however, it is difficult for the knowledge base designer to build control into a system. Data-directed inference, on the other hand, is based on production rules, which the system uses to derive all logical consequences of new data automatically. While control of inference is more natural within this paradigm, it uses space less efficiently, representing all knowledge explicitly.

RAD integrates both goal-directed and data-directed inference, allowing a knowledge engineer the freedom to decide whether each logical implication is more suitably represented as a backward rule or a forward rule. It also allows many inferences to be made quickly and efficiently through inheritance in the frame structure. Chapters 6 and 7 describe the *RAD* backward and forward chaining inference systems, respectively, and their integration with a truth maintenance system.

Truth Maintenance System

Another central feature of *RAD* is a nonmonotonic justification-based truth maintenance system (JTMS), which is integrated with the inference system at the architectural level. Based on [Doyle 1979], the *RAD* JTMS records logical inferences and dependencies among data as they are derived by both forward and backward chaining. This allows efficient revision of a set of

beliefs to accommodate new information, the retraction of a premise, or the discovery of a contradiction [Petrie 1989]. It also facilitates the generation of coherent explanations, which can be used for explanation-based learning as demonstrated by the MCC Argo project [Huhns and Acosta 1988]. We will see in Chapter 2 how it also forms the basis for negotiation among cooperating agents [Huhns and Bridgeland 1991]. Chapter 3 contains a discussion of data dependencies and the *RAD* JTMS. Chapter 8 describes the contradiction resolution mechanism.

Distributed Reasoning

The distributed reasoning facility allows reasoning to be distributed across multiple symbolic reasoners that can thus collaborate in solving a problem. To do this requires four things:

- A protocol for interagent communication,
- An "interpreter" to make the agents interpretable to each other in the case that they were developed using different representational frameworks,
- A common knowledge base of problem solving strategies that serves as the basis of collaboration, and
- A generalization of key reasoning mechanisms that transforms them from global methods suitable for use by a single agent into semilocal methods that can be distributed across agents.

The current release of *RAD* provides a protocol for interagent communication as well as a generalization of the JTMS facility. It allows multiple agents, each implemented in *RAD*, to communicate with each other. It supports a distributed truth maintenance system (DTMS) so that each agent can rely on the results of another's reasoning without having to keep track of the details of that reasoning. This DTMS enforces local consistency within each agent, while enabling negotiation about inconsistencies among agents.

Future versions will include communication protocols that will enable agents constructed in other rule-based languages, such as CYC and OPS5, to interact with *RAD* agents. Future versions also will increase the effectiveness and efficiency of the *RAD* agents by providing a common knowledge

base of problem solving methods. This knowledge base will support models for the beliefs, goals, and intentions of each agent. Agents will then have an understanding of each other and the roles that they play in an overall application. Their actions will then be flexible, but robust, and applicable to dynamic real-time problems such as those that arise when acccessing and managing integrated networks of information.

High-Performance Implementation

RAD represents a refinement of the Proteus language coupled with a high performance implementation based on the Warren Abstract Prolog Machine (WAM) [Warren 1983], which provides inference speed comparable to that of the fastest commercial Prolog systems. Instructions are generated from *RAD* rules by a WAM compiler and interpreted by a WAM emulator, both written in Lisp. One advantage of this scheme is that it facilitates the extension of the WAM instruction set to provide the functionality necessary for expert system applications, allowing the integration of the WAM with other Lisp-based *RAD* components [Bridgeland 1989]. Also, since the WAM instructions are interpreted directly by *RAD* (in contrast to Prolog systems that compile them further into native machine code), *RAD* programs generate relatively compact code, thereby allowing larger applications.

User Interface

Chapter 9 is devoted to facilities designed to aid the system builder. This chapter discusses the user-interface processes, the *RAD* built-in relations and commands, the syntax for communicating with agents, and the parser that interprets this syntax.

The user interface in *RAD* is implemented as a pair of processes, one to monitor a keyboard for input from a user and the other to communicate with computational agents. Eventually, it will allow communication with computational agents constructed in other expert system languages, such as OPS5 and KEE.

Current Implementation of *RAD*

RAD is implemented in a combination of Common Lisp, Rosette, and C. The main representation and symbolic reasoning mechanisms are implemented in

Common Lisp. The user interface and the communication component of the distributed reasoning system are implemented in C. Future releases may contain a substantially larger proportion in C, because we are exploring the possibility of implementing all of *RAD* in either C or C++. The network component of *RAD* is written in Rosette to take advantage of the OSI and TCP/IP communication protocols it provides. A window interface is implemented using X Windows.

A number of expert system applications developed by MCC shareholders [Steele 1989, Kodak 1988, Kirchen 1989, Harp and Sederberg 1988, Virdhagriswaran *et al.* 1987, Virdhagriswaran and Pitts 1987], based on Proteus, have confirmed the viability of our approach and the utility of the *RAD* functionality, particularly for the solution of design problems. The Appendix describes several additional examples of *RAD* usage.

Chapter 2

Distributed Reasoning

RAD supports two kinds of agents—human and computational—as well as databases, as shown in Figure 2.1. It does this by making use of recent technological advances in communication protocols and distributed artificial intelligence (DAI). DAI is concerned with the cooperative solution of problems by a decentralized group of agents. It is the appropriate technology for applications where

- expertise is distributed, as in design;
- information is distributed, as in office automation and enterprise integration;
- data are distributed, as in distributed sensing;
- decisions are distributed, as in manufacturing control; and
- knowledge bases are developed independently but must be interconnected or reused, as in next-generation knowledge engineering.

This chapter describes the primitives and features that *RAD* incorporates to support distributed reasoning among both human and knowledge-based agents. The agents are then able to cooperate in solving problems, to share expertise, to work in parallel on common problems, to be developed and implemented modularly, to be fault tolerant through redundancy, to represent multiple viewpoints and the knowledge of multiple human experts, and to be reusable. Additional benefits are discussed in [Huhns 1987], [Bond and Gasser 1988], and [Gasser and Huhns 1989].

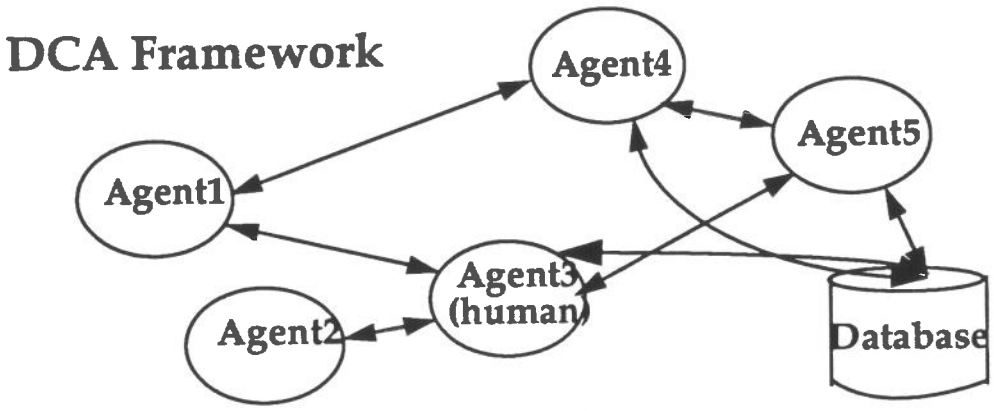


Figure 2.1: Architectural overview of the distributed communicating agent environment

2.1 Model for Distributed Reasoning

The following is a typical scenario for the use of *RAD*: there is a loosely-coupled network of experts, each being an expert in a particular field, and there is a problem that these experts must solve that is beyond the capabilities of any one of the experts. These experts can together solve the problem, but they must cooperate to do so. Furthermore, some of these experts might be knowledge-based systems, i.e., computational agents, some might be humans, and the remainder might be lower-level computational entities, such as databases, software simulators, and neural nets. (In this chapter, we refer generically to all of these as "agents.") *RAD* provides specific assistance for the development of computational agents, provides interfaces for humans to interact with other agents, and provides the overall framework within which all of these kinds of agents can operate and interact. Their interaction enables their cooperation and, ultimately, the solution of their problem.

The *RAD* framework and its corresponding collection of agents execute on a network of computer workstations. The agents operate within this framework asynchronously and, in general, autonomously. *RAD* permits the collection of agents to be dynamic, allowing agents to come and go. It accomplishes this by supporting a rudimentary learning capability, whereby

agents learn about each other when they receive messages. Rosette actors serve as communication aides, one representing each agent or database, to forward these messages (see Figure 2.2). They use a tree-space mechanism implemented through OSI and TCP/IP protocols. The actors also buffer messages when the agents they represent are busy. These actors exist as entities in an Extensible Services Switch or ESS (described in Section 2.4.2). Each ESS can be on a different host. Each ESS also contains a special actor, called *aideServer*, that maintains a partial directory of agent and database locations. The directory is updated when agents connect or disconnect from an ESS.

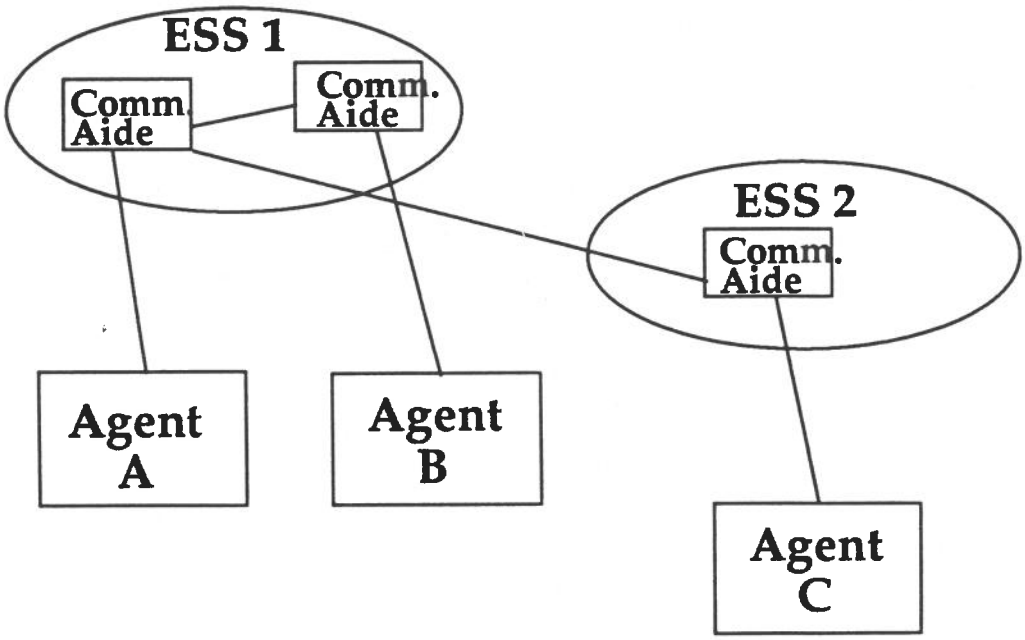


Figure 2.2: Each agent or database has an aide—a Rosette actor—that manages its communications

The design of the *RAD* framework is based on our observations of how a user interacts with a single, conventional knowledge-based system. A user can query and make assertions to the knowledge-based system, to which

the system responds with either an answer or an acknowledgement, as appropriate. As a side-effect of forward or backward inferencing, the system can independently inform or make a request of a user, who must answer all requests. Interactions can thus be either user-initiated or system-initiated.

Similarly, *RAD* agents communicate at a fundamental level by exchanging messages, specifically, queries and assertions. The agents can be either passive or active, i.e., they can either respond to questions and commands from another agent or initiate dialogs with another agent.

A problem is then solved by *RAD* in the following way. At any given time there will be a number of identically-structured[1] computational agents, each having its own domain specific knowledge, and a number of users, each assisted by an interface. A problem, or a goal to be satisfied, is given to one of these agents by a user. This agent does one of the following two things:

1. Solves the problem or satisfies the goal itself, using local knowledge and internal reasoning mechanisms.

2. Decomposes the problem or goal into subproblems, distributes each one of these subproblems to an appropriate agent, and then integrates the solutions returned by these agents.

In either case, the agent reports the final solution to the user. Note that each of the other agents, upon receiving a subproblem, also does one of the above two things.

The features required for this type of distributed problem-solving include a communication channel among the agents, a communication protocol for exchanging goals and solutions on this channel, and a distributed reason-maintenance system that enables globally-coherent solutions to be achieved. We have incorporated these features into the current version of *RAD*.

2.2 Architecture and Use

RAD extends this model to the multiple agent case by enabling system-initiated interactions to be directed to another agent, as well as to a local user. For this purpose, a destination field is added to each query or assertion. A value for this field, i.e., the name of an agent who should receive the query

[1] At least for now.

or assertion, can then be the subject of a reasoning process in the originating agent. Since an assertion can now come from several sources, its justification must be modified to record a source. An agent must also record the agents to whom it sent an assertion, so that it can inform them in case its belief in that assertion ever changes.

Agents are implemented as a *CLASS* in *RAD*. This implies that all agents[2] have to be instances of the class **AGENT**. Agents have the following basic attributes:

```
Name:       <Agent Identifier>
Reliable:   <A boolean>
Unreliable: <A boolean>
Said:       <List of assertions>
Was-told:   <List of assertions>
```

They store values of these attributes for themselves and each other. These attributes refer to a history of interactions with an agent and its trustworthiness. The agents exist in a fully distributed environment. Each agent can make queries of other agents and also serve as a knowledge base for other agents. That is, each agent works in a *Client-Server* mode. It is a *Client* when it requests other agents to solve a problem or satisfy a goal. It is a *Server* when it satisfies goals for other agents. An agent can be in one of the following three states:

- Busy
- Query
- Free

An Agent is in a BUSY state if it is performing internal inferencing operations. In this state, all requests and messages from other agents are buffered on its standard port. These requests are processed in order of arrival once the agent returns to the FREE state.

When an agent queries another agent, it can either go into a QUERY state until it receives a result or dispatch the request and carry on with other chores. The first choice refers to a *serial* way of solving problems and

[2]Agents must have compatible class structures.

the second one refers to a *parallel* way of solving problems. In the current implementation, the agents operate serially. When an agent is waiting for a reply to a query, it will not permit other agents either to perform destructive operations on it or to make queries of it, as this may lead to a potential deadlock. Its communication aide buffers all messages until one with an awaited tag arrives.

Agents can also tell facts to other agents. This occurs asynchronously, and the receiving agent does not acknowledge the fact. This feature can be used to simulate *parallel* problem-solving that was mentioned above. To do so, the agents' knowledge must be expressed as forward rules, which automatically trigger on the assertion of facts in an agent. We have developed a declarative implementation of the *Contract Net* protocol [Smith and Davis 1981] based on this principle.

A user can also utilize the windowing system to view and edit the knowledge base of a remote agent from the user's terminal. The interface is described in detail in Chapter 9.

The following are the salient issues in distributed reasoning:

- Remote queries
- Remote assertions
- Remote justifications (to enable the implementation of a distributed JTMS [Mason and Johnson 1989])
- Negotiation

2.2.1 Interagent Queries

An agent or user can make queries of other agents. This is one of the most basic operations in *RAD*. When an agent sends a query to a second agent, the second agent uses its reasoning mechanisms to find an answer. If it succeeds, then it returns its answer to the first agent, who records the answer, along with a justification that the source of the answer was the second agent. Meanwhile, the second agent remembers its answer, along with a justification that corresponds to the reasoning process it used and a note that it communicated this result to the first agent. The queries can be either simple

queries, i.e., queries at the user-interface level, or proof attempts of a literal of a rule at a remote agent.

More specifically, the goal to be proved is sent to the remote agent (the *server*) along with a *Client Id.* This registers a request at the *Server*'s aide. Eventually, this request is forwarded to the *Server*, which then attempts to prove the given goal. If a successful proof is obtained, a datum corresponding to the proved literal is created. This datum has the justifications for the proof of the goal. This datum also is associated with a *remote justification*, which has a field to store the Client Id. Note that this datum participates as any other datum in the local JTMS. The operations described above are performed for each successful proof of the given goal. Finally all the proved literals are returned as a list to the client. Now if the server ever changes its beliefs about some literals or rules that result in a change in status of a datum, it checks to see if that datum has an associated remote justification. If it does, the server sends an update message to all the listed clients.

The above mechanism requires the server to remember every proof it performs for its clients and to remember their names. Sometimes, this is unnecessary and wasteful of storage and computation. Therefore, we have also provided the option of having a goal proven by a remote agent, where the remote agent does not remember the result. The former option is called **prove** and the latter option **query**.

When an interagent query with caching of results by the server occurs in a forward or backward rule, its syntax must be

```
prove(<Agent name> <goal>)
   or
prove-once(<Agent name> <goal>)
```

When an interagent query with no caching of results by the server occurs in a forward or backward rule, its syntax is

```
query(<Agent name> <goal>)
   or
query-once(<Agent name> <goal>)
```

An example of a command from a user to `Agent1` to query Agent2 and `Agent1`'s subsequent reply is (note that the syntax is different from the interface than from within a rule)

```
User: query Agent2 color(?x ?y)
Agent1: color(My-Ford Red)
        color(My-VW Blue)
```

The messages that are exchanged by the communication aides encode the sender, the intended recipient, and a tag that uniquely identifies the message.

The Client now creates a datum of the type *Client-Datum* for each proof received from the server. This is an "image" of the server datum. Information about the server is stored in this datum. This datum is given a remote justification, which is a special kind of justification. Finally, this datum is stored in a local associative table at the client with the tag being the associative key. Note that this datum participates in the client's JTMS. Hence, the query could be a part of a rule. This datum is special, in that its status (`IN/OUT`) depends not only on local justifications, but also on proofs from remote agents. This is the justification for the above bindings. Finally, this datum is assigned the status `IN`.

2.2.2 Remote Assertions

In *RAD*, we have provided agents with the capability to "tell" facts to other agents. This kind of interagent assertion, or remote assertion, is analogous to an assertion that a user might have made. Remote assertions are made using the **Tell** command, whose syntax is given below. Such a fact, when asserted on a remote agent, is justified by a remote justification. This remote justification has the pointer to the agent that made that particular assertion. Since any number of users or computational agents can communicate with an agent, assertions are justified by the source of the assertion, which may be a user or an agent.

The syntax for a remote assert is

```
tell(<hostname> <assertion>)
```

2.2.3 Remote Justifications and Distributed TMS

Remote data and justifications are created as described in the previous section. When the remote agent changes its belief about a datum, i.e., changes its belief status, and this datum happens to be of type Server-Datum, then

the corresponding Client, which is responsible for the creation of this datum, is notified. At the receipt of this message at the client, the "image" datum at the client is accessed. It is this datum that needs to be acted upon. If the transition is from an `OUT` to an `IN`, a new remote justification is created at the Client and is pushed onto the justifications of the client datum and the status of the datum is made `IN`. If the transition is from an `IN` to an `OUT`, the client datum is made `OUT` by an **erase** operation. Again, the justification for the subsequent `ERASE` datum is a newly created remote justification. The information about the agents causing the above changes is stored in the remote justifications. The Client also has an option of ignoring the messages from its servers, when the remote data it depends on change their status. This is set by a flag.

Note that the above remote justifications are necessary only for proofs whose justifications are needed. This may exclude simple user-level queries, whose proof tree (or justification information) is transient. An example where the justification is needed is remote proof attempts on one or more antecedents of a forward rule. These remote justifications will have to be entered in the justification of any assertion of the consequents of this forward rule.

2.2.4 Negotiation

There is no requirement for two agents to agree completely. That is, each agent maintains a locally consistent set of beliefs using its own justification-based truth-maintenance system, but global consistency among the agents is not required. However, when two or more agents disagree about belief in a datum *and when this disagreement is encountered during problem solving*, then negotiation among the agents will ensue to resolve the disagreement. The negotiation is necessary to ensure that the global solutions to the problems posed to the agents are coherent.

The negotiation procedure involves an exchange of justifications among the agents. For example, if there is a disagreement about belief in datum D_1, then the agents exchange the justifications for their belief or disbelief in D_1. The agents then compare these justifications. If there is a disagreement about belief in any datum in the justifications, then the justifications for this disputed datum are exchanged, and the process recurses. It is expected, but not yet proven, that resolution of all disagreements among two agents in a

consistent world will be achieved by one agent informing another of some explicit knowledge that the other is missing. The negotiation process is an active area of research at MCC and will be described further in a forthcoming technical report.

2.3 Distributed Reasoning Example

Figures 2.3–2.5 show an example of the use and operation of our multiagent truth maintenance algorithm as two agents interact. Figure 2.3 shows the initial state of the knowledge bases for the two agents, Client, an investor, and Broker, a stockbroker. First, Client asks Broker to recommend a stock. Client knows what it can afford, but it does not have any knowledge about what stocks it would be wise to buy. It does know, however, that Broker knows about this problem. Broker does have some general knowledge about what stocks to recommend and about what stocks it can afford.

When Broker receives Client's query, it attempts to find an answer by applying its rules. It succeeds and binds tt ?X to `XCorp`. It also creates a server datum for this conclusion, containing the conclusion, the justification for the conclusion (rules 1 and 2), and a pointer to the agent to whom this conclusion was reported (Client). Broker recommends `XCorp`, which causes Client to believe that he should buy that stock, as shown in Figure 2.4. However, Broker then learns (not shown) that the basis for his recommendation, that `XCorp` is cash-rich, is no longer valid. He revises his beliefs and notifies Client that he has retracted his recommendation for `XCorp`. The final knowledge bases for the agents are shown in Figure 2.5.

It is interesting to note that these agents maintain different beliefs about whether or not they can afford to buy the stock of `XCorp`. Allowing this difference in belief—this global inconsistency—is useful in this case. It allows different viewpoints to be represented, it simplifies the representation of knowledge, in that the predicate `afford` really should have an additional argument indicating *which* agent can or cannot afford the stock, and it eliminates the interactions that would be needed to resolve the difference. Of course, the system would detect, and subsequently correct, the difference if the agents ever share or discuss this predicate. Details of the justifications that are constructed when two agents exchange beliefs are shown in Figure 2.6.

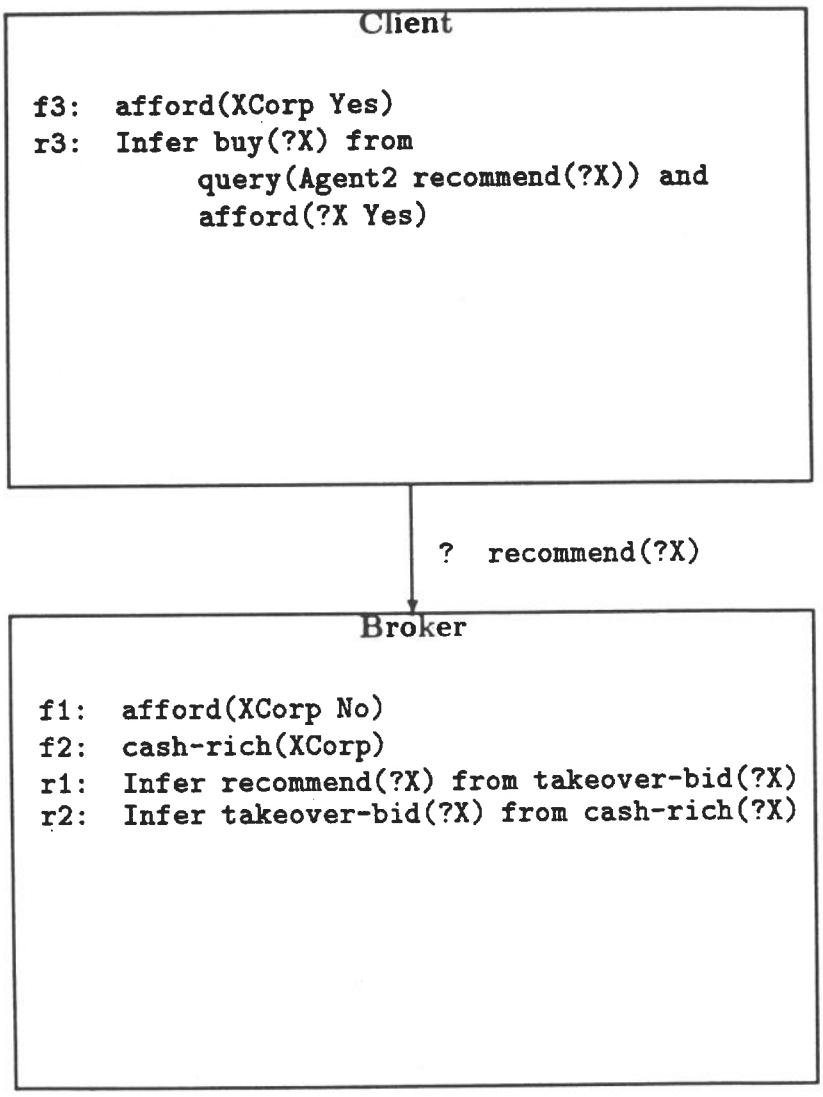


Figure 2.3: Initial knowledge bases of two interacting multiagent-JTMS-based agents, before Client queries Broker for a recommendation

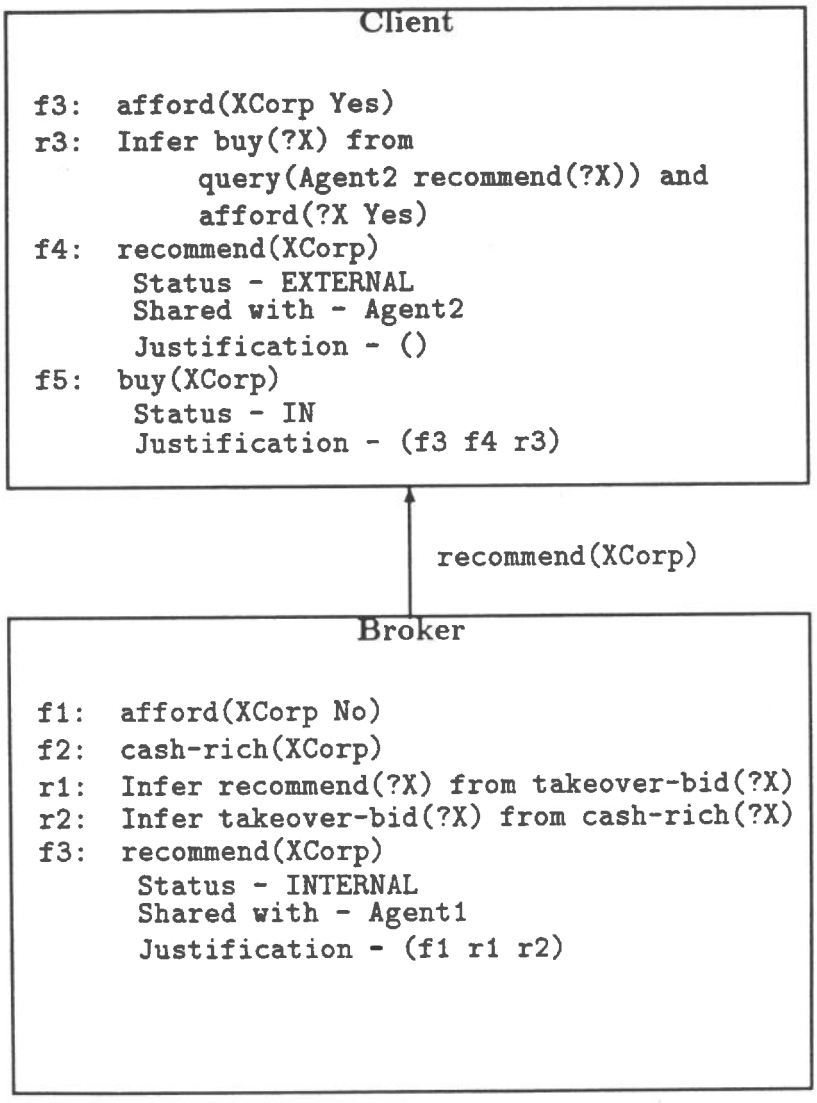


Figure 2.4: Knowledge bases of the agents after Broker has replied to Client's query

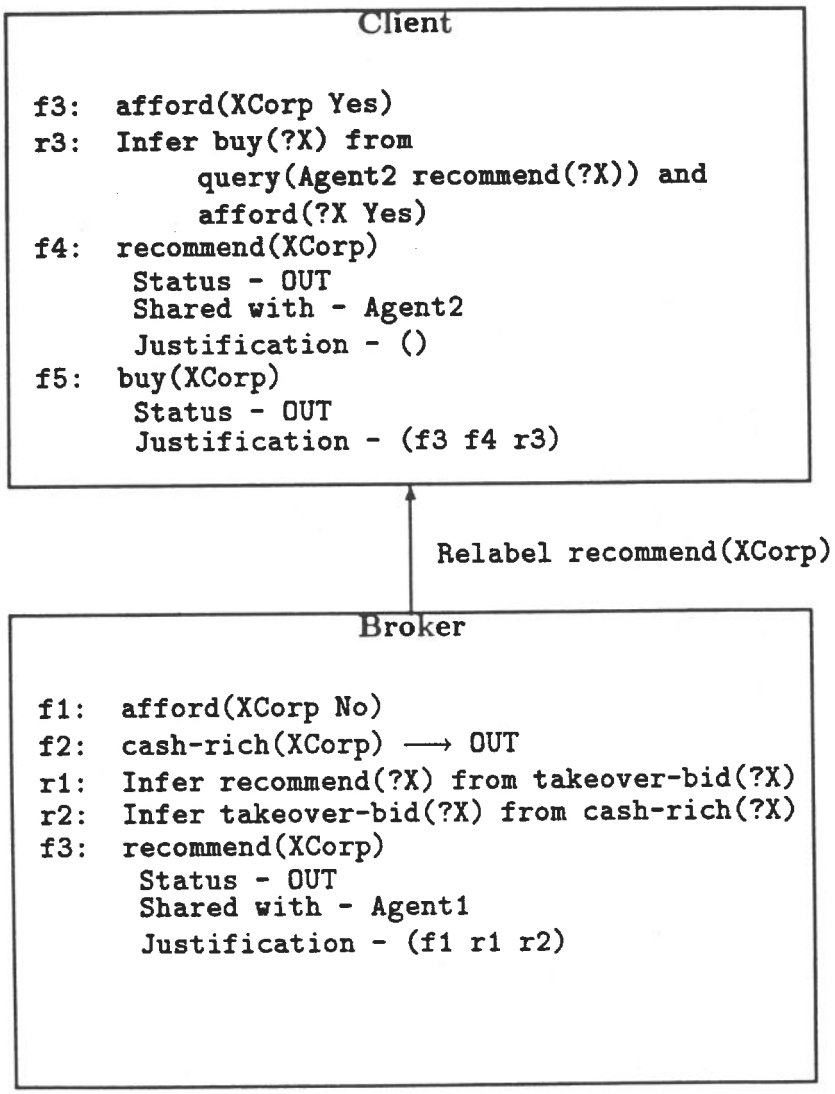


Figure 2.5: Final knowledge bases of the agents after Broker has notified Client that a fact must be relabeled

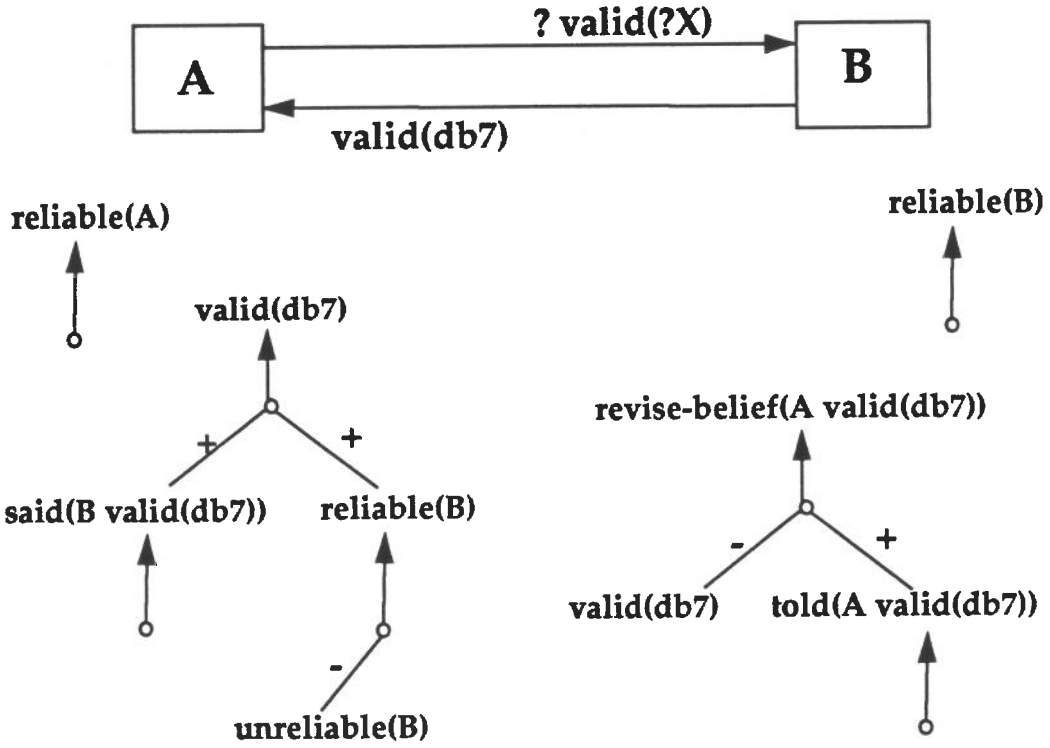


Figure 2.6: Justification networks that are constructed by two agents that exchange beliefs (see Chapter 3 for notation)

2.4 Implementation Issues

2.4.1 Protocol for Message Passing

The communication aides utilize the Rosette tree-space mechanism to pass messages. This is shown in Figure 2.7. The agents establish TCP/IP connections from the machine on which they are executing to a machine running an Extensible Services Switch or ESS (please see Section 2.4.2 for details). Establishment of a connection causes the creation of a communication aide and a tree-space interface for each agent.

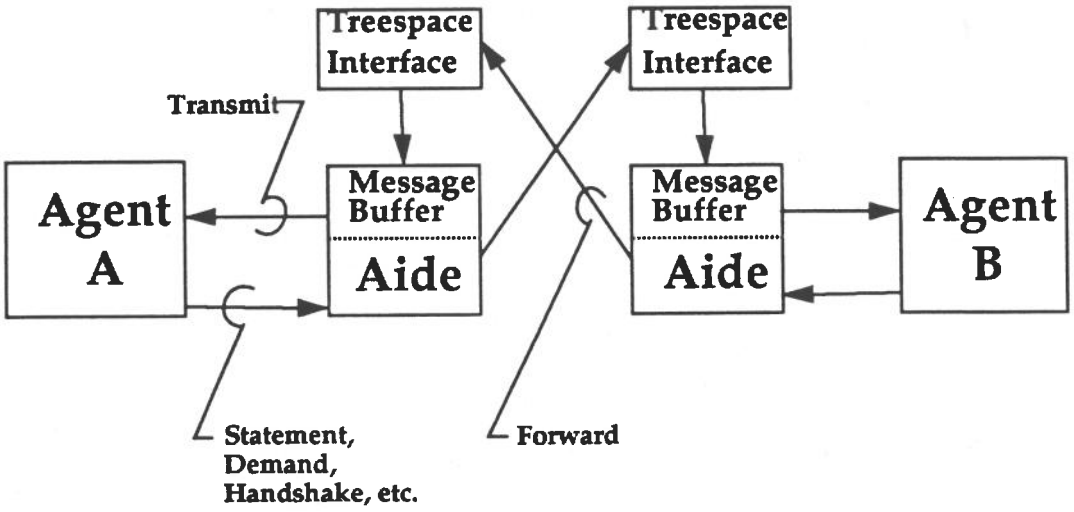


Figure 2.7: Each communication aide has an associated tree-space interface that passes messages for agents or databases

Once a connection is established, agents and databases can exchange six basic types of messages: demands, responses, database queries, statements, errors, and handshakes. Each of these message types encodes information about the sender, the communication aide, the recipient, the identifying tag, and the contents. The message types and their specific behavior are described next.

Demand: a message for which the sender expects a response to be returned. Queries between agents are *Demands*, but so are assertions and com-

mands from a user, since they require an acknowledgement. The semantics of a demand is that the agent will not assume that the remote agent has performed the operation until a response has been received. A demand message has the following syntax:

```
(outDemand <my-aide> <destination> <new-tag> <message>)
```

The local communication aide forwards this message through the tree space to its destination and then waits for a response with a tag that matches `<new-tag>`. While it is waiting, it buffers all other messages for its agent. This means that agents will remain idle until they receive a response to their demand. Interfaces, however, can continue to send out other messages.

Database Demand: a message to be sent to a database. The message contents are an SQL statement to be processed by the database. It has the following syntax:

```
(Eval (db-query DBAgent <SQL-statement>[])[<destination>])
```

Since databases are passive entities, it convenient to let the response of a database demand arrive as a result of computing the given db-query on the destination.

Response: a message in response to a *Demand* message. It uses the tag from the demand to which it is responding. It has the following syntax:

```
(outResponse <my-aide> <destination> <old-tag> <message>)
```

Statement: a message from one agent to another, where a reply is not expected. This is an asynchronous input to an agent. An example of a *Statement* is a print message to the interface. A statement has the following syntax:

```
(outStatement <my-aide> <destination> <new-tag> <message>)
```

Error: a message issued as a response to a *Demand* message, when the *Demand* message caused an error in the agent. It has the following syntax:

```
(outError <my-aide> <destination> <old-tag> <message>)
```

Handshake: a message that an agent sends to its communication aide, informing it that the agent is FREE and is ready for the next message. It has the following syntax:

```
(handshake <my-aide>)
```

In order for the agents and interfaces to be robust in the presence of unexpected messages, expecially when they are waiting for a response to one of their demands, we have implemented message handling by the communication aides according to the state diagram shown in Figure 2.8. The state machine specifies the operational semantics of messages that are sent and received by agents and interfaces.

2.4.2 Extensible Services Switch (ESS)

Each agent must be connected to an ESS. The ESS then provides communication facilities and a simple directory mechanism for maintaining the locations of agents, databases, and other ESS's.

Whenever an agent is started, it registers itself with an ESS. The syntax for registration is

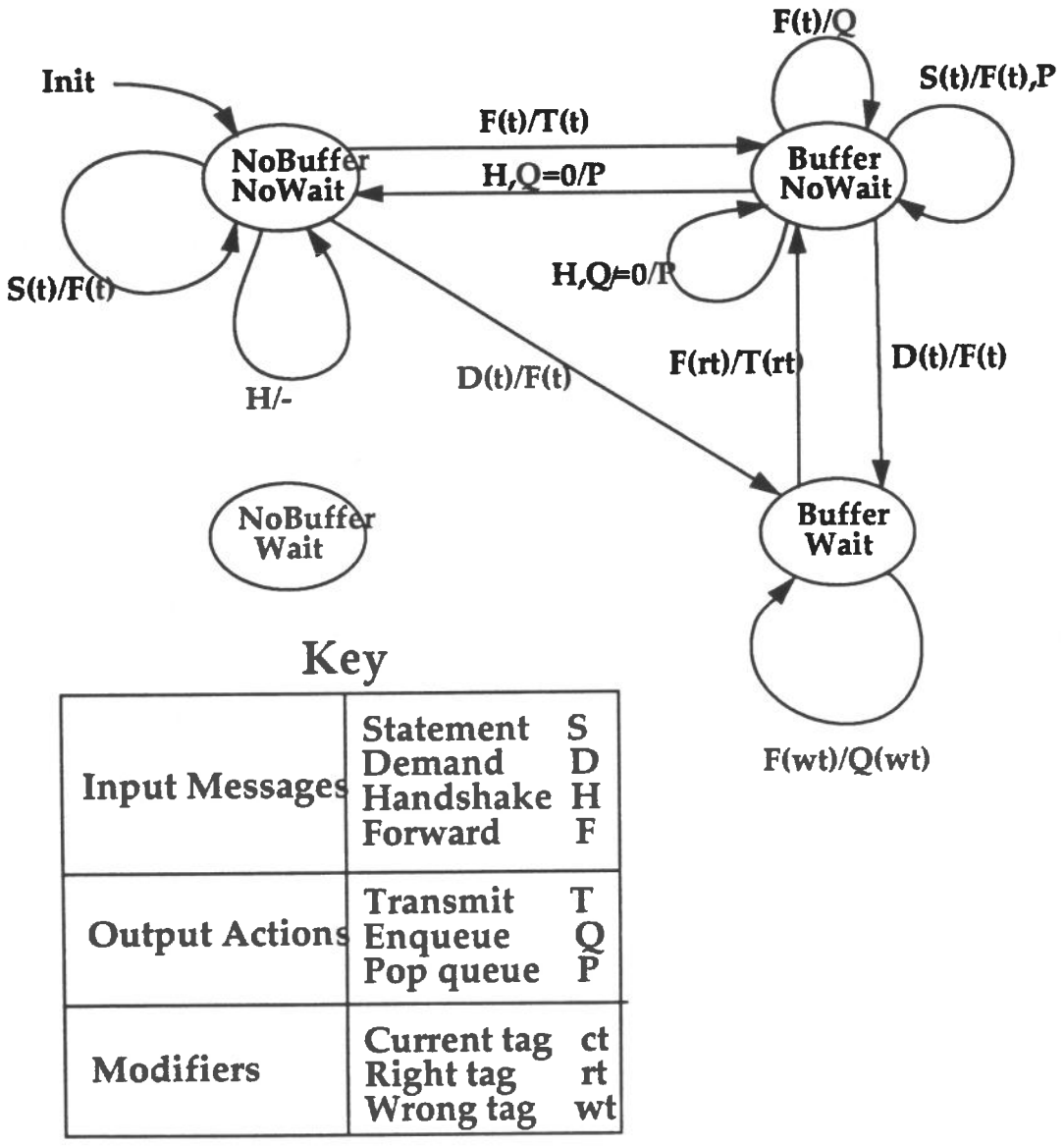
```
(makeAide aideServer <Aide Name> <Agent Name>)
```

Whenever an agent halts or ceases to exist, it unregisters from the ESS by sending the message

```
(killAide aideServer <Agent Name>)
```

At any time, an agent can unregister from one ESS and then register with another. The other messages supported by the ESS for the purpose of agent interactions are

- **db-connect**, for connecting a database to an ESS,
- **db-disconnect**, for disconnecting a database from an ESS,
- **essOf**, for determining whether a given agent is registered and finding its ESS, and



Key

Input Messages	Statement	S
	Demand	D
	Handshake	H
	Forward	F
Output Actions	Transmit	T
	Enqueue	Q
	Pop queue	P
Modifiers	Current tag	ct
	Right tag	rt
	Wrong tag	wt

Figure 2.8: A state machine for the operational semantics of messages sent and received by computational agents and interfaces

- **who**, for producing a list of all agents and databases, and their ESS's known to an ESS.

The remainder of this document primarily describes the behavior, functionality, and structure of a single agent.

Chapter 3

Truth Maintenance

Each datum (or element of a *RAD* knowledge base) represents a potential belief. The status of this belief, which is subject to change, is reflected in the *support-status* of the datum, the value of which may be `IN`, indicating that it is currently believed, or `OUT`, indicating current disbelief. This value is assigned by a justification-based truth-maintenance system (JTMS) in accordance with a list of *justifications* that have been attached to the datum.

Each justification consists of a pair of lists of data, the `IN`-list and the `OUT`-list of the justification. A justification is said to be *valid* and is considered to represent reason for belief in its associated datum if each element of its `IN`-list is `IN` and each element of its `OUT`-list is `OUT`. The justified datum is said to depend *monotonically* on each member of the justification's `IN`-list and *nonmonotonically* on each member of the `OUT`-list.

Also associated with each datum is a list of other data called its *supporters*. The supporters of a datum are considered to be responsible for its current support-status.

It is the function of the JTMS to assign support-statuses and supporters to data in a manner that is consistent with their justifications, and to adjust these assignments continually as required by the addition of new justifications and the retraction of old ones. More precisely, the state of the knowledge base, as constructed by the JTMS, must satisfy two requirements: *stability* and *well-foundedness*. A *stable* state is one that satisfies the following conditions:

1. A datum is `IN` if it has at least one valid justification. In this case its list of supporters is the result of appending the `IN`-list and `OUT`-list

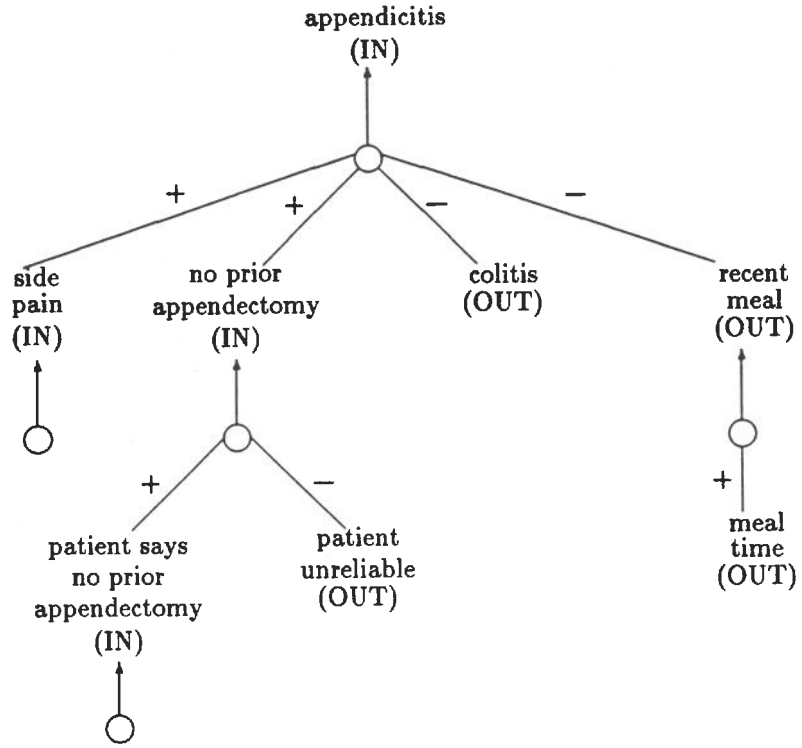


Figure 3.1: A stable well-founded state

of one of its valid justifications. This justification is identified as the *supporting justification.*

2. A datum is `OUT` if it has no valid justification. Its supporters then include one representative of each of its invalid justifications: either an `OUT` member of the `IN`-list or an `IN` member of the `OUT`-list.

The requirement of well-foundedness is that no set of beliefs be mutually dependent, i.e., there may be no sequence of data $d_0, \ldots, d_n$, all of which are `IN`, such that $d_0 = d_n$ and for $i = 1, \ldots, n$, d_{i-1} is a supporter of d_i.

An example of an admissible state is shown in Figure 3.1. In this graph and those that follow, each circle corresponds to a justification, with an arrow pointing to the justified datum, positive arcs connected to the elements of the `IN`-list, and negative arcs to elements of the `OUT`-list. Thus, the datum representing a diagnosis of appendicitis has a valid justification with a two-element `IN`-list and a two-element `OUT`-list. The belief that the patient has a

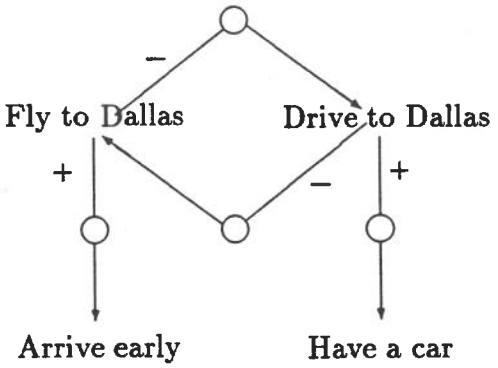


Figure 3.2: Alternative assumptions

side pain is supported by a justification with an empty IN-list and an empty OUT-list and is said to be a *premise*. The datum representing the unreliability of the patient has an empty list of justifications and is therefore OUT. If this datum were to acquire a new valid justification, then its support-status as well as those of the data that depend on it (directly or indirectly) must be reevaluated, ultimately forcing the diagnosis OUT. This phenomenon, the development of a new belief resulting in the abandonment of an old one, characterizes *nonmonotonic reasoning*.

In the presence of nonmonotonic dependencies, the status-assignment problem may not have a unique solution. In the situation shown in Figure 3.2, the JTMS may succeed either by making Fly-to-Dallas (and hence Arrive-early) IN and Drive-to-Dallas (hence Have-a-car) OUT, or by giving the opposite assignments. This choice between alternative hypothetical assumptions can only be made arbitrarily, and may have to be revised later as new justifications are produced (e.g., if Drive-to-Dallas acquires a new valid justification while Fly-to-Dallas is IN).

Circularities involving nonmonotonic dependencies may also impose unsatisfiable constraints on the JTMS, a situation that may be difficult to detect. Two simple examples of this are shown in Figures 3.3 and 3.4. Note that the network of Figure 3.4 does have a stable state (in which all data are IN), but this state is ill-founded, and therefore inadmissible.

Figure 3.5 illustrates a situation in which a datum Q is given a new justification and, in order to restore stability, the JTMS must examine not only Q and the data that depend on Q, but also those on which Q depends. In this example, if only the support-statuses of P and Q were reassigned, the system would be forced to report an unsatisfiable circularity. In order

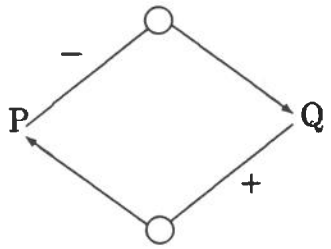


Figure 3.3: Unsatisfiable dependencies

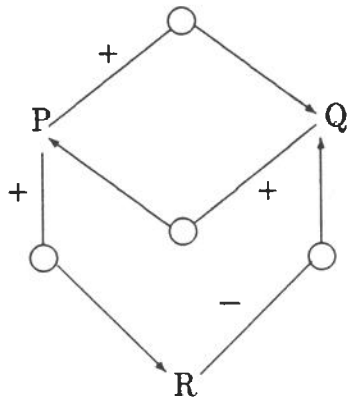


Figure 3.4: No well-founded stable state

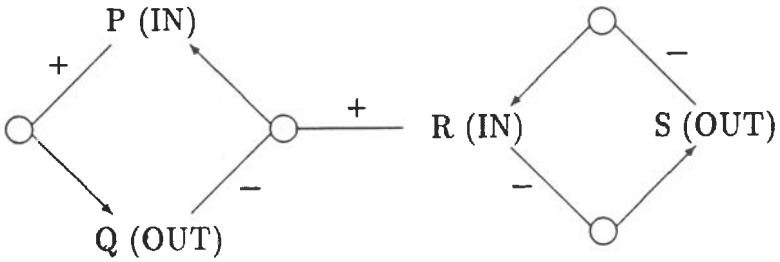


Figure 3.5: A solvable odd loop

to succeed, it must also consider R (on which Q now depends) and thus S (which depends on R). The only solution is to put S `IN` and P, Q, and R `OUT`.

As described in [Russinoff 1985], the *RAD* JTMS is *complete* in the sense that given any database with any set of justifications, it will achieve a stable well-founded state if such a state exists, and otherwise will recognize and report failure. This represents an improvement over the original TMS [Doyle 1979], as well as other published procedures for truth maintenance [Charniak *et al.* 1980, Goodwin 1984]. These systems all fail (perhaps even fail to terminate) in the presence of certain circular dependencies that have been characterized as *odd loops*. An odd loop is a cycle of arcs with an odd number of minus signs, as in Figures 3.3, 3.4, and 3.5. A dependency network containing such a loop may or may not be satisfiable. (We have already observed that the network in Figure 3.5 does admit a solution.) While the presence of odd loops complicates the truth maintenance task and is generally considered undesirable, they are sometimes unavoidable in practice, particularly in dependency networks that are based on input from several users.

In the following chapters, we shall see how data dependency networks are created in *RAD* by the user, by the frame system, and by both forward and backward inference.

Chapter 4

Frames and Assertions

The data on which the JTMS operates represent statements about objects. Before discussing the structure of these data, we shall describe the objects that they concern. These objects, called *frames*, are the subject of this chapter.

4.1 Classes, Subclasses, and Instances

In the initial state of the system, there exist several frames. One of these, named **CLASS**, plays a special role as discussed below. The others are **AGENT**, **DATABASE**, **RELATION**, **LIST**, **CONS**, **NULL**, **SYMBOL**, **NUMBER**, **FIXNUM**, **FLOAT**, **BIT-VECTOR**, **VECTOR**, and **STRING**. By means of the commands described in Chapter 9, the user may enlarge this set by creating new frames.

There are two primitive relations defined on frames: *Instance* and *Subclass*. If a pair (x, y) is an element of the instance relation, we say that *x is an instance of y*, or that *y is the type of x*. For a pair (x, y) in the subclass relation, we say that *x is a subclass of y* or that *y is a superclass of x*. Figure 4.1 depicts these relations as they are defined in the initial state. Broken lines are drawn from instances to types; solid lines are drawn from subclasses to superclasses. Note that **CLASS** is the type of every system-defined frame.

Two other important relations are defined in terms of these primitives: *Subclass** and *Instance**. The relation *Subclass** is defined as the reflexive transitive closure of the relation *Subclass*. Thus, *x is a subclass* of y* (equiv-

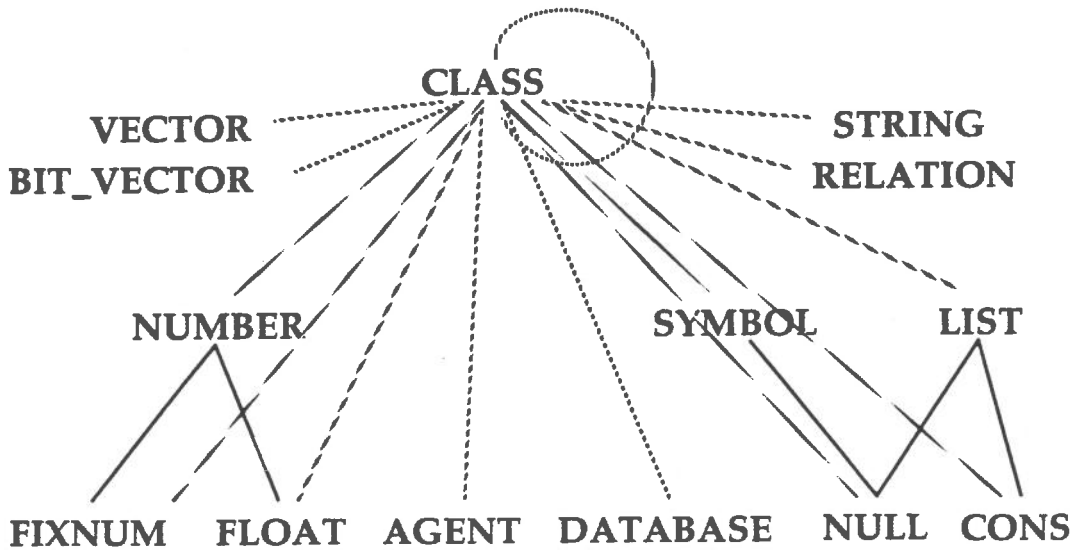


Figure 4.1: Built-in classes. Solid lines denote subclass relationships and dashed lines denote instance relationships.

alently, *y is a superclass* of x*) if either x is identical to y, or x is a subclass of a subclass* of y. The relation *Instance** is defined as follows: *x is an instance* of y* iff x is an instance of a subclass* of y. In this case we may also say that *x is a y*. In particular, a *class* is by definition a frame that is an instance* of **CLASS**. Note that every system-defined frame is a class, including **CLASS** itself.

As new frames are created by the user, he may also extend these relations by assigning types to instances and creating subclass relationships (between user-defined classes only). This must be done in such a way, however, that at each stage of the development, the following properties are preserved:

1. The instance relation is a function, i.e., for each frame x there exists a unique frame y such that x is an *Instance* of y.

2. **CLASS** is the only frame that is an *Instance* of itself. Thus, whenever a new frame is created, some preexisting frame must be specified as its type.

3. The relation *Subclass** is a partial order. That is, if x is a *Subclass** of y and y is a *Subclass** of x, then x and y are identical.

4. If x is an *Instance* of y, then y must be an *Instance** of **CLASS**.

5. If x is a *Subclass* of y, then x and y must each be an *Instance** of **CLASS**.

Thus, according to the last two of these properties, only a class may have a subclass, superclass, subclass*, superclass*, instance, or instance*. There is a further restriction on the classes that may be instantiated by the user: a user-defined frame may be an instance of **CLASS**, **RELATION**, or of any user-defined class, but it may not be an instance of **LIST**, **CONS**, **NULL**, **SYMBOL**, **NUMBER**, **FIXNUM**, **BIT-VECTOR**, **VECTOR**, **STRING**, or **FLOAT**. Instead, whenever the system encounters a Common Lisp object whose datatype is the name of one of these system-defined classes, the object automatically becomes an instance of the named class. For example, if the number *3* is read, it becomes an instance of **FIXNUM** and thus a member of **NUMBER** (in other words, a number). When the symbol `NIL` is encountered, it is recognized as the unique instance of the class **NULL**, and hence both a list and a symbol.

An example of a user-defined system of frames is illustrated in Figure 4.2. This example involves eight new classes, all of which are subclasses of the class **PERSON** and instances of the class **CLASS**. For clarity, classes are denoted in bold-face and other user-defined frames in typewriter font. `HILARY`, for example, as an instance of **TA**, is not a class, but is a TA, a graduate, a staff, a student, an employee, and a person.

4.2 Metaclasses

Of course, the existence of classes as frames provides the advantage of being able to reason about classes at the same level at which one reasons about the objects of which they are comprised. It is often desirable to be able to reason about classes of classes as well. The only class we have seen so far that has classes as instances is **CLASS** itself. Any class that has a class as an instance must be a subclass of **CLASS**, in which case all of its members are

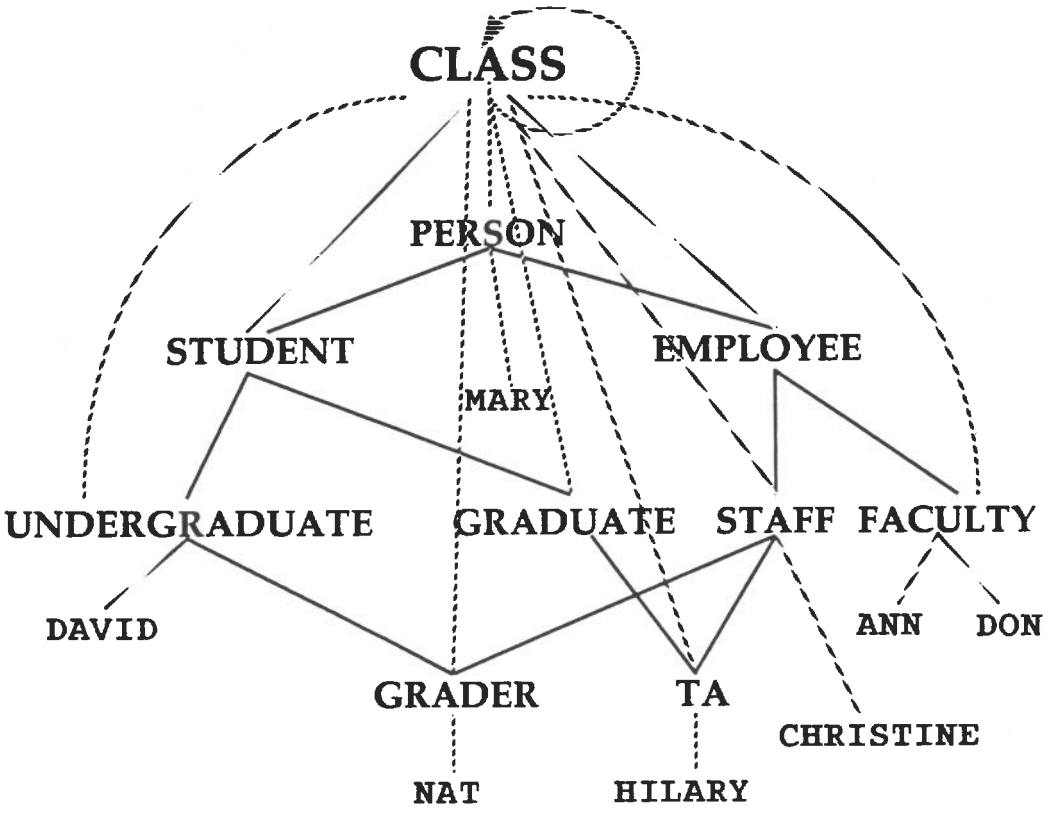


Figure 4.2: User-defined classes

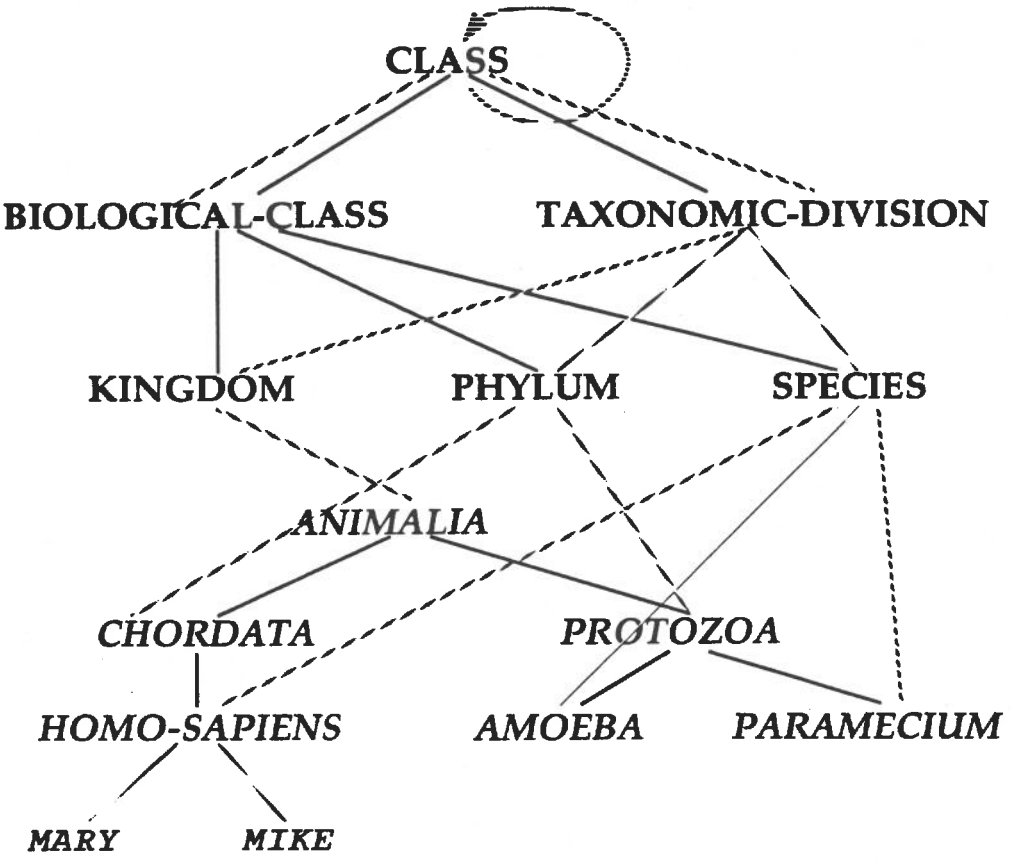


Figure 4.3: User-defined metaclasses

classes. Such a class is called a *metaclass*. While **CLASS** is the only system-defined metaclass (and the only one in the example of Figure 4.2), additional metaclasses may be created simply by asserting a subclass relationship from any user-defined classes to **CLASS**.

The example of Figure 4.3 includes several user-defined metaclasses (denoted in large bold print). This example is based on six classes of animals: **ANIMALIA** (the class of all animals) and five of its subclasses, which are related as indicated by the solid arrows connecting them. These classes could be constructed simply as instances of **CLASS**. But in order to represent and utilize the knowledge that these classes share more

than mere classhood, we first define a metaclass called **BIOLOGICAL-CLASS**, intended to include the animal classes among its instances. In fact, the animal classes are partitioned into smaller metaclasses by defining them as instances of **KINGDOM**, **PHYLUM**, and **SPECIES**, which are subclasses of **BIOLOGICAL-CLASS**. Note that **KINGDOM**, **PHYLUM**, and **SPECIES** are themselves not simply instances of **CLASS**, but are classes by virtue of being instances of the metaclass **TAXONOMIC-DIVISION**. Moreover, they are metaclasses by virtue of their links to the metaclass **BIOLOGICAL-CLASS**.

4.3 Variables and Types

Along with the objects that appear in RAD data, there are also occurrences of variables. These are denoted as symbols with initial character "?". Variables, as usual, represent unspecified objects. The process of unification, which is central to the mechanisms of forward and backward chaining, is built on the basic operation of binding (i.e., assigning values to) variables. An unbound variable `?X` may be bound either to an object or to another unbound variable `?Y`. In the latter case, if `?Y` is subsequently bound, its binding also becomes the binding of `?X`.

Since objects in this system are classified by their types, it is natural and useful to classify variables as being of a certain type [Aït-kaci, *et al.* 1985]. This is done by appending the name of a class to the variable name, using ":" as a separator, as in `?X:STUDENT`. The consequence of assigning a type to a variable is that any binding of the variable is required to be an instance of the variable's type.

In the presence of typed variables, the standard unification algorithm must be altered in several ways. First, before a variable is bound to an object, it must be verified that the object is an instance of the variable's type. Second, before a variable is bound to another variable, it must be verified that the types of the two variables are compatible, so that it will be possible later to bind them to the same object. Thus, the two types must have a nontrivial common subtype. Finally, when a variable `?X` is bound to a variable `?Y`, the type of `?Y` must be replaced in order to ensure that any later binding of `?Y` is consistent with the type of `?X`. The new type of `?Y` should be the most general common subtype, or *greatest lower bound*, of the

type of ?X and the old type of ?Y. For example (see Figure 4.2), a variable ?X:UNDERGRADUATE could be bound to a variable ?Y:STAFF, with type of ?Y replaced by **GRADER**. ?Y could then be bound to NAT, but no longer to HILARY, which would violate the type restriction on ?X.

Thus, the modified unification algorithm depends on the computability of the greatest lower bound (g.l.b.) of two variable types. Unfortunately, the partially ordered set of classes does not form a *lattice*, i.e., the g.l.b. of two classes may not exist. The classes **STUDENT** and **EMPLOYEE** of Figure 4.2, for example, have two common subclasses, **GRADER** and **TA**. Since neither of these is a superclass of the other, neither can be said to be the g.l.b. of **STUDENT** and **EMPLOYEE**.

This problem is solved by generalizing the notion of *type*. A type is now defined to be a set of classes, none of which is a subclass of another. A type t_1 is a subtype of a type t_2 if each of the classes of t_1 is a subclass of at least one of the classes of t_2. An object is said to belong to a type if it is an instance of at least one of its classes. It follows that an object belongs to a type if it belongs to any of its subtypes.

Under this ordering, the set of all types forms a lattice. The type of a variable may be any instance of this lattice. The identification of each class c with the type $\{c\}$ induces an embedding of the partially ordered set of classes into the lattice of types. In this context, the g.l.b. of two classes may always be computed. Thus, the g.l.b. of **STUDENT** and **EMPLOYEE** is the type {**GRADER**,**TA**}.

The empty set of classes, denoted BOTTOM, is a subtype of every type. This type contains no objects and is not allowed as the type of a variable. If the g.l.b. of the types of two variables (e.g., ?X:STAFF and ?Y:FACULTY) is BOTTOM, then these variables cannot be unified; their types contain no common objects.

The set of all maximal classes is also a type, denoted **T**. It is a supertype of every type, and every object belongs to it. If no type is specified for a variable, then the variable's type is taken to be **T** as a default. There is no restriction on the binding of such a variable.

The cost of the expressive power of typed variables is the resulting complication of the unification algorithm. In order to minimize this cost, the g.l.b. operation must be a fast computation. This is accomplished by means of an encoding scheme that associates with each class c a bit-string $\mathcal{B}(c)$, in such a way that c_1 is a subclass of c_2 if and only if $\mathcal{B}(c_1)$ is (bit-wise) less

than or equal to $\mathcal{B}(c_2)$. For a type $t = \{c_1, \ldots, c_k\}$, $\mathcal{B}(t)$ is constructed as the logical-or of the $\mathcal{B}(c_i)$. The g.l.b. operation then reduces to logical-and. The details of this scheme are described in [Aït-kaci, *et al.* 1985].

4.4 Assertions

There are two main classifications of data: *rules* and *assertions*. Rules, which are used by the system to derive assertions from other assertions, are classified as *forward rules* and *backward rules*. These are the subjects of Chapters 6 and 7. Assertions, as described in this section, represent simple statements. Specifically, an *assertion* is a proposition that resides in the knowledge base, where a proposition is a list whose members are a relation symbol followed by arguments.

An assertion is further classified as *general* or *particular* according to whether or not it contains any variables. The variables in a general assertion are understood to be universally quantified. Thus, the assertion

```
User: owes(?U:undergraduate ?F:faculty 15)
```

represents the statement "Each undergraduate owes each faculty member 15 dollars."

4.4.1 Instance Slot Values

A frame may assume values for a given relation if the frame is an instance of the class for which the relation is defined (see Chapter 5). In this case, an *instance slot* corresponding to the relation is attached to the frame. One or more values may be stored in this instance slot. Thus, if a relation *Description* has argument-types **PERSON** and **HUMAN-ATTRIBUTE**, then the frame `NAT` may acquire *Description* values. An assertion such as

```
User: description(nat tall)
Zeus: Instance Slot Value DESCRIPTION-1 accepted.
```

results in the creation of a datum called an *Instance Slot Value*, which is stored in `NAT`'s *Description* slot. This datum is justified as a premise (i.e., with empty `IN`-list and `OUT`-list). Thus, an instance slot value is just an assertion pertaining to a relation for an instance of a class.

When a relation is initially defined, it is either specified as *Single-valued* or is *multiple-valued* by default. A frame may assume any number of coexisting values for a multiple-valued relation. If *Description*, for example, is multiple-valued, then asserting a new *Description* for NAT via

```
User: description(nat thin)
```

has no effect on the old value. Thus, a query for NAT's *Description* produces both values:

```
User: ?? description(nat ?X)
Zeus: DESCRIPTION(NAT TALL)
      DESCRIPTION(NAT THIN)
```

For a single-valued relation, on the other hand, a frame may have only one effective value at any time. Suppose the relation *Nationality* is defined for class **PERSON** and declared to be single-valued, and that a *Nationality* is asserted for NAT:

```
User: nationality(Nat French)
Zeus: Instance Slot Value NATIONALITY-1 accepted.
```

If another value is later asserted for the *Nationality* of NAT, then the old value is overridden by the new one:

```
User: nationality(Nat Swiss)
Zeus: Instance Slot Value NATIONALITY-2 accepted.

User: ?? nationality(Nat ?X)
Zeus: NATIONALITY(NAT SWISS)
```

Actually, this restriction to a single value is enforced by the JTMS: whenever a value is asserted for a single-valued relation, it is added to the OUT-list of each justification of any preexisting conflicting value. Thus, the old value remains in the database, but is ignored in answering the query because it is now OUT. This method not only provides for the construction of explanations, such as

```
User: why NATIONALITY-1
Zeus: Instance Slot Value NATIONALITY-1
```

```
JUSTIFICATION:
    in-list  NIL
    out-list  (NATIONALITY-2)

is invalidated by

Instance Slot Value NATIONALITY-2
```

but also allows an old value to be reinstated if the overriding value is removed:

```
User: erase nationality-2
Zeus: OK.

User: ?? nationality(nat ?X)
Zeus: NATIONALITY(NAT FRENCH)
```

In fact, *RAD* always ensures that the value that is IN is the one with the most recently created justification that is currently valid. Note that if several conflicting values for a slot are asserted in succession, then a reassertion of the original value will result in a complicated dependency network, including odd loops. This implementation, therefore, requires a complete JTMS as discussed in Chapter 3.

Instance slot values do not have to be binary. For example, consider representing "Nat owes David $5." This would be written in *RAD* by using the ternary relation *Owes* as follows:

```
User: assert-instance (relation owes)
User: argument-types(owes (person person number))
User: single-valued(owes)
User: owes(nat david 5)
```

Further, the default that "Nat owes everybody $10" could be written

```
User: owes(nat ?X:person 10)
```

4.4.2 Class-Slot Values

Values for a relation may be attached to subclasses of its defining class as well as to its instances. Each of these subclasses contains a *class slot*, the

values in which may be inherited by any instance of the class to which it belongs. For example, the command

```
User: description(?X:person aerobic)
Zeus: Class Slot Value DESCRIPTION-1 accepted.
```

adds a value to the *Description* class slot of **PERSON**: This datum represents the belief that every instance of **PERSON** is AEROBIC. Similarly, any subclass of **PERSON** may acquire class values for this relation:

```
User: assert-instance (class texan)
User: assert-subclass (person texan)
User: ?? description(?X:texan ?Y)
Zeus: DESCRIPTION(?X:TEXAN AEROBIC)
```

The *Description* values associated with an instance of **TEXAN** are then the instance values specifically assigned to it, along with the class values assigned to the superclasses of its type. Hence,

```
User: assert-instance (texan connie)
User: description(connie pretty)
User: ?? description(connie ?X)
Zeus: DESCRIPTION(CONNIE PRETTY)
      DESCRIPTION(CONNIE AEROBIC)
```

For single-valued relations, a more complicated mode of inheritance is used. A value for a single-valued class slot may be inherited by an instance of the class only if no value has been assigned to that instance's instance slot. Class slot values for these relations are therefore called *default values.*

Default values assigned to a given class override each other in the same manner as instance values for a given frame, so that only one default value assigned to a class may be IN at any time. If a value for a relation is sought for a given frame, the frame's instance slot is first examined for an IN value. If there is none, then each of the superclasses of the frame is examined (in depth first order) until an IN default value is found.

Suppose that for the single-valued attribute *Nationality*, the default values AMERICAN, CHINESE, and INDIAN are asserted for the classes **PERSON**, **STUDENT**, and **GRADUATE**, respectively. Suppose further that NAT's *Nationality* is asserted to be FRENCH and that DON's is GERMAN. Then a query

for the *Nationality* values for all instances of **PERSON** would produce the following:

```
User: ?? nationality(?X:person ?Y)
Zeus: NATIONALITY(DON GERMAN)
      NATIONALITY(ANN AMERICAN)
      NATIONALITY(HILARY INDIAN)
      NATIONALITY(NAT FRENCH)
      NATIONALITY(DAVID CHINESE)
      NATIONALITY(MARY AMERICAN)
```

If a new default value were now asserted for **STUDENT**, only DAVID's *Nationality* would change:

```
User: nationality(?X:student carolinian)

User: ?? nationality(?X:student ?Y)
Zeus: NATIONALITY(HILARY INDIAN)
      NATIONALITY(NAT FRENCH)
      NATIONALITY(DAVID CAROLINIAN)
```

If the default value for **GRADUATE** were retracted, then *HILARY* would inherit from **STUDENT**:

```
User: erase nationality(?X:graduate indian)

User: ?? nationality(?X:student ?Y)
Zeus: NATIONALITY(HILARY CAROLINIAN)
      NATIONALITY(NAT FRENCH)
      NATIONALITY(DAVID CAROLINIAN)
```

Chapter 5

Relations in RAD

5.1 Relations as Objects

In *RAD*, an *object* is a cluster of knowledge, i.e., a thing to which attribute values are attached and about which propositions are formed and inferences are drawn. An object is also a thing that is subject to a classification according to which it inherits properties. Of all the structures that are manipulated by an inference system, which ones are to be handled explicitly as objects in this sense?

In an object-oriented or frame-based system, any structure about which one cares to reason should have the status of an object. In Proteus, the only built-in objects are a few system-defined classes. Predicates are not objects in the above sense, and neither are rules, assertions, or slots. These structures are allowed to occur as arguments of predicates, but they may not have attribute values and they cannot inherit properties. This has proved to be a deficiency—the machinery that has been developed for classifying and reasoning about objects is not available in dealing with these less fortunate structures, even though it might be quite useful. Raising their status to make them objects seems to be desirable in terms of the power of our system, the simplicity of its implementation, and the clarity of the user's model. Thus, the following are full-fledged objects:

Classes Every object is an instance of at least one class. If classes are objects themselves, then they must be instances of other classes. These classes of classes are called *metaclasses*, and they can be manipulated in the

same ways as ordinary classes.

Relations Conceptually, a relation of arity n is a set of n-tuples of objects. In Proteus, these are partitioned into "predicates" and "attributes." A predicate may have arbitrary arity and there are no restrictions on the form of its arguments. An attribute is always binary, and has a well-defined domain. That is, its first argument is constrained to belong to some specified class for which the attribute is defined. Thus, an attribute allows values to be attached to objects—it may be thought of as a mapping from some domain of objects into some set of values. If attributes (more generally, relations) themselves have the status of objects, then properties such as reflexivity, symmetry, etc. may be explicitly associated with them. It would also allow a classification of relations according to mode of inheritance and other properties.

The structure of a frame system is defined by the primitive relations among its objects. Each of these relations may be viewed as a built-in attribute, defined for some class of objects. Some of the most obvious of these are

1. *Type*, defined for the universal class of all objects;
2. *Instance* and *Subclass*, defined for the class of all classes;
3. *Argument-types* and *Single-valued*, defined for the class of relations;

Other built-in attributes represent the links between attributes and slots, slots and objects, rules and rule instances, objects and their parts, etc. One advantage of representing these primitives as attributes is that it allows the structure of the frame system to be defined within the frame system itself. Another advantage is that if instance and subclass links (which comprise the paths through which properties are traditionally inherited) are special cases of slot values, then it begins to make sense to think about supporting inheritance through arbitrary relations. Relations in *RAD* have three significant characteristics:

- In *RAD* there is a single class of object—relation—that is of arbitrary arity. A given relation can either be *Specificity-ordered*, in which case it does support inheritance, or not.

- In *RAD* relations are objects. This allows inheritance of information to relations.
- In *RAD*, there is a class **RELATION**. Every relation is an instance of that class.

5.2 Relations on Relations

Most of the following relations on relations are lazy: e.g., a relation can be asserted to be *Axiomatic*, and if later that assertion is killed, then the relation will no longer be *Axiomatic*. A few are not lazy and once asserted are permanent, such as *Specificity-ordered*, *Single-valued*, and (to some extent) *Argument-types*. The restrictions that apply are explained in detail with each of these three relations. Note that it is only difficult, but not impossible, to implement these three relations lazily.

Axiomatic *reln* Primitive Axiomatic

Most relations have instances that are objects in the JTMS, i.e., an assertion about such a relation would have a support status and justifications. However, the instances of some relations are not JTMS objects, because it would be cumbersome and inefficient to tag everything that depended on them. Such relations are said to be *axiomatic*. For example, *Argument-types* (defined below) is axiomatic; if it were not, a premise assertion of the backward rule

```
Infer foo(?X) from ( bar(?X) baz(?X) )
```

would have to be justified with the facts that *Foo*, *Bar*, and *Baz* all take the argument types that they in fact do.

Note that although axiomatic relations can be single-valued, it is an error to assert an "extra" value for one. (This error handling is not yet implemented.)

Primitive *reln* System Axiomatic

Some relations are *Primitive*, in that users can only make ground unit assertions with these relations. This restriction is imposed because some relations (e.g., *Subclass*) are handled specially by *RAD* machinery that assumes only ground unit assertions. If the user attempts to assert nonground

or nonunit classes about *reln*, *RAD* will warn the user of his mistake and refuse to accept the assertion.

System *reln* System Axiomatic

Some relations are completely defined by the system. For example, *Subclass** is defined as the transitive closure of *Subclass*, and it is a mistake for the *RAD* user to attempt to assert clauses about *Subclass** directly. Such attempts will be rejected with an error message, because *Subclass** is a *System* relation.

A relation cannot be both primitive and a system relation, although this is not currently enforced.

Action *reln* System Axiomatic

The group of relations that perform actions are designated by the unary relation *Action*. *Action* relations are typically used as consequents in forward rules.

Argument-types *reln list-of-types* Primitive Axiomatic

Each relation has a fixed arity, and restrictions on the types of its arguments. *List-of-types* is a list of types of logical variables. A type can be either a class, a conjunction of classes (written as a list of the classes), or the unrestricted type (written as T).

The arity of *reln* is the length of *list-of-types*, and only tuples of arguments that unify with the types in *list-of-types* are permitted as tuples of arguments to the relation. Thus *Argument-types* generalizes the Proteus *domain* and *range* constructions to all the arguments of a relation.

Operationally, *Argument-types* has the following effects:

- An assertion that is not consistent with the *Argument-types* assertion will be rejected with an error message. For example if

```
argument-types(power-consumption
               (electrical-device number power-unit))
```

and a user asserts

```
power-consumption(fan-25 100 joules)
```

RAD will reject the assertion because the user has the wrong units.

- Only proofs that are consistent with *Argument-types* assertions will be found. Thus, no proofs will be found of `power-consumption(zebra ?x ?y)`.

Note that because *Argument-types* is multidirectional, it is also good for queries of what relations are defined on an argument of some type. For example, one can determine the relations defined on fan-25 by querying `argument-types(?x (fan . ?ignore))`.

If argument-types is not specified for a relation, but the arity of the relation is deduced from other knowledge, the default argument-types is a list of unrestricted types (**T**) of the right arity. The argument-types of a relation can be changed, but one cannot change the arity of a relation after it has been set.

Persistent *reln* Primitive Axiomatic Single-valued

Persistent relations survive a clear operation on the knowledge base. There are two restrictions:

1. The only persistent classes are the built-in ones **CLASS**, **AGENT**, **RELATION**, **FIXNUM**, etc. Hence, a *Persistent* relation cannot refer to any other class.

2. A *Persistent* relation cannot be defined in terms of relations that are not *Persistent*. If `FOO` is defined by "Infer foo(?x) from bar(?x)" and `FOO` is *Persistent*, than `BAR` must also be.

Note that there is currently no error-checking for either of these.

All library routines of built-in predicates are persistent, so they do not have to be automatically reloaded after clearing the knowledge base. Note that since classes cannot be made persistent, this limits the use of typed variables in defining persistent relations.

Single-valued *reln* Primitive Axiomatic

A *Single-valued* relation is one that has only a single set of second through nth ground arguments for a ground first argument. As in Proteus, the latest assertion takes precedence over earlier ones. Single-valuedness is implemented with the JTMS.

A relation cannot be asserted to be *Single-valued* after clauses of the relation have been asserted. In this way *Single-valued* is not lazy. Similarly,

the *Single-valued* property of a relation cannot be retracted after the relation has clauses.

Specialization *special-reln general-reln* Primitive

Special-reln is a specialization of the relation *General-reln.* The two relations must have the same arity, and the argument types of *Special-reln* must be nonstrict subclasses of the argument types of *General-reln.* The built-in relations that are specializations of each other are `specialization(subclass subclass*)` and `specialization(instance instance*)`.

A proof of *Special-reln* for some arguments will suffice to prove it for *General-reln* for the same arguments.

Specificity-ordered *reln* Primitive Axiomatic

Relations that support inheritance are *Specificity-ordered.* These relations must have an arity > 0 and must have a domain that is an actual class, not just a type. Unlike Proteus, inheritance in *RAD* is not restricted to binary relations.

A relation cannot be asserted to be *Specificity-ordered* after clauses of the relation have been asserted. In this way, *Specificity-ordered* is not lazy. Similarly, the *Specificity-ordered* property of a relation cannot be retracted after the relation has clauses.

Triggerable *reln*

This probably should be a relation, but it currently is not.

Unidirectional *reln in-arg-indices* Primitive Axiomatic

Most relations in *RAD* are, by default, *multidirectional.* That is, any argument can potentially be used either for input or for output. However, some relations are either inherently or conceptually *Unidirectional*, meaning that the bindings for some arguments cannot (or should not, due to considerations of efficiency or conceptual clarity) be inferred given bindings for some others.

Unidirectional means that any query on *reln* must have ground bindings for all the arguments given in *in-arg-indices.*[1] From an implementation

[1] We could, of course, alter the semantics (or provide another relation) to make a weaker statement about the relation involved, namely, that if any of the arguments specified by *in-arg-indices* is nonground, then all arguments not in *in-arg-indices* are nonground as well. This intuitively does not seem as useful, although in such cases there would still be room for some optimization.

standpoint, the advantage of having some relations be unidirectional is that it could allow us to index instances of the relation on an argument or arguments. We could thus generate more efficient WAM/Lisp code for such relations.

Unidirectional has been defined here as a primitive relation, although most users would not actually want to concern themselves with the generation of efficient WAM code. Moreover, it would even be easier to implement as a system relation, since most of the relations for which unidirectionality is useful or necessary are predefined. In any case, we make *Unidirectional* axiomatic since the relation in question would have to be recompiled every time such an assertion went in or out (unless we had some kind of redundant compilation).

Note that *Unidirectional* is a single-valued relation. That is, for a given relation there can be only one set of "input" arguments specified.

5.3 Predefined Relations

A set of relations are already defined when *RAD* is started. Some of these predefined relations are essential to the operation of *RAD*, while others are predefined strictly for the convenience of users.

There are three important properties that the relations described below can have:

Action Some relations are *action* relations. Only a proposition beginning with an action relation can appear as a consequence of a forward rule. Intuitively, action relations are those that have some side-effect. Action relations are noted as such when they are described.

Library Most of the relations defined below are *built-in*: they are already defined when you start *RAD*, and they persist after you clear the *RAD* knowledge base. Some relations are not built-in but are instead defined in *RAD* library files that must be loaded before the relation is defined. Library relations are noted as such in the descriptions that follow, along with the the library file where they are defined.

Triggerability Most predefined relations form propositions that will not trigger a forward rule. Those that will are noted as such.

5.3.1 Type Relations

There are some relations that test the type of their arguments, succeeding if the argument is of the correct type, or failing (and backtracking) if it is not. Note that if the argument is a variable, the variable is "dereferenced" to its binding before the type checking is done.

Atom *thing*
Atom succeeds if *thing* is a symbol.

Atomic *thing*
Atomic succeeds if *thing* is either a symbol or a number.

Ground *thing*
Ground succeeds if *thing* is either atomic, or a cons whose car and cdr are ground.

Integerp *thing*
Integerp succeeds if *thing* is an integer.

Nonvar *thing*
Nonvar succeeds if *thing* is not an unbound logical variable.

Numberp *thing*
Numberp succeeds if *thing* is a number.

Var *thing*
Var succeeds if *thing* is an unbound logical variable.

5.3.2 Comparison Relations

These three relations compare their arguments.

Eq *thing1 thing2*
Eq succeeds if *thing1* and *thing2* are exactly the same object, either:

- Both lisp objects and EQ, according to lisp, or
- The same unbound logical variable.

Neq *thing1 thing2*
Neq succeeds if *thing1* and *thing2* are not exactly the same object, i.e., it succeeds exactly when *Eq* fails and vice versa.

= *thing1 thing2*
= succeeds if *thing1* and *thing2* unify.

5.3.3 Metarelations

The following relations take one or more propositions as arguments. They can be used in the antecedents of forward rules or backward rules.

And *prop1 prop2* Both arguments are clausal

And succeeds if both *prop1* and *prop2* are provable, with compatible bindings for the logical variables. Since there is an implicit *And* around the antecedents of a rule, this explicit *And* is really only useful inside an *Orr*. *And* and *Orr* can be arbitrarily nested.

Bagof *template proposition bag* *proposition* is clausal

Bagof finds all the proofs of *proposition*. For each proof, *Bagof* collects an instance of *template* which is consistent with *proposition*, and finally unifies this list with *bag*.

Excuse *proposition* *proposition* is clausal and triggerable

Excuse succeeds if *proposition* is provable via backward chaining. *Excuse* prevents any proof of *proposition* from affecting the justification of the proof in progress. Thus, no JTMS record of *proposition* will appear in any result of a proof of *Excuse*.

Known *proposition* *proposition* is clausal and triggerable

Known succeeds if *proposition* is provable via relation ground assertions, i.e., without resorting to rules, nonground assertions, or slot values.

Orr *prop1 prop2* Both arguments are clausal

Orr succeeds if either *prop1* or *prop2* is provable. Note that *RAD* tries the two propositions in order: successive proof attempts of *Orr* will cause successive proof attempts of *prop1* until failure. Then *prop2* is tried.

Provable *proposition* *proposition* is clausal

Provable succeeds if *proposition* is provable using backward chaining—from the point of view of backward chaining the proposition `(provable (foo ?x))` is identical to `(foo ?x)`. *Provable* is useful in antecedents of forward rules because it is not triggerable.

Prove *agent-name proposition* *proposition* is clausal

Prove succeeds if *proposition* is provable by the agent *agent-name*. The agent returns all possible instantiations of *proposition*, which are then used one at a time. For each such clause, an assertion that the remote agent SAID that clause is recorded.

Prove-once *agent-name proposition* *proposition* is clausal

Prove-once succeeds if *proposition* is provable by the agent *agent-name*. If so, an assertion is recorded that the remote agent SAID that instance of the proposition.

Query *agent-name proposition* *proposition* is clausal

Query succeeds if *proposition* is provable by the agent *agent-name*. The agent returns all possible instantiations of *proposition*, which are then used one at a time. For each such clause, an assertion that the remote agent SAID that clause is recorded. Unlike *Prove*, the remote agent does not remember the proofs it generates.

Query-once *agent-name proposition* *proposition* is clausal

Query-once succeeds if *proposition* is provable by the agent *agent-name*. If so, an assertion that the remote agent SAID that instance of the proposition is recorded. Unlike *Prove-once*, the remote agent does not remember the proof it generates.

Remember *proposition* *proposition* is clausal

Remember succeeds if *proposition* is provable. If *proposition* is proved using a backward rule or some assertion more general than *proposition*, then *proposition* is asserted with the proper justification as a side-effect of the *Remember* proof. Future attempts to prove *proposition* will not have to reprove it using the same method.

Said *agent-name proposition* *proposition* is clausal

Said succeeds if the agent *agent-name* said *proposition* to this agent.

Unless *proposition* *proposition* is clausal

Unless succeeds if *proposition* is unprovable, much like Prolog's *not*. Unlike *not*, it will create a nonmonotonic dependency between *proposition* and the results (if any) of the proof of *Unless*. (*Unless* is actually not implemented as a relation, but rather as a macro for the *RAD* reader. However, for all practical purposes, it can be treated as a relation.)

Was-told *agent-name proposition* *proposition* is clausal

Was-told succeeds if this agent was told *proposition* by agent *agent-name*.

5.3.4 Pathological Relations

Fail
The relation *Fail* always fails and backtracks.

True
The relation *True* always succeeds.

5.3.5 *RAD* Knowledge Base Relations

The following relations affect the state of the *RAD* knowledge base, and hence they are all action relations suitable for consequents of forward rules. They are also sometimes useful as antecedents of backward or forward rules. If used in that manner, they always succeed, and in succeeding change the *RAD* knowledge base in the manner specified for each relation.

Accept *proposition* Action; *proposition* is clausal
Accept causes *proposition* to be asserted in the *RAD* knowledge base. The justification of *proposition* is that the agent who made the accept "said so" and that the agent is reliable unless known to be unreliable.

Assert *proposition* Action; *proposition* is clausal
Assert causes *proposition* to be asserted in the *RAD* knowledge base. The justification of *proposition* is that of the proof so far, either the proof-in-progress for an *Assert* that appears as a rule antecedent, or the complete proof of the forward rule for an *Assert* that appears as a forward rule consequent.

Assert-Instance *class frames* Action
Assert-instance creates instances of the class *class*, with names given by *frames*. When it is used as a consequent of a forward rule, this action relation enables a class system to be expanded at run-time.

Assert-Subclass *class frames* Action
Assert-subclass creates subclass relationships from the superclass *class* to each of its subclasses given by *frames*. When it is used as a consequent of a forward rule, this action relation enables a class system to be refined at run-time.

Consult *pathname* Action

Consult loads a *RAD* file into the knowledge base. If the file has already been loaded, consult will not reload it. *RAD* consult files are described in more detail in Chapter 9.

Contradiction Action

Contradiction creates a contradiction and thus invokes the contradiction resolution mechanism (see Chapter 8).

Erase *proposition* Action; *proposition* is clausal

Erase causes *proposition* to be OUT. The justification of the erasure is that of the proof so far, as with *Assert*.

Kill *proposition* Action; *proposition* is clausal

Kill deletes all evidence of *proposition* from the *RAD* knowledge base.

Remove-Subclass *class frames* Action

Remove-subclass deletes existing subclass relationships from the superclass *class* to each of its subclasses given by *frames*. When it is used as a consequent of a forward rule, this action relation enables a class system to be modified at run-time.

Tell *agent proposition* Action; *proposition* is clausal

Tell causes *proposition* to be asserted in the knowledge base of the agent *agent*, as described in Chapter 2. The justification for *proposition* has an `IN`-list consisting of `reliable(SOURCE-AGENT)` and `said(`*proposition* `SOURCE-AGENT`, and an empty `OUT`-list, where `SOURCE-AGENT` is the name of the agent that executed the relation *Tell*. The `SOURCE-AGENT` maintains a justification consisting of either the proof-in-progress for a *Tell* that appears as a rule antecedent, or the complete proof of the forward rule for a *Tell* that appears as a forward rule consequent.

5.3.6 Class System Relations

The following relations are used for testing or generating classes and instances in the *RAD* class system. Each of the relations is multidirectional. For instance, one can use *subclass* to test whether **GRADER** is a direct subclass of **EMPLOYEE** (by querying `subclass(grader employee)`), to find classes that are direct subclasses of **EMPLOYEE** (via successive proofs of `subclass(?x employee)`), or to find pairs of classes such that one is a direct subclass of the other (using `subclass(?x ?y)`).

Instance *frame class* Library: builtins.rad

Instance is true if *frame* is an instance of *class*, that is if the type of *frame* is *class*.

Instance* *frame class* Library: builtins.rad

*Instance** is true if *frame* is an instance of *class*, that is if the type of *frame* is a *Subclass** of *class*, as described in Chapter 4.

Subclass *sub super* Library: builtins.rad

Subclass is true if *sub* is a direct subclass of *super*. *Subclass* is the primitive class relation built from the command **assert-subclass**, as described in Chapter 9.

Subclass* *sub super* Library: builtins.rad

*Subclass** is true if *sub* is a subclass of *super*. *Subclass** is the reflexive transitive closure of *Subclass*, as explained in Chapter 4.

5.3.7 Interface with Lisp

RAD has a fast and versatile interface with lisp, available through the following relations.

Is *thing lisp-form* Action

Is evaluates the lisp s-expression *lisp-form*, with the following caveats:

1. Lisp special forms (e.g., `setq`) cannot be used in *lisp-form*. (Instead of `setq`, use `set`.)

2. Evaluation of a *RAD* logical variable is interpreted as dereferencing. Thus if *lisp-form* is `(cons ?x a)`, `?x` is bound to `a`, and `a` is bound to 5, *Is* will evaluate the form to produce a cons of the symbol a and the number 5.

The result of evaluating the form is then unified with *thing*. If unification is unsuccessful, *Is* fails.

It is anticipated that most uses of *Is* will fall into a few stereotypical cases. Therefore a more concise syntax has been implemented within the *RAD* reader. Escapes to Lisp can be prefixed by a comma; the *RAD* reader will parse these and generate the appropriate calls to *Is*. (Refer to Chapter 9.

Do *lisp-form* Action

Do evaluates the lisp s-expression *lisp-form*. The evaluation caveats of *Is* also apply to *Do*. *Do* always succeeds, so to be useful, *lisp-form* should perform some side-effect.

5.3.8 Querying the User

Ask *prop* *prop* is clausal; Library: ask.rad

Ask takes a proposition as an argument and queries the user for an instance of this proposition (with exceptional cases described below). If the user replies positively with a valid instance, then that instance is automatically inserted in the knowledge base and *Ask* then succeeds by matching the goal with the new assertion. If not, then *Ask* fails.

For example, if information pertaining to a relation P may only be supplied directly by the user, then P may be defined by a single rule:

```
Infer P(?X ?Y) from Ask(P(?X ?Y))
```

If Q is defined by

```
Infer Q(?X ?Y) from P(?X ?Y)
```

then

```
User: ? q(a ?x)
Zeus: P(A ?X)?
User: reply zeus ''p(a 1)''
Zeus: Assertion P(A 1) accepted.

User: ? q(b ?X)
Zeus: P(B ?X)?
User: reply zeus ''''
Zeus: NO SOLUTION
```

Here, the user responds positively to the first query with the instance `P(A 1)`, and negatively to the second with an empty string.

Note that a query may be repeated, as it may have several valid responses:

```
User: ? q(a ?x)
Zeus: P(A ?X)?
User: reply zeus ''p(a 1)''
Zeus: Q(A 1)

User: next
Zeus: P(A ?X)?
User: reply zeus ''p(a 2)''
Zeus: Q(A 2)
```

In order to avoid unnecessary repetitions of queries, the system maintains a history of the queries that have already been posed to the user. If the user has already responded negatively to a query about a given goal (or any goal of which the given goal is an instance), then the query is cancelled and *Ask* simply fails:

```
User: ? q(?x ?y)
Zeus: P(?X ?Y)?
User: reply zeus ''''
Zeus: NO SOLUTION

User: ? q(a ?X))
Zeus: NO SOLUTION
```

In Chapter 9, we describe the top-level command **ask**, which allows the system builder to customize the format in which a query is posed.

Ask-once *prop* *prop* is clausal; Library: ask.rad

This relation has the same behavior as *Ask*, except that once the user has been queried about a goal, *Ask-once* will never query him about it again (regardless of the user's original response). This is useful in case it is known that a goal has only one valid instance:

```
User: Infer age(?X ?Y) from ask-once(age(?X ?Y)))
Zeus: Backward Rule AGE-1 accepted.

User: ? age(bob ?X)
Zeus: AGE(BOB ?X)?
User: reply zeus ''age(bob 35)''
```

```
Zeus: AGE(BOB 35)

User: next
Zeus: NO SOLUTION
```

5.3.9 Miscellaneous Built-In Relations

Count-proofs *proposition count* *Proposition* is clausal; Library: misc.rad
Count-proofs is true if there are exactly *count* proofs of *proposition* in the *RAD* knowledge base. Note that the JTMS status of count-proofs proof is weird: it does not depend on the status of any of the actual proofs counted.

Element *elt list* Library: misc.rad
Element is true if *elt* is an element of the list *list*. (This is the same as **member** from Prolog.)

Frules-indexed *reln rules* Library: misc.rad; System Axiomatic
Frules-indexed relates a relation *reln* to the forward rules in which it appears as an antecedent. It is *Unidirectional*, i.e., its first argument must be ground.

Host *database name* Library: dai.rad
The relation *Host* is defined for the built-in class **DATABASE**. Its value, supplied by *RAD*, is the name of the host machine for the database.

Print *format print-elements* Library: misc.rad
Attempting to prove *Print* or asserting *Print* as a consequent of a forward rule will cause output to be printed to the window of the user interface process. *format* is lisp's format directive string, while print-elements is a **list** of arguments for that string.

Reliable *agent* Library: dai.rad
The relation *Reliable* is used to provide justifications for data learned from other agents.

Unreliable *agent* Library: dai.rad
The relation *Unreliable* is used to provide justifications for data learned from other agents.

!

RAD supports the prolog control primitive cut ("!"), both in backward rules, and as described in Chapter 7 in forward rules as well. The operational semantics of cut is identical to that in Prolog, and beyond the scope of this manual.

5.4 User-Defined Relations

Corresponding to each user-defined class is a (possibly empty) set of user-defined relations associated with that class. Every relation can be defined for a unique class by means of the relation *Argument-types* (defined in Section 5.2). The frames that may have values for a given relation are the instances of the class for which it is defined, i.e., the instances of the class listed first in *Argument-types*. Thus, in the example of Figure 4.2, if relations called *Hourly-wage* and *Title* are defined for the classes **EMPLOYEE** and **FACULTY**, respectively, then the frames that may have *Hourly-wage* values are `NAT`, `HILARY`, `ANN`, and `DON`, while only `ANN` and `DON` may assume *Titles*.

The manner in which relations are inherited from their defining classes provides motivation for the construction of user-defined metaclasses, such as those of Figure 4.3. If **ANIMALIA**, **CHORDATA**, etc. had simply been defined as instances of **CLASS**, there would have been no way for them to acquire relation values. But as instances of the user-defined class **BIOLOGICAL-CLASS**, they may assume values for any relations defined for that class. If *Common-name* is among these relations, then the statement "the *Common-name* of **PROTOZOA** is `ONE-CELLED-ANIMAL`" makes sense. The partitioning of **BIOLOGICAL-CLASS** into subclasses allows the definition of relations that pertain to some biological-classes but not to all. For example, one might refer to the number of species of protozoa, but not to the number of species of amoeba, while the number of species of animalia is not practically measurable. The relation *Number-of-species*, therefore, should not be defined for the class **BIOLOGICAL-CLASS**, but rather for its subclass **PHYLUM**.

Chapter 6

Backward Inference

The primary means by which computation is carried out in *RAD* is through a goal-directed theorem-proving process called *backward inference*. This process permits goals, in the form of propositions, to be proven by a combination of unification and backward chaining.

When a proposition is presented to the *RAD* theorem prover as a *goal*, it attempts to derive an instance of it from the data in the knowledge base. An *instance* of a proposition is a second proposition that results from the first by performing some set of variable substitutions.

One way in which it might do this is to unify the goal with an assertion. The process of unification amounts to finding the most general common instance of two propositions. If a goal is unifiable with an assertion that is `IN`, then the resulting instance of the goal is returned as the result of the proof. For example, if the knowledge base of Figure 4.2 contains

```
DESCRIPTION(?X:STUDENT IDEALISTIC)
```

then the goal `DESCRIPTION(?X:EMPLOYEE ?Y)` could succeed by returning the instance `DESCRIPTION(?X:(GRADER TA) IDEALISTIC)`.

A goal may also be proved with the use of a *backward rule*. A backward rule is composed of a proposition, called its *consequent*, and one or more *antecedents*. A rule represents the belief that any instance of its consequent is true whenever any compatible instances of its antecedents are true. If a goal is unified with the consequent, then the corresponding instances of the antecedents become subgoals—recursively proving all of these subgoals completes the proof of the original goal. This process is known

as *backward chaining* or *goal-directed inference*, and is the basis of Prolog [Clocksin and Mellish 1981] and other logic programming systems.

For example, in order to derive a value for MICHAEL's *Uncle* slot, the goal UNCLE(MICHAEL ?X) may be unified with the consequent of the rule

```
Uncle-rule: Infer uncle(?X ?Y)
            from (parent(?X ?Z)
                  brother(?Z ?Y))
```

creating the subgoals PARENT(MICHAEL ?Z) and BROTHER(?Z ?Y). Suppose that the first of these is matched with the consequent of

```
Parent-rule: Infer parent(?X ?Y) from mother(?X ?Y)
```

and is thus replaced by the subgoal MOTHER(MICHAEL ?Y), which is matched with

```
Instance slot value MOTHER-7
  MOTHER(MICHAEL SUZY)
```

The second subgoal, which becomes BROTHER(SUZY ?Y), may then be derived from

```
Instance slot value BROTHER-23
  BROTHER(SUZY DAVID)
```

The instance UNCLE(MICHAEL DAVID) of the original goal is thereby proved by backward chaining.

The *RAD* backward inference mechanism is based on an extended Warren Abstract Machine emulator [Warren 1983]. The assertions and backward rules associated with a given relation, which form its *definition* with respect to backward inference, are compiled into a sequence of instructions that are interpreted by the emulator when a goal is processed. Although the user need not be aware of the workings of the compiler and the emulator, the code generated for a relation may be examined by means of the top-level command **bgrind** (Chapter 9).

Certain built-in relations (see Chapter 9) are defined directly by WAM instructions, either for efficiency or because their desired behavior cannot be represented by rules. An example of a relation that could not effectively be defined in terms of rules is the built-in relation *Unless*. This relation takes a single argument, which is a proposition, as in

```
Parent-rule-2: Infer has-child(?X)
               from (mother(?Y ?X)
                     unless(adult(?Y)))
```

When an antecedent of this type is processed, the system attempts to prove the proposition that occurs as the argument of *Unless* (under the current variable bindings). If this proof attempt fails, then the subgoal succeeds; if the proof succeeds, then the subgoal fails.

6.1 Asserting Proof Results

When a proposition proved by backward chaining is explicitly added to the knowledge base as an assertion, it receives a justification that is constructed upon examination of the proof. Thus, the proposition derived in the first example of this chapter results in

```
Instance Slot Value UNCLE-2
  UNCLE(MICHAEL DAVID)
```

which acquires a justification with IN-list (MOTHER-7 BROTHER-23 UNCLE-RULE PARENT-RULE), i.e., all the data involved in the proof, and OUT-list ().

Nonmonotonic dependencies are constructed from proofs that involve *Unless* antecedents. For example, if the proposition (HAS-CHILD SUZY) were derived from the backward rule Parent-rule-2 and Assertion MOTHER-7 above, and

```
Assertion HAS-CHILD-5
  HAS-CHILD(SUZY)
```

were derived as a result, then the IN-list of its justification would be (MOTHER-7 PARENT-RULE-2), but the justification would also reflect the dependency of the derivation on the failure to prove ADULT(MICHAEL). This would be done by making the OUT-list (ADULT-2), where

```
Assertion ADULT-2     (OUT)
  ADULT(MICHAEL)
```

is an unjustified assertion, created for the purpose of this justification (unless it already existed). If ADULT(MICHAEL) were to be asserted later, then ADULT-2 would become IN, and HAS-CHILD-5 would go OUT, as it should.

The case of an UNLESS goal with unbound variables presents a new problem. Suppose, for example, that we have a relation *Orphan* with an associated rule

```
Orphan-rule: Infer orphan(?X)
             from unless(parent(?X ?Z))
```

Then the goal ORPHAN(ANNIE) will succeed if PARENT(ANNIE ?Z) fails. In this case, the new justification for

```
Assertion ORPHAN-2:
  ORPHAN(ANNIE)
```

should contain only ORPHAN-RULE in its IN-list, but there is no assertion, general or particular, which could be placed in the OUT-list to record the nonmonotonic dependency. This problem is solved by the introduction of a new datatype:

```
Failed Goal PARENT-2     (OUT)
  PARENT(ANNIE ?Z)
```

A *failed goal* is a datum that is created only in this situation. When a proof succeeds as a result of a failure to prove a proposition that follows *Unless* in an antecedent, the proposition that failed is inserted in the knowledge base without justification as a failed goal. It represents the belief that some instance of the proposition is true. That is, any variables in a failed goal are understood to be existentially quantified. The failed goal PARENT-2 above, which represents the belief that ANNIE has some parent, would appear in the OUT-list of the justification of ORPHAN-2.

If some instance of a failed goal is asserted at any time, the system automatically creates a monotonic dependency of the failed goal on the assertion. Thus, if the assertion ORPHAN-2 were justified as described above, and

```
Assertion PARENT-3:
  PARENT(ANNIE WARBUCKS)
```

were later to become IN, then the failed goal PARENT-2 would also be forced IN and hence ORPHAN-2 would go OUT.

6.2 Proof Strategies

In the process of proving a goal by backward chaining, the goal is matched against each element of a set of assertions and rules until unification succeeds. If a resulting subgoal fails and the prover backtracks to the original goal, the matching process is resumed until another match is found.

6.2.1 Relations

It is not surprising that the ordering of the list of data to be matched against a goal may not only affect the efficiency of a proof but may actually determine whether a proposition is provable at all. Consider, for example, the relation *Not-instance**, defined by

```
Infer not-instance*(?X ?C:class)
from (instance*(?X ?C) ! fail())
```

and

```
not-instance*(?X ?C)
```

(The symbol "!" is the standard Prolog *cut* and *Fail* is a built-in relation that represents the empty relation.) This relation, which represents *X is not an instance* of class C*, behaves as desired only if the data are applied in the order indicated, with the rule preceding the assertion.

In Prolog, the order in which a set of data is created is strictly preserved by the inference system. Of course, this means that a relation must be recompiled whenever a new rule or assertion is added. One difference between an expert system built with *RAD* and most Prolog programs is that an expert system is likely to generate a large number of assertions dynamically during its execution. In order to avoid excessive recompilation, *RAD* makes a distinction between ground assertions (i.e., assertions without variables) and rules. When a ground assertion is created, it is not compiled, but instead inserted at the end of a list that is interpreted directly by the inference mechanism. In an attempt to prove a goal, the list of relevant ground assertions is always processed first, followed by the (compiled code for the) relevant rules. Nonground assertions—like backward rules—are compiled in the order in which they are asserted.

6.2.2 Multiple-Valued Relations

Different goal-matching strategies are required for relations associated with frames, i.e., specificity-ordered relations. For these, each instance slot value is attached to the frame to which it pertains. In fact, this is an important advantage of declaring a relation to be specificity-ordered: when a goal is being processed that concerns values for a fixed frame, the values for that frame only are retrieved, while other frames are ignored.

In the case of a multiple-valued relation, all relevant instance slot values are matched against a goal before any backward rules are applied. The first argument of a goal pertaining to a slot may be either a variable or a frame. If it is a frame, then the instance slot values for that frame are tested first, followed by the class slot values for all superclasses of the frame's type (in depth first order). Finally, a list of all the backward rules for the slot is traversed.

If the first argument of the goal is a variable, the strategy is complicated. In this case, each class that has a subtype in common with the type of the variable must be examined for matching values. Consider, for example, the goal `(DESCRIPTION ?X:STAFF ?Y)`, where *Description* is the multiple-valued relation that was defined for class **PERSON** in Chapter 4. In processing this goal, the g.l.b. of **STAFF** and each subclass **C** of **PERSON** must be computed. There are three possibilities:

1. The g.l.b. of **STAFF** and **C** may be `BOTTOM`. (This occurs when **C** is **FACULTY**.) In this case, **C** is ignored.

2. The g.l.b. may be **C** itself, i.e., **C** may be a subclass of **STAFF**. (This occurs when **C** is **STAFF**, **GRADER**, or **TA**.) In this case, the variable ?X may be bound to either an instance of **C** or to a variable whose type is **C**. The list of all instances of **C** is examined first. Each instance becomes the binding of ?X while ?Y is bound to each of its *Description* values. After all of these instance values have been examined, the type of ?X is coerced to the g.l.b. and the class values of **C** are examined in order.

3. The g.l.b. may be a nontrivial proper subtype of **C**. (This occurs when **C** is **PERSON**, **STUDENT**, **GRADUATE**, or **UNDERGRADUATE**.) In this case, ?X may be bound to he instances of **C**, which

are therefore ignored, but the class values of C are examined as in case 2.

After all classes and all relevant slot values have been examined, the list of backward rules is traversed as in the previous case.

6.2.3 Single-valued Relations

The strategy for satisfying a goal pertaining to a single-valued relation is somewhat simplified by the design decision that a query concerning the values of a single-valued relation for a fixed frame should not return more than one instance. Therefore, in the case in which the first argument of the goal is a frame rather than a variable, only one instance of the goal may be proved—there is no backtracking. To satisfy such a goal, the frame is first examined for an instance value that is `IN`. (There can be only one such value.) If none is found, then the system tries to derive a value using backward rules. Only if this fails is the default value for the appropriate class retried. Thus, default values, unlike class slot values for multiple-valued relations, are used only as a last resort.

If the first argument of the goal is a variable, then the prover begins by binding this variable to some instance of its type, and attempts to satisfy the goal according to the procedure described in the preceding paragraph. If this fails, or if an additional solution is sought, then the variable is bound to another frame, until it has been bound to all possible frames. This accounts for the result of the query in Chapter 4 for all provable instances of `(NATIONALITY ?X:PERSON ?Y)`. Note, in particular, that although the proposition `(NATIONALITY ?X:STUDENT CHINESE)` has been asserted as a default value, and this value remains `IN`, this proposition is not retrieved as a provable instance of the goal. The reason for this is the difference between default values of single-valued relations and class values of multiple-valued relations—a default value for a class is not necessarily inherited by the instances of the class. Therefore, the proposition `(NATIONALITY ?X:STUDENT CHINESE)` should not be considered a general assertion. A default value should be used only in computing a value for a particular object, and not for a class of objects.

Chapter 7

Forward Inference

A backward rule has effect only when it is relevant to a goal being processed by the system. The insertion of a backward rule, therefore, affects only the implicit informational content of the knowledge base, without causing new assertions to be added explicitly. Consider, for example, the backward rule PARENT-RULE of Chapter 6, which states that all mothers are parents.

```
Parent-rule: Infer parent(?X ?Y) from mother(?X ?Y)
```

In the presence of the assertion

```
Instance slot value MOTHER-7
  MOTHER(MICHAEL SUZY)
```

this rule enlarges the implicit knowledge base to include the proposition PARENT(SUZY MICHAEL), without actually creating a new assertion.

The same logical implication expressed by PARENT-RULE could alternatively be represented as the *forward rule*:

```
Parent-rule-3: If mother(?X ?Y)
               then parent(?X ?Y)
```

While the two rules are logically equivalent, they are used quite differently. The forward rule takes effect not when a goal matches its consequent PARENT(?X ?Y), but rather when an assertion matches its antecedent MOTHER(?X ?Y). In this event (assuming the new assertion is IN), another

assertion, representing the corresponding instance of PARENT(?X ?Y), is automatically added to the knowledge base. The PARENT assertion is then justified by the MOTHER assertion and the rule. Thus, when the match between MOTHER-7 and PARENT-RULE-3 is discovered, the result is the new datum

```
Instance Slot Value PARENT-4
  PARENT(MICHAEL SUZY)
```

which is justified with an IN-list (MOTHER-7 PARENT-RULE-3) and OUT-list NIL. This process is known as *forward chaining* or *data-directed inference.*

7.1 Overview of Forward Chaining

A forward rule may have any number of antecedents and consequents. Antecedents have the same form as those of backward rules. When a new assertion is unified with an antecedent of a forward rule, the set of remaining antecedents is presented to the backward inference system as goals. For each simultaneous proof of these goals, a *firing* of the rule occurs, i.e., its consequents are processed.

Consequents of a forward rule are *actions* and involve one of the action relations, such as *Assert*, *Consult*, *Do*, *Erase*, *Is*, *Kill*, *Print*, or *Tell.* Intuitively, these are all relations that have some side effect—either to the *RAD* knowledge base, to another agent, to the user interface, or to the Lisp interpreter. All consequents that are not explicitly actions are considered to be *Assert* actions. Thus, if if you want to assert some proposition as a consequent of a forward rule, you need only write the proposition itself; the *Assert* relation is implicit.

When a rule is fired, each of the consequences is executed. For assertions, execution means that an instance of the proposition to be asserted is actually asserted. The justification for this assertion is constructed from the data involved in the derivation of the antecedents, as described in Chapter 6. For other sorts of action propositions, the corresponding action is executed. The actions are described with the rest of the built-in predicates in Chapter 9.

Suppose, for example, that the knowledge base contains

```
Rule-1: If (patient(?X)
            should-take(?X ?Y))
```

```
        then (under-treatment(?X)
              print("Prescription for ~A:  ~A" (?X ?Y)))
```

when

```
Assertion PATIENT-2
  PATIENT(BILL)
```

is added. The first antecedent of PATIENT-2 is matched with RULE-1, triggering an attempt to prove SHOULD-TAKE(BILL ?Y). Suppose that the instance SHOULD-TAKE(BILL ASPIRIN) is derived. Then after

```
Assertion SHOULD-TAKE-1
  SHOULD-TAKE(BILL ASPIRIN)
```

is added to the knowledge base, the rule RULE-1 fires:

```
Assertion UNDER-TREATMENT-1
  UNDER-TREATMENT(BILL)
```

is added, justified by RULE-1, PATIENT-2, and SHOULD-TAKE-1, and

```
Prescription for BILL:  ASPIRIN
```

is printed.

Some thought is required in determining whether a given implication should be represented as a forward rule or a backward rule. A backward rule offers the advantage of increasing the inherent knowledge of a system without incurring the expense (in both time and space) of creating new assertions. It may be necessary, however, for this knowledge to be represented explicitly in order for it to take some desired effect.

Suppose, for example, that the assertion

```
Assertion ORPHAN-3     (IN)
  ORPHAN(GEORGE)
```

is added as a result of the rule ORPHAN-1 of Chapter 6:

```
Orphan-rule: Infer orphan(?X)
             from unless(parent(?X ?Z))
```

Then the OUT-list of its justification contains a datum corresponding to the last antecedent of the rule,

```
Failed Goal PARENT-5    (OUT)
  PARENT(GEORGE ?Z)
```

which was unprovable at the time ORPHAN-3 was created. Suppose that

```
Assertion MOTHER-12
  MOTHER(GEORGE MARY)
```

were asserted later. It would then be desirable for PARENT-5 to come IN and for ORPHAN-3 to go OUT. The backward rule PARENT-1, however, could not cause this to occur. Although an instance of PARENT-5 would become provable, that instance would not be discovered. It would probably be preferable in this case to code the rule in the form of the forward rule PARENT-RULE-3 instead. This would produce a new assertion

```
Assertion PARENT-6
  PARENT(GEORGE MARY)
```

on which PARENT-5 would become monotonically dependent, and ORPHAN-3 would go OUT as desired.

7.2 Forward Chaining in More Detail

The above description of forward chaining is accurate but not very complete. For some *RAD* programs, it is important to know not only *what* happens but exactly *when* things happen.

When a unit clause is asserted, the new assertion is placed on an agenda. As long as the agenda is nonempty, *RAD* does not return to the user. Instead elements are popped from the agenda and executed—used to trigger forward rules. (By default the agenda is LIFO—a stack—but a user can control which element is picked from the agenda with the function `pick-cselt`.)

When an agenda element is picked, *RAD* looks for an antecedent from a forward rule to unify with it. If more than one such antecedent will unify, the antecedent from the rule with the highest *priority* is chosen. Priorities are integers from 1–99 associated with forward rules, and a rule has priority 50 by default. A forward rule with a different priority is created by enclosing a two-digit number in square brackets before the **If** clause of the rule:

```
Rule-2: [40] If (foo(?X ?Y)
         bar(?Y ?Z)
         baz(?Z ?X))
     then fbb(?X ?Z)
```

When an antecedent from a rule has been chosen, *RAD* attempts to unify the agenda element with the antecedent. If the unification fails, *RAD* looks for another antecedent, choosing antecedent from rules with higher priorities over antecedents from rules with lower priorities.

If the unification succeeds, *RAD* attempts to find proofs of the conjunction of the remaining antecedents in the context of the variables bound in the successful unification. For example, if the agenda element `BAR(12 ?X)` triggers the above rule, then *RAD* attempts to prove `AND(FOO(?x 12) BAZ(?z ?x))`. These proofs are found by standard *RAD* backward chaining as in Chapter 6. In particular, no `OUT` rules or clauses will be used in any proof of the remaining antecedents.

For each proof of the antecedents, *RAD* executes the consequences once. This means that a single triggering of a single forward rule can lead to multiple firings of that rule, if there happen to be multiple proofs of the remaining antecedents in the knowledge base. Since the remaining antecedents are just a conjunction *RAD* propositions, a well placed cut ("!") among the antecedents can served to prune the proofs. For example, the rule

```
Rule-3: [40] If (foo(?X ?Y)
         bar(?Y ?Z)
         baz(?Z ?X) !)
     then fbb(?X ?Z)
```

will allow (at most) one firing of the rule for every triggering of the antecedents.

Some of the consequents of the fired rule may assert new facts. These assertions are added to the *RAD* knowledge base, and if they are unit clauses, they are added to the agenda. Nothing new is picked from the agenda until the currently active agenda element has triggered all the rules it can.

After finishing with one antecedent, *RAD* looks for another antecedent that will unify with the current agenda element, continuing the process of firing forward rules. Only when there are no more antecedents to unify will another element be picked from the agenda.

Some antecedents trigger their rules by unifying their first argument rather than the whole proposition with the active agenda element. For example, if `KNOWN(FOO(?X))` (described in more detail in Chapter 5) is an antecedent in a rule, it can be triggered by `FOO(?X:NUMBER)`. This is consistent with *Known*'s semantics: *Known* is true if there is a ground `IN` instance of its first argument in the *RAD* knowledge base—i.e., if its first argument is provable without resorting to nonground clauses or backward rules. Thus it makes sense for *Known* to be triggerable by unification of the agenda element with *Known*'s first argument.

Some propositions (e.g., those involving the relation *Bagof*) cannot be triggered by an agenda element. The special predicates that are not triggerable are noted in Chapter 5.

Chapter 8

Contradiction Resolution

8.1 Introduction

RAD contradiction resolution performs a restricted sort of abductive inference [Pierce]. A *RAD* contradiction represents a user-designated conflicting set of beliefs. Given a contradiction, *RAD* attempts to hypothesize a set of propositions that, if believed, would cause disbelief in at least one element of the conflict set. Further, *RAD* attempts to create a consistent justification for the hypothesized propositions. When successful in doing so, *RAD* is said to have resolved the contradiction. In terms of the JTMS, contradictions are resolved by retracting some set of assumptions in the foundations of the JTMS. An *assumption* is a datum that is currently believed (`IN`) based on a valid justification with a nonempty `OUT`-list. An assumption is retracted if an elective in the `OUT`-list of the assumption receives a valid justification. *RAD* contradiction resolution involves attempts to disbelieve assumptions and to find reasons to believe electives.

RAD contradictions are instances of a special kind of datum. They may have only a single justification and the JTMS will always attempt to find a stable, well-founded knowledge base state of support-statuses in which all contradictions are `OUT`. If none can be found, the Contradiction Resolution Mechanism (CRM) will attempt to produce a set of electives and associated new justifications which will make such a state possible. If it is successful, the contradiction is `OUT` and hence resolved. If the CRM fails, the contradiction is added to a set of unresolved contradictions that remain `IN`. Resolution of

these is attempted whenever they are consequences of data whose support status is altered by the JTMS. The *RAD* CRM implements the extended dependency-directed backtracking algorithm described in [Petrie 1987].

The semantics of a *RAD* contradiction is that it represents an undesirable situation, indicated by a combination of belief of the data in the `IN`-list and lack of belief in the data in the `OUT`-list of the justification for the contradiction. Although the term "contradiction" is used for historical reasons, in a justification-based TMS such as used by *RAD*, contradictions have no logical import and only denote a user-defined problem. There is no requirement in *RAD* that if an assertion and its negation are both `IN`, then there is a contradiction. In fact, there is no definition of "negation" in *RAD*. *RAD* does not try to prove the logical consistency of its knowledge base. A user must explicitly specify the conditions that *RAD* will recognize as a contradiction in the knowledge base.

The user can specify these conditions for a contradiction either interactively or via a rule. If a forward rule has the assertion `CONTRADICTION()` as its consequent, then when that rule fires, a contradiction is inserted into the knowledge base with the same justification as any other consequent of that forward rule. A user can contradict a conjunction of beliefs by using their names as arguments to the top-level command **contradict**. The last answer supplied by a top-level query can be contradicted by the command **wrong**.

The relations, *Fix*, *Prefer*, and *Defeat* enable a user to provide domain knowledge to guide contradiction resolution. These relations are specially recognized by the *RAD* CRM. They represent explicit knowledge about the use of other assertions in an inference procedure, and so represent a metalevel of knowledge. As a result, they are called "metarelations".

8.2 The Contradiction Resolution Process

8.2.1 The FIX Phase

The CRM is based on a simple notion of defeasible reasoning. As a simplification (see *Defeat* below for a complication), a valid justification is *defeasible* if it has a nonempty `OUT`-list or if every valid element of the justification set of some element of its `IN`-list is defeasible. We say that an assertion is defeasible if every valid justification supporting it is defeasible. The classi-

cal definition of an assumption in a JTMS is a special case of a defeasible assertion. This more general notion of defeasibility suggests a recursive procedure. Given any culprit to be defeated, i.e., made `OUT`, the procedure is to attempt to defeat all of its valid justifications. For each justification, this is done by attempting to make some element of its `OUT`-list `IN` (namely an elective) or by taking one of the elements of its `IN`-list as a new culprit to defeat. Ultimately, the result will be an AND of assertions (electives) to be made `IN`. Obviously, there will be many possibilities. The purpose of the metarelations is to suggest guidance for this search.

The relation *Fix* takes three arguments: the target, the fix-culprit, and the fix-elective. A *Fix* is defined for a particular potential contradiction if the *Fix* has a target that will unify with an `IN` supporter of that contradiction should it be created. The purpose of such a *Fix* is to identify some particular assumption (the fix-culprit) that is likely to be in the foundations of the contradiction and, if so, for which a retraction attempt should be made. The fix-elective identifies a possible elective for that assumption. That is, the *Fix* may so suggest a possible way of retracting the assumption.[1]

The CRM begins with a datum, called the culprit, to be made `OUT`. Initially, the culprit is the contradiction. First, an attempt is made to use domain knowledge to make the culprit `OUT` by using a *Fix* assertion to invalidate one of the `IN` supporters of the culprit. Each of the `IN` supporters in turn becomes a culprit for which there may be a *Fix*. An available *Fix* for a culprit is obtained when the culprit proposition unifies with the target of some provable *Fix*. A qualified *Fix* is derived from the available *Fix* when its fix-culprit unifies with the proposition of some datum in the foundations of the culprit (the target of the *Fix*). If some element of the `OUT`-list of the supporting justification of the fix-culprit unifies with the fix-elective of the qualified *Fix*, then the *Fix* is applicable and that element becomes the elective to be made `IN` by the CRM (making a datum `IN` is described in Section 8.2.2).

An example of a *Fix* assertion might be

```
Fix(Conclusion(?Patient ?Degree ?Meas ?Chem)
    Symptom(?Patient ?Symptom)
    Mistaken-observation(?Patient ?Symptom))
```

[1] Any of these arguments may be left unspecified. However, the more specificity, the more useful *fixes* will be. An assertion of `Fix(?X ?Y ?Z)` would not be useful.

This *Fix* asserts that if any conclusion becomes a culprit to be made OUT, and the conclusion depends on belief in some symptom, then a mistaken observation of that symptom may retract belief in it. That mistaken observation will only become an elective if it is either already in the OUT-list of the supporting justification of the symptom, or can be placed there by a *Defeat* assertion.

A *Defeat* assertion provides a way of inserting an *Unless* condition retroactively. An *Unless* condition in a rule indicates a non-monotonic condition. One way of avoiding performing the computation involved in testing this condition is to use the *Known* and *Unless* relations in conjunction. However, this will check to see if the condition is asserted in the database and will invalidate the result of the rule firing if the condition is ever asserted. In contrast, a *Defeat* assertion represents a condition that will never even be considered unless the result of the rule firing becomes implicated in a contradiction.

The *Defeat* metarelation also takes three arguments. The first of these is the defeat-culprit, the second is the in-supporter, and the last is the defeat-elective. A *Defeat* asserts that the defeat-elective may be added to the OUT-list of each justification of the defeat-culprit that contains the in-supporter in its IN-list. Thus the in-supporter is a way of distinguishing among justifications, if needed. If the *Defeat* is meant to apply to justifications with empty IN-lists (for example, premise justifications), the word none may be given as the in-supporter. If any nonempty IN-list should apply, a variable, such as ?X, can be used for the in-supporter. In general, though, a *Defeat* represents an argument against a particular reason for believing the defeat-culprit. If the *Defeat* is meant to apply to a contradiction justification, the defeat-culprit should be the term contradiction. In this case, the in-supporter should unify with one of the antecedents of the rule that produced the contradiction and the defeat-elective will be added to the OUT-list of the justification of the contradiction. An example of a *Defeat* might be

```
Defeat(Symptom(?Patient ?Symptom)
       none
       Mistaken-observation(?Patient ?Symptom))
```

To illustrate how this *Defeat* and the previous *Fix* could be used, consider a conclusion of a high sodium concentration for a patient named Jane, generated by the following assertion and rules:

```
Symptom(Jane dehydrated)
```

```
If Symptom(?Patient dehydrated)
then Conclusion(?Patient low amt H2O)

If (Conclusion(?Patient low amt H2O)
    Remember(Normal(?Patient amt Na)))
then Conclusion(?Patient high conc Na)

Infer Normal(?Patient amt Na)
from Unless(Result(?Patient low amt Na))
```

If the final conclusion of a high sodium concentration for Jane were to support a contradiction, belief in this conclusion could be retracted by hypothesizing that a mistaken observation was made about the symptom which supported this conclusion. The *Fix* focuses on this hypothesis and the *Defeat* makes it applicable. Notice that none of the rules includes an *Unless* clause concerning mistaken observations.[2] The *Defeat* assertion allows the assertion `Mistaken-observation(Jane dehydrated))` to be introduced into the knowledge base and into the `OUT`-list of the assertion of the symptom during contradiction resolution. This still does not resolve the contradiction. It still remains to make the new assertion `IN` to resolve the contradiction (described in Section 8.2.2).

There may be more than one *Fix* that is applicable to a culprit. One of the uses of the metarelation *Prefer* is to allow the user to specify domain knowledge to decide between *Fix*es. Available *Fix*es are ranked on the basis of their fix-electives. An alternative would be to consider the fix-culprits. But the reasoning used is that, ultimately, reasons for distinguishing between assumptions rest on reasons for belief or lack of belief in their electives. It is the electives that compete for valid justifications from the contradiction resolution process. Thus, the electives should be ranked with respect to each other in terms of preference of attempted justification. *RAD* contradiction resolution uses a very general mechanism of preferences to do so.

[2]The use of the *Unless* clause in these rules is worth explaining. The second forward rule has an antecedent that asserts that the patient's amount of sodium is normal. This is proven via the backward rule due to the lack of any reason to believe sodium level is low. By using a backward rule and the relation *Remember*, an assumption of normality is introduced into the database. This is the standard way to achieve a "positive" assumption in a *RAD* database.

One datum is preferred to a second if *Prefer* is the relation symbol of a provable assertion for which the first and second arguments unify with the first and second data, respectively. A preference is such a provable assertion. The semantics is that belief in the first argument is preferred to belief in the second for the purposes of the belief revision being performed by contradiction resolution. One belief is dominated by another if it is provable that the latter is preferred over the former. The CRM will first choose a *Fix*, the fix-elective of which is not dominated by that of another. As an example, suppose we add the following rules and assertions to those above:

```
If (Lab-test(?Patient low ?Meas ?Chem)
    Conclusion(?Patient high ?Meas ?Chem))
then contradiction()

Lab-test(Jane low conc na)

Fix(Lab-test(?Patient ?Degree ?Meas ?Chem)
    Lab-test(?Patient ?Degree ?Meas ?Chem)
    Test-error(?Patient ?Degree ?Meas ?Chem))

Defeat(Lab-test(?Patient ?Degree ?Meas ?Chem)
       none
       Test-error(?Patient ?Degree ?Meas ?Chem))
```

The first rule and assertion will cause a contradiction since the symptoms indicate that Jane should have a high concentration of sodium, but the lab-test shows just the reverse. Now two *Fix*es are applicable in this case. A doctor will generally recheck the symptoms before he rechecks the test because it is cheap and easy to do so. This domain knowledge can be represented by the assertion

```
Prefer(Mistaken-observation(?Patient ?Symptom)
       Test-error(?Patient ?Degree ?Meas ?Chem))
```

which asserts that belief in a mistaken observation is preferable to that in a lab test error for the purpose of contradiction resolution. Notice also that this

preference could have been provable. For instance, this preference could rest on a default assumption that the patient is not being remotely diagnosed, in which case it is conceivable that rechecking the symptoms might be expensive. Such knowledge could be represented by substituting the following more specific rule for the above preference:

```
Infer Prefer(Mistaken-observation(?Patient ?Symptom)
      Test-error(?Patient ?Degree ?Meas ?Chem))
from Unless(remote-patient(?Patient))
```

It is sufficiently important to mention again that *Fix*es and *Defeat*s can be similarly qualified by the antecedents of rules. Also, no single argument of any of the metarelations need be even partially instantiated. For instance, the assertion

```
fix(conclusion(?X ?Y ?Z ?U) symptom(?V ?W) ?T)
```

would cause an attempt to retract any supporting assertion of a symptom with any elective possible. Finally, we note that the same preferences used to choose the order in which *Fix*es are tried may also be used to guide the dependency-directed backtracking of [Petrie 1987] as explained in the next section.

The *Fix* operation may be summarized as follows. When a contradiction becomes `IN`, before dependency-directed backtracking is used to find an elective, the CRM will look for a *Fix*. The CRM will find all of the available *Fix*es for the `IN` supporters of the contradiction. If there are none, then dependency-directed backtracking is used as described below. Given a set of available *Fix*es, a best *Fix* will be chosen such that its elective is not dominated by that of another applicable *Fix*. In the above example, the mistaken-observation *Fix* would be tried prior to trial of the test-error *Fix*, since the latter is dominated by the former.

When a *Fix* is chosen for a culprit, an attempt is made to qualify the *Fix* and make it applicable. This entails a search of the foundations of the culprit. First, the culprit itself is examined to see if it unifies with the applicable fix-culprit. This is why the first two arguments of a *Fix* may be identical. Failing a qualification at this level, the `IN` supporters of the culprit are examined. An attempt is made to find new *Fix*es which are applicable to these supporters. A new best *Fix* is chosen at this time. The target of the best *Fix* becomes

the culprit. In the case that an IN supporter becomes the new culprit, the search begins at that datum. Otherwise, the search continues where it was interrupted by the attempt to collect new *Fix*es.

Given that the qualification search proceeds to the IN supporters of some datum, an attempt is made to qualify the best *Fix* for the culprit on each of the supporters. If the *Fix* becomes qualified, then an attempt is made to make the *Fix* applicable and make IN the resulting fix-elective. If the qualified *Fix* is not applicable or the fix-elective cannot be made IN, then the *Fix* is removed from further consideration and another best *Fix* is chosen. A *Fix* is also discarded and a new best *Fix* chosen when the foundation search of a culprit is complete and the *Fix* has not been successful.

8.2.2 Dependency-Directed Backtracking

If no *Fix* can be used to make a culprit OUT, dependency-directed backtracking is invoked. Given a culprit to make OUT via DDB, the CRM attempts to either determine that it is already invalid or invalidate each of its justifications. A justification is invalidated by making some element (elective) of its OUT-list IN or (culprit) of its IN-list OUT. The former is tried first. Preferences are used to select the elective or culprit to pursue first. An elective is chosen that is not dominated by any other element in the OUT-list. A culprit is chosen that is not preferable to any other in the IN-list. If the CRM is in interactive mode, the user will be queried to confirm this choice. The candidate electives or culprits will be enumerated. The user may respond with a carriage return only if the first choice is acceptable. Otherwise, one of the other candidates may be selected by supplying its number.

An elective is IN already if it has already been made IN or currently has a support status of IN and is not in the transitive closure of the consequences of data for which the CRM has constructed new consequences. If an elective is not already IN, the CRM will first attempt to make IN an elective by proving it. If the proof is successful, a justification for the elective is provided from the proof. If the elective has a relation symbol for which a query schemata has been defined, the user will be queried as to the belief in the elective. An answer of "yes" gives the elective a premise justification. An answer of "no" indicates that the datum is not suitable as an elective. An answer of "maybe" causes the elective to be *abductively* justified as described in [Petrie 1987]. This new justification has the semantics that belief in the elective is assumed

in order to resolve the contradiction.

The elective will also receive such a justification if it is provable that belief in the elective is preferable to belief in the contradiction; the first argument of such a preference unifies with the elective and the second is the relation *Contradiction.* In our example so far, the following preferences would be sufficient to allow the CRM to make `IN` the various possible electives:

```
Prefer(Mistaken-observation(?Patient ?Symptom) contradiction())
Prefer(Test-error(?Patient ?Degree ?Meas ?Chem) contradiction())
Prefer(Result(?Patient low amt ?Chem) contradiction())
```

Abductive justifications are constructed so as to become invalid later if belief in the elective is no longer necessary to resolve the contradiction, as they represent a higher level assumption that belief in the elective is so necessary. It may not be possible to ensure this property for a particular elective because, as described in [Petrie 1987], the justification for an elective may produce an odd loop. In such a case, *RAD* will query the user as to whether the offending assertion may be safely omitted from the constructed justification. (This should be a relatively uncommon occurrence.) In addition, the elective is not permitted to receive the constructed justification if it would cause a previous contradiction to become `IN`.

The metarelation *Support* may be used to construct a justification for the elective. The elective is supported if *Support* is the relation symbol of a provable assertion with a single argument which unifies with the elective. In this case, the proven assertion will be added to the knowledge base with a justification based on its proof, and will be included in the `IN`-list of a new justification constructed for the elective. The `OUT`-list of this justification will contain the contradiction. In an intuitive sense, *Support* is the inverse of *Defeat.* The latter is used to argue against reason for belief in a culprit. *Support* is used to argue for belief in an elective. In terms of default reasoning, *Support* provides a way to indicate additional computation that should be performed to test the validity of an assumption. Initially, belief in the culprit may have been justified by failure of some limited attempt to prove the elective. If this culprit is later implicated in a contradiction, *Support* can be used to generate additional proof attempts. If the elective receives a *Support* justification, its semantics are that the elective received new support in order to resolve a contradiction.

If an elective cannot be proven and a new justification cannot be constructed for it, the CRM will attempt to validate one of the justifications of the elective, if any, by recursively making all of the data in the OUT-list OUT and in the IN-list IN. When trying to invalidate a justification and failing to make some elective from the OUT-list IN, the CRM will attempt to add elements to the OUT-list by proving any applicable *Defeat* assertions. The added elements will also become candidate electives. If no elective can be made IN, the CRM will pick some element of the IN-list as a new culprit to make OUT. No element will be chosen that is preferred to another.

As an example of this operation, suppose that, instead of the preferences above, the following query schema were defined for the example (see Chapter 9 for a description of ask):

```
ask((result(?x low ?y ?z)
      "Is the ~a of ~a for ~a low?"
      (?y ?z ?x) () t ())
    (mistaken-observation(?x ?y)
      "Were you mistaken in observing that ~a seemed ~a ?"
      (?x ?y) () t ())
    (test-error ?x ?y ?z ?u)
      "Was the test showing ~a to have a ~a ~a of ~a in error?"
      (?x ?y ?z ?u) () t ())
)
```

Then when Jane's lab-test results caused a contradiction, given the initial simple preference of belief in mistaken observations over that of lab tests, the CRM would first apply the *Fix* about the former, discover that there was a *Defeat* which made it applicable, and then try to prove the mistaken observation. There being no other way for the proof to succeed, and a query schemata having been defined for the assertion, the user would be queried

```
Were you mistaken in observing that JANE seemed DEHYDRATED?
```

If the user answered "no", the next *Fix* would be applied resulting in the query

```
Was the test showing JANE to have a LOW CONC of NA) in error?
```

A "no" answer to this query would exhaust the possible *Fixes* and cause dependency-directed backtracking to be attempted.

The supporters of the contradiction in this instance are the result of the lab test and the conclusion about high sodium. The OUT-list being empty, the CRM would attempt to make one of these assertions OUT. Since we already know that the lab test was not in error, the CRM would not be able to defeat the result of the lab test.

However, the conclusion of a high concentration of sodium rests upon an assumption of a normal amount of sodium. This assumption would be retraction if it could be believed that there was a low amount of sodium. Although there are no backward chaining rules with which to prove a low amount of sodium, there is a query schemata defined for this assertion. So the user would be queried:

```
Is the AMT of NA for JANE low?
```

An answer of either "yes" or "maybe" would resolve the contradiction. The "maybe" answer would produce the abductive justification that the preference above would. An answer of "no" would result in an unresolvable contradiction since there are no alternative electives except for the possibility of a mistaken observation, which has been eliminated by a previous query.

8.3 User Hints

We now give some useful hints for the use of contradiction resolution.

8.3.1 General Assertions

If the top-level command **wrong** is used after a query about the inherited value of a multiple-valued slot for a frame, contradiction resolution will attempt to invalidate the justifications for the class slot value. This is also true for general assertions using relations. The semantics are that belief in the particular value for the slot rests on an assertion, the class slot value, that all instances of the class have that value for the slot. Contradiction of that value for one instance of that class entails contradiction of the more universal assertion that all instances have that value.

This is almost always a stronger contradiction than intended. In *RAD*, general assertions may not be defeated. As an example, suppose that we have a class of **BANKER** with a class value slot of `wears(?X:banker watch)` where *Wears* is a multiple-valued slot. If we want to contradict that a particular banker wears a watch, and the class value slot had a premise justification, resolution of the contradiction might be attempted with a *Defeat* such as:

```
defeat(wears(?X:banker watch)
       none
       there-exist(timeless-banker))
```

Of course, there must be a way to prove such a defeat-elective. One way would be to provide a rule such as:

```
Infer there-exist(timeless-banker)
from (instance*(banker ?X)
      timeless(?X))
```

Such a rule would attempt to prove that each banker was timeless (rules or a query schemata must be given to allow such a proof). If it were able to do so for one, then all bankers would lose their watches. *RAD* does not permit this: the *Defeat* will essentially ignored.

If the user anticipates that such a class value slot for a multiple-valued slot might not hold for all instances of the class, but only represents a typical value, it is better to forego inheritance and make the slot values individual assumptions with a rule such as:

```
If (instance*(banker ?X)
    unless(timeless(?X))
then
   wears(?X watch)
```

This rule would allow individual bankers to lack watches without affecting others.

8.3.2 Retractable Default Assumptions

Default assumptions are those that rest on the lack of belief in assertions that are neither sufficiently important nor likely to make all attempts to find

a reason to believe them without further cause. The last rule of the previous section is intended to implement the default assumption that bankers typically wear watches. If the assertion that a banker is "timeless" is provable by use of a query schemata, then the user will be queried as to whether each banker is timeless. This is not what we desire in a default assumption. Rather we want to assume that a banker is not timeless unless we have specific reason to believe so. But if insufficient domain knowledge is supplied to justify the elective (typically an *Unless* condition) during contradiction resolution, then the assumption is not retractable.

There are three methods in *RAD* for deferring proof attempts on an elective in order to retract an assumption. The simplest is to use *Known* together with *Unless* to avoid any attempt to prove the *Unless* condition except for checking for a ground instance of the condition already in the database. The user may supply backward chaining rules which infer the condition, but they will not be invoked until contradiction resolution time.

The user may want to make limited proof attempts on the *Unless* condition. If so, *Known* is not used and the initial proof attempts can be represented by ordinary *RAD* backward chaining rules concluding the *Unless* condition. Further proof attempts which should be made only at contradiction resolution time can be made by writing *RAD* backward chaining rules which conclude the *Support* metarelation on the *Unless* condition. For example, if the assertion

```
support(timeless(?X))
```

were provable, then the CRM would be able to defeat the assertion that some particular banker wears a watch. This *Support* may itself have been simply asserted, proven by a user query, or proven by more complex inferencing.

The other method of deferring proof attempts on an *Unless* condition is to omit the *Unless* clause and defer introduction of an elective in the knowledge base with *Defeats*. In this case, assumptions are created when needed by modification of `OUT`-lists. However, this only occurs during contradiction resolution. Prior to a *Defeat* being used, the assertion of the defeat-elective will not cause the defeat-culprit to go `OUT` since it has not yet been made an assumption resting on lack of belief in that particular elective. Thus *Defeats* should be used for rare defaults not usually considered: this introduces an element of heuristic inconsistency into the database.

Finally, electives need neither be proven nor *Supported* but only assumed if it is provable that they are preferable to a contradiction. For example, if the assertion

```
Prefer(timeless(?X) contradiction)
```

were provable, the forward rule would also represent a retractable assumption. This uses of *Prefer* permits the CRM to generate an *abductive* justification as described in [Petrie 1987]. The general idea is that *reductio ad absurdem* reasoning is used. That a banker is timeless will be believed because it will cause a contradiction not to believe it.

8.3.3 Preferences

Preferences are very general. Even though they are non-numeric, this does not mean that they may not be based on numbers. For instance, the following rule decides between two assertions based on their associated costs:

```
Infer Prefer(auto(?X) auto(?Y))
from (cost(?X ?Cost1)
      cost(?Y ?Cost2)
      ,<(?Cost1 ?Cost2))
```

As another example of generality, it is easy to express the knowledge that any element of some one set is preferable to that of another. Suppose that `bigger(set-1 set-2)` establishes a relationship between two sets. Then we may prefer elements of one set to another, based on that relationship with the rule:[3]

```
Infer Prefer(?X ?Y)
from (element(?X ?U)
      bigger(?U ?V)
      element(?Y ?V))
```

It is also easy to prefer one kind of assertion over any other. Suppose that assertions with relation *Boy* were to be preferred over any other. This would then be accomplished by asserting:

[3]Warning: Transitivity is not guaranteed by such rules.

```
Prefer(boy(?X) ?Y)
```

When used to determine a choice between *Fixes*, it is important to remember that the initial choice of which *Fix* to pursue is based on the applicable fix-electives: as they have been instantiated by unification with the target. For instance, suppose we have:

```
Infer Prefer(lies(?X) jokes(?X))
from salesman(?X)

Fix(speaks(?X) known(?Y) jokes(?Z))

Fix(speaks(?X) known(?Y) lies(?Z))
```

If these *Fixes* were used to resolve a contradiction involving, say, the assertion `speaks(Polly)`, the preference could not be used to decide between the two *Fixes*. the value of `?Z` would be unbound at the time the choice of *Fixes* was to be made. This can be solved instead by concluding the preference which instantiates the possible liar or joker:

```
Infer Prefer(lies(?X) jokes(?X))
from (says(?X ?Y)
      unless(clown(?X))
      salesman(?X))
```

Finally, when trying to invalidate a justification of a culprit there is no way to express a preference for pursuing a particular element of the `IN`-list rather than one of the `OUT`-list. All elements of an `OUT`-list are chosen as electives and must fail before elements of the `IN`-list are examined. Preferences are only applicable in choosing between elements of an `OUT`-list or between elements of an `IN`-list. If there is another choice to be made, the user should restructure the knowledge representation.

8.4 Final Caveats

During contradiction resolution, the user may answer "yes-no" queries with "maybe". This has exactly the same effect as a "no" answer unless the query is about the belief in the elective that the CRM is currently trying to

prove. In that event, the elective receives a valid justification constructed by the CRM. This does not happen when the user answers "maybe" to a query about a goal generated by the attempted proof of the elective. In any case, if the elective cannot be proved and there is no query schemata for it, contradiction resolution will fail to use it to resolve the contradiction.

In general, the knowledge-guided contradiction resolution described here is a novel and complex method of computing. For the end-users of expert systems, it should provide additional functionality. The ability to disagree with a computer seems highly desirable. However, the novelty, complexity, and added functionality of this experimental computation can be expected to make the job of the knowledge engineer even more difficult. The interested *RAD* user is encouraged to try examples from this report and to experiment extensively before committing to development of an application. In any case, simple applications of the CRM, avoiding the full complexity and power of the system, are feasible.

Chapter 9

Syntax and the User Interface

The user interface is an interpreter that allows the user to enter commands, which are then executed by *RAD*. These commands allow the user to create, use, and modify knowledge bases. Commands can also be stored in files, then loaded into *RAD* and executed via a **consult** command.

In order to interact with agents, there must first be an Extensible Services Switch (ESS) running that provides communication and tree-space facilities. The ESS can be started by executing the following command on some workstation (assumed to be named "Delphi" here)

```
% ess -comm -client -tss
```

An arbitrary number of *RAD* agents and user interfaces can then be started. For an ESS on a host machine named "Delphi," an agent named "Zeus," and a user logged in as "User," this is done in the following two steps:

1. Create an agent by executing the following command in a window on "Delphi" or on another workstation

```
% radagent ;;This invokes lisp, in the PI package.
           ;;You must call "run" to start an agent.
<cl> (run 'Zeus 'Delphi)
      RAD Inference Engine
  Proprietary and Confidential
       MCC Carnot Project
```

```
Connecting to ESS on Delphi+17001
Agent ZEUS is registered with the ESS.

RAD Agent Zeus is ready.
```

2. Create a user interface process in another window, or on another workstation, by

```
% raduser -u User -e Delphi
      RAD User Interface
  Proprietary and Confidential
      MCC Carnot Project

Connecting to ESS on Delphi+17001...
ESS: Aide user-aide is ready.

User:
```

The prompt, **User:**, indicates who has said what or whose turn it is to talk.

The complete syntax for the user interface command is

```
raduser [-a agent]
        [-d]
        [-e ess-host [ess-port]]
        [-u user-name]
```

The options are

-a initialize connection with the given agent.

-d execute in debug mode; this mode is useful to yacc hackers.

-e find the ess on the specified host and, optionally, at the port where it is known to be listening. For example, the command

```
% raduser -e Cronos 17002
```

will cause a search for the ess on the host Cronos at port 17002. Since we anticipate that users might often use the same ess, we have added a shell variable RADSERVER that is the name of the default ess host. So for example, if you put

```
setenv RADSERVER delphi
```

in your `.cshrc`, then you don't need to use the -e option unless you wish to override your default. Please notice that the host name must be in lower case.

-u use the given user-name for a prompt, instead of the default. The default is the value of the shell variable RADUSER, if present; otherwise, it is the value of the shell variable USER.

You can connect the user interface to a different ESS by using the **ess** command. In either case, once connected to an ESS, the user interface can directly communicate with any other agent or interface that is also connected to this ESS. You specify the particular agent with which you want to communicate by using the **contact** command. To communicate with an agent connected to an ESS different than your own, you must use the **location** command to tell your ESS where the agent is. This enables you to communicate with an agent executing anywhere reachable through OSI. The argument to the **contact** command is either an agent name or a two-element list whose first element is a host name and whose second is a port number. An example of how to use these commands is the following:

```
(assume that User is currently served by an ESS on delphi,
and that agent Zeus is registered with the ESS on delphi.)
User: ess maui
User is unregistered from the ESS on delphi.
Interface: Looking for ESS at maui port 17001.
Connecting to ESS at maui+17001...

ESS: I am at maui port 17001.
ESS: Aide user-aide is ready.
User: location zeus delphi
User: list-agents
```

```
ESS: Registered agents and databases:
      user is at maui
      zeus is at delphi
User: contact zeus
Interface: pinging agent zeus...
zeus: READY
User:
```

The **ess** command can also accept either a host name or a host-port pair.

If you are running raduser under a unix shell, then control-C will interrupt and restart it. If running under GNU Emacs, do control-C control-C instead.

9.1 *RAD* Syntax

This section describes the syntax for *RAD*. A *RAD* program is a sequence of *statements*, each of which takes one of the following forms:

- A command, followed by its arguments. Some commands take a variable number of arguments; if there is more than one argument, the arguments must be enclosed in parentheses.
- A clause, such as an assertion or rule. In this case, an **accept** command is implicit.

Unit clauses take the form *relation(args)*. Lists and conses are written as in Lisp. All escapes to external functions (written in Lisp or (eventually) C) are prefaced by a comma, but are otherwise written as unit clauses. (They are written this way, rather than in the syntax of their parent language, for the sake of compatibility should the externals be implemented differently.) Variables are prefaced by a question mark.

Backward rules have the form

Infer *consequent* **from** *antecedents*

while forward rules have the form

If *antecedents* **then** *consequents*

where *consequent* is a unit clause and *antecedents* and *consequents* are either unit clauses or lists thereof. Forward rules may have an optional priority associated with them, which, if present, appears in brackets before the **If**. Unit clauses and rules may also be given labels by which they may be referenced later. This is done by writing the label name, followed by a colon, before the clause.

Users have great freedom in formatting their input as they like for the sake of readability. Input to *RAD* is not case-sensitive, except that case is preserved within literal strings, and for external tokens in the event that the external language is case-sensitive. Symbols can be written with any of the characters

$$\texttt{A}, ..., \texttt{Z}, \texttt{a}, ..., \texttt{z}, \texttt{0}, ...\texttt{9}, -, \_, /, *$$

Spaces, tabs, and newlines are mostly insignificant, except as follows:

- These characters normally serve to delimit lexical tokens. They can generally be omitted in cases where other punctuation is involved. For example, a rule label like `"foo:"` will be parsed as two tokens.

- Comments, which are prefaced by a semicolon, extend to the end of the input line.

- There can be no whitespace between a relation or function name and the left parenthesis that begins its argument list. This is necessary to distinguish between strings like `"foo(bar)"`, which is a single term, and `"foo (bar)"` which is two. Likewise, there can be no whitespace between the name of a typed logical variable and its type specifier: `"?x:baz"` is a typed logical variable, whereas `"?x :baz"` is an untyped variable followed by a Lisp keyword.

- When a syntax error is detected, the parser applies a very simple heuristic: the remainder of the input line is discarded, and an attempt is made to begin parsing the next statement at the next input line.

The following example is a simple consult file that uses most of the features of the above syntax. Almost all operations, particularly changes to the class lattice, are in the form of assertions on appropriate primitive relations. Thus, a *RAD* consult file can consist mostly of clauses, with the **accept** being implicit.

```
;;; Echo assertions
echo t

;;; Turn syntax-checking off
check nil

;;; Consult two files
consult ("my-utilities.rad" "my-classes.rad")

;;; Create three new classes
assert-instance (class mammal)
assert-instance (class dog cat)

;;; Manipulate the class lattice
assert-subclass (mammal dog)
assert-subclass (mammal cat)

;;; Define append using backward rules
append(nil ?x ?x)

infer append((?x . ?tail) ?y (?x . ?newtail))
      from append(?tail ?y ?newtail)

;;; Create a forward rule with priority of 20
frule1: [20] if (a(?x) b(?x) c(?x))
             then d(?x)

;;; Reference an external function
foo-rule: infer foo(?x ?y ?z ?w)
          from =((?x . ?y)
                 ,assoc(?z ?w :test #'eq)) ; Lisp escape

sys-rule: infer system-call(?x ?y)
          from =(?y ,system(?x))     ; C escape
```

9.2 The *RAD* Reader

The *RAD* reader is a collection of functions in the *RAD* agent that takes input from the parser and translates it into a form that is usable to the rest of the system. Part of the *RAD* reader's job is to *dereference* the names of objects that are passed to it. For example, in the following command, the strings "man" and "mortal" are taken to be the names of the relations *Man* and *Mortal*, respectively; these are defined if they do not already exist. This is because the *RAD* parser will recognize them as relations and will pass this information in the code that it generates for the agent.

```
User: Infer mortal(?x) from man(?x)

User: man(Socrates)
```

Likewise, the string "?x" is taken to be the name of a logical variable, ?X. As for "Socrates", it depends on whether or not "Socrates" is the name of something existing in the knowledge base. If SOCRATES were, for example, an instance of the class **PHILOSOPHER**, then the string would be taken to refer to that instance. However, if "Socrates" were not the name of any knowledge base object, it would simply be taken to stand for the symbol SOCRATES.

A common mistake, therefore, would be to define some string that has previously been interpreted as a symbol name to be the name of something else. Suppose, for example, the following were entered:

```
User: assert-instance (class philosopher)

User: man(socrates)

User: assert-instance (philosopher socrates)

User: ? man(socrates)
```

The query will unexpectedly fail. This is because the "Socrates" to which the query refers is the instance of **PHILOSOPHER**, whereas the one to which the assertion refers is just a symbol. (Of course, the query will work if the assertion and the **assert-instance** commands are interchanged.) *RAD* will

generate a warning if the user attempts to create an object with the same name as a symbol that it has previously encountered, but the operation will be performed anyway.

9.2.1 The Slash – /

The slash is used extensively in the names of a number of objects that are internal to *RAD*. Its use in the names of user-defined objects is therefore potentially dangerous and strongly discouraged.

9.2.2 *RAD* Reader Macros

A number of syntactic constructs are implemented at the reader level by *macros*. These signal the reader to perform some kind of expansion on the input.

Question Mark – ?

By itself, the question mark expands into the name of the function `PROVE`, which is an internal *RAD* function that implements a user query. Before another question mark, it expands into `PROVE-ALL-INSTANCES`, the *RAD* function that proves all instances of a goal.

A question mark appearing before anything else is, of course, taken to be the first character in the name of a logical variable. Avoid any use of the question mark that is not consistent with these conventions.

Comma – ,

The comma is intended to provide a more concise and usable interface to Lisp than is provided by function declarations and the relations *Is* and *Do*. In general, the appearance of a comma in the antecedent of a forward or backward rule indicates that the following expression should be evaluated in Lisp. The reader examines the expression and generates appropriate calls to the relation *Is*. For example:

```
Infer q(?x ?y) from ( ,f(?x) ,?y) )
```

is translated internally to

```
Backward Rule Q-1     (IN)
Infer Q(?X ?Y) from ( IS(NIL (NOT (F ?X)))
                      IS(NIL (NOT (EVAL ?Y))) )
```

Commas can appear directly before a Lisp expression in the antecedent of a forward or backward rule, as in

```
Infer p(?x) from ,f(?x)
```

A proof of p(?x) will succeed as long as the lisp function (F ?X) does not evaluate to NIL for the current binding of ?X. Symbols can also be evaluated using the comma, although special forms cannot.

Expressions with commas can be used within relations, for example:

```
Infer p(?x) from q(,*foo*)
```

is translated to

```
Backward Rule P-2  (IN)
Infer P(?X) from ( IS(?/NEW0 (EVAL (QUOTE *FOO*)))
                   Q(?/NEW0) )
```

Note that the system automatically generates a logical variable to hold the result of the symbol evaluation.

Most importantly, however, the bindings of logical variables can be evaluated in Lisp. For instance,

```
Infer p(?x) from ,?x
Backward Rule P-3     (IN)
Infer P(?X) from IS(NIL (NOT (EVAL ?X)))
```

succeeds if the current binding of ?X is `(CAR '(A))`, or any Lisp expression that does not evaluate to nil.

Logical variables can appear anywhere in the scope of a comma. For instance, the following are all legal:

```
Infer q(?x ?y) from p( ,?x ,?y)

Infer p(?x) from ,f(?x)

Infer q(?x ?y) from p( ,f(?x) ,?y)
```

Functions that are referenced within the scope of a comma must be visible in the package PROT. This is true of symbols as well. The usual Common Lisp syntax conventions apply when referring to objects in other packages. Passing an unbound logical variable to Lisp, however, is not in itself an error. In this case, the variable will evaluate to itself.

Since commas get compiled into calls to *Is*, the user should observe the same care as if the use of *Is* were explicit. In the following example,

```
Rule-1:  If ,f(?x) then p(?x)
Forward Rule RULE-1     (IN)
      If IS(NIL (NOT (F ?X)))
      then P(?X)
```

RULE-1 will never fire, since *Is* is not triggerable. Another problem can arise if all references to a logical variable in the head of a forward rule appear within the scope of a comma. Such a variable will never be bound to anything. For instance,

```
If p( ,?x) then q(?x)
Forward Rule FRULE-1     (IN)
      If ( IS(?/NEW0 (EVAL ?X))
          P1(?/NEW0) )
      then
P(?X)
```

There is no way to assign a binding to ?X at top level. Hence the rule will not behave as one might expect.

The *RAD* reader allows only the first comma preceding a Lisp expression or logical variable to be a macro character. Subsequent commas revert to their traditional meaning, and must occur within backquoted expressions. For example,

```
Infer foo() from ,,X
```

produces an error. If a sequence of evaluations is desired, then it must be made explicit, *to wit:*

```
Infer foo() from ,(eval x)
```

Lisp macros can also be evaluated, but caution should be exercised since macro expansion takes place at rule compilation time rather than at proof time, even though this is not apparent from looking at the rule. For example, if the following macro is defined in lisp,

```
(defmacro bar (x) '(prog1 t (format t ''~&Hi ~A'' ,x)))
```

then RAD would have the following behavior:

```
User: Infer baz(?x) from ,bar(?x)

Zeus: Backward Rule BAZ-1     (IN)
Infer BAZ(?X) from IS(NIL (NOT (BAR ?X)))

User: ? baz(mom)
Zeus: Hi MOM
Zeus: Yes.
```

If the macro is then redefined as follows

```
(defmacro bar (x) '(prog1 t (format t "~&Bye ~A" ,x)))
```

then RAD would have the following behavior:

```
User: ? baz(mom)
Zeus: Hi MOM
Zeus: Yes.
```

In order to produce the output "Bye MOM" one would have to erase `BAZ-1` and reassert the same rule.

Unless

Although users can normally think of *Unless* as a built-in relation, it is implemented as a macro. Unfortunately, this is not completely transparent at the user interface level. For example,

```
Infer inanimate(?x) from unless(living(?x))

Backward Rule INANIMATE-1     (IN)
Infer INANIMATE(?X)
       from
      /UNLESS/0(?X PI::*ARG-END-SYM* LIVING)
```

/Unless/0 is a relation that is automatically created by macro expansion. Although this expansion is not transparent, users need not concern themselves as the semantics remain the same as if *Unless* were actually defined as a relation.

9.2.3 Error Detection

In addition to the syntactic checking performed by the parser, the *RAD* reader tries to detect errors that result from input that is meaningless in light of the contents of the knowledge base. While some errors will still throw the user into the debugger, many common ones can be easily detected at the reader level.

Error detection is enabled at the start of `radagent`. Users can turn error detection off and on via the **check** command.

Errors fall into two classes, depending on their severity:

Warnings are printed when the user tries to do something questionable, where the action is otherwise syntactically correct and where aborting the action would be difficult or costly. The action is performed anyway. An example would be attempting to define a frame whose name had previously been used as an uninterpreted symbol.

Errors are printed when faulty syntax is detected before a command is processed. The command is not executed and control returns to *RAD* top level. An example would be supplying too many arguments to a top-level command.

Warnings

- *Attempt to redefine assumed symbol:* A warning is printed if an object is created that shadows an uninterpreted symbol that the *RAD* reader has previously encountered. For example:

```
foo(bar)

User: assert-instance (class bar)

Zeus: Warning: BAR, previously assumed to be of type
```

```
SYMBOL, is being redefined to be of type CLASS.
```

- *Attempt to redefine object: RAD* is creating an object that shadows an object previously defined by the user. For example:

```
User: f-1()

User: f()

Zeus: Syntax Warning: Attempt to redeclare or redefine
the predicate F-1 to be of type ASSERTION.

The new definition will shadow the old.
```

  Note that an actual error results if the *user* attempts to redefine an object (other than a symbol). See below.

- *Relation arity violation:* A relation has been supplied with the wrong number of arguments. This is normally an error (see below); a warning is generated only in the case of the **ask** and **translate** commands:

```
argument-types(baz (symbol))

translate baz(?x ?y) "~A is a ~A BAZ!" (?x ?y)

Syntax Warning: The arguments to the predicate BAZ,
?X ?Y, are in excess of its arity (1).
```

Errors

- *Type error:* An argument to a relation was of the wrong type.

```
User: assert-instance (relation foo)
User: argument-types(foo (class))
User: foo(5)
Zeus: The first argument to FOO, 5, is not of type CLASS.
```

- *Attempt to redefine:* The user has attempted to redeclare or redefine an object.

```
User: dog(5)

User: if d1() then assert-instance(relation dog-1)

User: d1()
Zeus: Attempt to declare Assertion DOG-1 to be a RELATION.
```

- *Relation arity violation:* A relation was referenced with the wrong number of arguments.

```
User: argument-types(foo (symbol))
User: foo(dog cat)
Zeus: Syntax Error: The arguments to the relation FOO,
DOG CAT, are in excess of its arity (1).
```

- *Unrecognized type:* The user qualified a logical variable to be an unknown type.

```
User: assert-instance (relation foo)
User: ? foo(?x:dawg)
Zeus: Syntax Error: DAWG is not a recognized type.
```

9.3 Top Level Commands

This section describes the top level commands available to the user within the stream interface. They can be entered from a keyboard or consulted by *RAD* from a file.

9.3.1 Creating and Modifying a Knowledge Base

accept *pattern* — (*&rest patterns*)

This command adds its arguments to the knowledge base, each with a justification consisting of the premise that an agent **said** the pattern and the

default that the agent is *reliable*, i.e., the agent cannot be proven to be *unreliable*. Note that the command name **accept** is typically omitted, without causing any ambiguity.

```
User: accept has-son(King Prince)

User: has-son(King Duke)
```

If the relation *Has-son* did not already exist in the knowledge base, it would be created and given an arity of two. However, if `KING`, `Duke`, and `PRINCE` did not already exist, they would be accepted as uninterpreted symbols by *RAD*.

An accept may also be a backward rule:

```
User: Mortality-rule: Infer mortal(?x) from man(?X)
Zeus: Backward Rule MORTALITY-RULE accepted.
```

Again, the relations *Man* and *Mortal* are created if they do not already exist. The user-supplied name, `"Mortality-rule:"`, is optional.

Finally, an accept may be a forward rule:

```
User: If mother(?x ?y) then parent(?x ?y)
```

The rule may be preceded by a symbol to be used as the name of the rule and/or by a two-digit priority:

```
User: Parent-rule: [25] If mother(?x ?y) then parent(?x ?y)
```

assert *pattern* — (*&rest patterns*)

This command adds its arguments to the knowledge base, each with a *premise* justification. Note that the command name **assert** is typically omitted, without causing any ambiguity.

```
User: assert has-son(King Jester)
Zeus: Assertion HAS-SON-7    (IN)
```

An assertion may also be a backward or forward rule.

assert-instance (*class &rest names*)

This command creates instances of the given class. If *names* is omitted, a name for the instance is generated by the system.

```
User: assert-instance (class dog cat)

User: show dog
Zeus: DOG is an instance of CLASS
  Subclass of:     none
  Subclasses:      none
  Instances:       none
```

assert-subclass (*super &rest subs*)

This command causes *super* to be made an immediate superclass of each of the *subs*. All arguments must be existing classes.

```
User: assert-subclass (thing animal vegetable mineral)

User: show thing
Zeus: THING is an instance of CLASS
  Subclass of:     none
  Subclasses:      ANIMAL, VEGETABLE, MINERAL
  Instances:       none
```

clear

This command reinitializes *RAD* by removing data and rules that are not system-defined.

consult *file* — (*&rest files*)

This loads the indicated consult files into *RAD*. The files *must* be written as strings.

```
consult "test1.rad"
```

contradict *datum-name*

This command takes an existing datum as an argument and contradicts it. See also **wrong**, which can be used to contradict the last proof result.

erase *datum-name* — (*&rest data*)

The given data are to be made OUT. The JTMS then updates the support status of other data accordingly.

```
User: erase mortal-1
Zeus: OK.

User: why mortal-1
Zeus: Assertion MORTAL-1     (OUT)
      MORTAL(SOCRATES)

JUSTIFICATION:
    in-list    NIL
    out-list    NIL

was erased by assertion ERASED-1

User: show erased-1
Zeus: Assertion ERASED-1     (IN)
      ERASED(MORTAL-1)
```

kill *thing* — (*&rest things*)

This deletes objects from the knowledge base. The objects are actually removed and their names are bound to a special symbol, DELETED. Contrast this with **erase**, which leaves data around, but puts them OUT. Moreover, **erase** only makes sense for data, i.e., things that actually have a support status, whereas **kill** can be used on data, relations, and frames.

```
User: kill (foo bar-1)
```

remember

When typed after a successful query, **remember** creates a datum representing the result of that query and asserts it. This is primarily useful for saving the result of a long proof so that the backward chainer would not have to repeat the proof.

```
User: ? likes(joe ?x)
Zeus: LIKES(JOE CANDY)

User: remember
Zeus: Remembered as Assertion LIKES-2.

User: show likes-2
Zeus: Assertion LIKES-2     (IN)
      LIKES(JOE CANDY)
```

remove-subclass (*super &rest subs*)

This removes all parent-child links between *super* and each element of *subs*. All arguments must be classes, of course.

```
User: remove-subclass (thing animal)
```

reply *agent "clause"*

Reply causes *clause* to be sent to the agent *agent*. The *clause* can be a datum, rule, or value, but it **must** be in the form of a string.

```
Zeus: P(A ?X)?
User: reply zeus "P(A 5)"
```

tell *agent clause*

Tell causes *clause* to be asserted in the knowledge base of the agent *agent*, as described in Chapter 2. The justification for *clause* has an `IN`-list consisting of `reliable(SOURCE-AGENT)` and `said(SOURCE-AGENT CLAUSE)`, and an empty `OUT`-list, where `SOURCE-AGENT` is the name of the agent that executed the relation *Tell*. Note that *clause* may be a rule as well as a unit clause.

toggle *datum-name* — (*&rest data*)

This command changes the support status of *datum-name* from `IN` to `OUT` or `OUT` to `IN`, as appropriate. Any other datum dependent on *datum-name* may change its support status also.

wrong

This command is used to contradict the last goal proven by **?** or **next**. See also **contradict**, which can be used to contradict an existing datum.

9.3.2 Querying and Examining Knowledge Bases and Databases

? *goal*

This command attempts to prove a goal by backward chaining. If the query succeeds the solution is printed, with appropriate variable bindings if variables were present in the goal pattern. If the query fails, then `NO SOLUTION` is printed.

```
User: age(hilary 7)
Zeus: Assertion AGE-1 accepted.

User: ? age(hilary ?x)
Zeus: AGE(HILARY 7)
```

?? *goal*

This is like **?**, except that it constructs all proofs of a goal pattern. All successfully proved instances of the goal are output immediately.

```
User: ?? age(?x 7)
Zeus: AGE(HILARY 7)
      AGE(JENNIFER 7)
```

explain

Explain, when typed after a successful query, shows why the query succeeded. For example,

```
User: ? mortal(socrates)
Zeus: MORTAL(SOCRATES)

User: explain
Zeus: MORTAL(SOCRATES)

was derived from the following:
Assertion MAN-1     (IN)
  MAN(SOCRATES)

Backward Rule MORTAL-1     (IN)
  Infer MORTAL(?X) from MAN(?X)
```

next

When typed after a successful call to ?, **next** initiates another proof attempt of the same goal in order to derive another solution, until all instances of the goal have been derived.

```
User: ? age(?x 7)
Zeus: AGE(HILARY 7)

User: next
Zeus: AGE(JENNIFER 7)

User: next
Zeus: NO SOLUTION
```

query *agent goal*

Query succeeds if *goal* is provable by the agent *agent*. The agent returns all possible proofs of *goal*, which are then used one at a time. For each successful proof, an entry for the proof is created in a table of remote proofs, with a pointer to *agent* who supplied the proof.

query-once *agent goal*

Query-once succeeds if *goal* is provable by the agent *agent*. If so, it creates an entry for the proposition in its table of remote proofs, with a pointer to *agent* who supplied the proof.

db-query *database pattern*

Db-query succeeds if the database *database* contains any tuples that match *pattern*. The database returns all possible matching tuples, which are then used one at a time by the agent that issued the **db-query** command. Internally, the interface converts this command into an equivalent query in SQL syntax that is processed according to the RDA protocol. For this query to succeed, the database must already be opened (see the commands **db-connect** and **db-disconnect**).

```
User: contact zeus
Zeus: READY
User: db-connect TravelDB delphi
User: db-query TravelDB hotelinventory(?hotel ?city)
Zeus: hotelinventory("Hilton" "San Diego")
      hotelinventory("Sheraton" "Denver")
      hotelinventory("Hyatt" "Austin")
User: db-disconnect TravelDB
```

show *object* — (*&rest objects*)

Show displays the printed representation of its arguments, which may be data, relations, or frames.

```
User: show age-1
Zeus: Assertion AGE-1     (IN)
  AGE(HILARY 7)

User: show dog
Zeus: DOG is an instance of CLASS
  Subclass of:    none
  Subclasses:     none
  Instances:      FIDO
```

why *datum*

This command explains the support status of a datum. If the datum is IN, the IN- and OUT-lists of its justification are printed. If it is OUT, but has justifications, then the data that invalidate its justifications are printed.

```
User: why mortal-1
Zeus: Assertion MORTAL-1     (IN)
  MORTAL(SOCRATES)

was derived from the following:
Forward Rule FRULE-0     (IN)
  If
   MAN(?X)
  then
   MORTAL(?X)

Assertion MAN-1     (IN)
  MAN(SOCRATES)

User: erase mortal-1
Zeus: OK.

User: why mortal-1
Zeus: Assertion MORTAL-1     (OUT)
  MORTAL(SOCRATES)

JUSTIFICATION:
    in-list     (FRULE-0 MAN-1)
    out-list     NIL

was erased by Assertion ERASED-1
```

9.3.3 Customizing the Interface

ask *pattern string bound-vars unbound-vars t-or-nil prompts*

This command creates an object called a *query schema*, which determines a format in which the user may be prompted for information during the course of a proof, by means of the built-in relations *Ask* and *Ask-once* (see Chapter 6). A query schema consists of the following components:

pattern is a clause, possibly containing variables, with which the query schema is associated.

string is a string to be used as the second argument to the Lisp function `FORMAT`; it may contain occurrences of `~A`.

bound-vars is a list of some of the variables that occur in *pattern*; its length must be the number of occurrences of `~A` in *string*.

unbound-vars is a list of the variables occurring in *pattern* that do not belong to *bound-vars*.

t-or-nil is either `T` or `NIL`, depending on whether or not the query determined by *string* requires a yes-or-no response.

prompts is a list of strings of the same length as *unbound-vars*.

A query schema is relevant to a goal if the goal is an instance of its *pattern* with each variable in *bound-vars* replaced by a constant. When the user is to be queried for an instance of a goal that is being processed by the backward chainer, the list of query schemata associated with the relation of the goal is examined until either the end of the list is reached or one is found that is relevant to the goal. In the first case, the default format (see the definition of **Ask**) is used for the query. Otherwise, a format is determined by the relevant query schema as described below.

As an example, consider the command

```
ask birthplace(?X ?Y ?Z) "Enter the birthplace of ~A"
      (?X) (?Y ?Z) NIL ("  city: " "  state: ")
```

Suppose that after this command has been issued, the user is to be queried for an instance of the goal `BIRTHPLACE(MICHAEL ?C ?S)`. Since this goal is an instance of the pattern `BIRTHPLACE(?X ?Y ?Z)` with `?X` (the only variable in *bound-vars*) replaced by the constant `MICHAEL`, the query schema defined by the command is relevant. The *string* is then printed with `~A` replaced by `MICHAEL`. The user is then prompted for values of `?Y` and `?Z`:

```
Enter the birthplace of MICHAEL
  city: BOSTON
  state: MASSACHUSETTS
```

In this case, the resulting assertion

```
BIRTHPLACE(MICHAEL BOSTON MASSACHUSETTS)
```

is successfully matched against the goal.

If *t-or-nil* is T, as in

```
ask birthplace(?X ?Y ?Z) "Do you know where ~A was born?"
       (?X) (?Y ?Z) T ("  city: " "  state: ")
```

then a yes-or-no response is expected after *string* is printed, determining whether to proceed with the query:

```
Do you know where MICHAEL was born? (Y or N) Yes
  city: BOSTON
  state: MASSACHUSETTS

Do you know where BOB was born? (Y or N) No
```

Query schemata are matched against a goal in the order in which they were originally defined. If several query schemata are relevant to a goal, than this order is significant. Generally, they should be entered in order of decreasing specificity.

```
ask birthplace(?X ?Y ?Z) "Was ~A born in ~A,~A?"
    (?X ?Y ?Z) () T ()

ask birthplace(?X ?Y ?Z) "Do you know where ~A was born?"
    (?X) (?Y ?Z) T ("  city: " "  state: ")

ask birthplace(?X ?Y ?Z) "Do you know anyone born in ~A?"
    (?Z) (?X ?Y) T ("  who? " "  what city? ")
```

In this case, all three schemata are relevant to the goal `BIRTHPLACE(JIM CLEVELAND OHIO)`, but only the first would be used in querying about this goal.

check *t-or-nil*

The **check** command controls whether or not the agent checks for errors in the parser input. It takes one argument, which can be `T` or `NIL`: **check nil** suppresses the checking for and printing of errors and warnings, while **check t** turns error detection back on. Note that this applies only to the agent: the syntax checking performed by the parser cannot be disabled.

echo *t-or-nil*

This command turns on the echoing of user-defined assertions and rules. This behavior is enabled by default when *RAD* starts up, and can be shut off with **echo nil**.

translate *pattern string vars*

This command specifies *translation schema* to be associated with certain patterns, in order that they can be printed in a more readable form. Each schema is specified by three things:

- A *pattern* to be translated when printed.
- A *string* specifying the translation. This should be a string acceptable to the **format** function, with ~A's used to mark the place of logical variables.
- A list of *variables* that are passed as arguments to **format**. These must be a subset of the variables appearing in *pattern*.

For example,

```
translate age(?x 1) "~A is a year old." (?x)

translate age(?x ?y) "~A is ~A years old." (?x ?y)
```

creates two query schema associated with the relation *Age*. If one were to assert `AGE(GEORGE 15)`, the following would be printed:

```
Assertion AGE-1    (IN)
  GEORGE is 15 years old.
```

Note that the translation schema will be matched in the order they are declared. Users should therefore enter the most specific schema first.

9.3.4 Miscellaneous Commands

bgrind *reln*

This takes a relation *reln* and outputs the WAM code generated for its backward rules.

fgrind *reln*

This command takes a relation *reln* and outputs the WAM code generated for its forward rules.

contact *arg*

The **contact** command is used to connect the user interface to different agents. For agents connected to a different ESS than the one to which you are connected, you must first issue the **location** command. This enables a user to communicate with an agent executing anywhere that is reachable by TCP/IP or OSI. The argument to the **contact** command is either an agent name or a two-element list whose first element is a host name and whose second is a port number.

db-connect *database host*

The **db-connect** command is used to connect the user interface to a database. This causes an open-db instruction in the RDA protocol to be executed. The arguments to the **db-connect** command are a database name and the name of the host where the database is located.

db-disconnect *database*

The **db-disconnect** command is used to disconnect the user interface from a database. This causes an close-db instruction in the RDA protocol to be executed. The argument to the **db-disconnect** command is a database name.

ess *arg*

The **ess** command is used to connect the user interface to an ess and, thereby, directly to the set of agents that the ess serves. The argument to the **ess** command is either a host name or a two-element list whose first element is a host name and whose second is a port number.

location *arg*

The **location** command is used to connect the user interface to agents that are connected to and serviced by other ESS's. The two arguments to the **location** command are an agent name and the host where that agent's ESS is executing.

halt-agent *agent*

The **halt-agent** command is used to stop the execution of an agent *with which you are in contact* (see the **contact** command). The argument to this command is the name of the agent to be halted. The agent unregisters itself from its ESS and returns control to the operating system.

list-agents

The **list-agents** command is used to display all of the agents and databases that are known to the ESS, for example

```
User: list-agents
ESS: Registered agents and databases:
      user is at delphi
      zeus is at delphi
      traveldb is at olympus
User:
```

help *command*

This command provides a simple online help facility for most commands.

```
User: help assert-instance
Zeus: ASSERT-INSTANCE (Class &Rest Names)
```

quit

This exits the user interface and returns the user to the operating system. This does not affect the operation of any agents with which the user interface may have been communicating.

Chapter 10

Installation

10.1 Environment

The Distributed Communicating Agent software currently operates on the following system architectures:

- Sun 4 with SunOS 4.x and Allegro 4.0.1 Common Lisp
- Sun 3 with SunOS 4.x and Allegro 4.0.1 Common Lisp
- DECstation 5100 with Ultrix V4.2 and Lucid Common Lisp Version 4.0

In order to build the Distributed Communicating Agent software, you must have the following software tools installed:

- C development tools, e.g., cc, libc.a, and include files
- C library containing socket system calls, if not already in libc.a
- Common Lisp

Additionally, in order to execute the software, you must have a Rosette Extensible Services Switch (ESS) executing with its client, communication, and tree-space options available.

10.2 Sources

The source files for the Distributed Communicating Agent software are in four directories. The files are described relative to a Carnot root directory, typically `/usr/carnot`. The shell ENVIRONMENT variable `CARNOTDIR` must be set to this prior to making any of the components, e.g.,

```
% setenv CARNOTDIR /usr/carnot
```

The four directories, depicted in Figure 10.1, contain

1. sources for *RAD* agents,

   `~carnot/semantic/dca/release/base/src/agent/*`

2. sources for *RAD* interfaces (human agents),

   `~carnot/semantic/dca/release/base/src/interface/*`

3. a library of definitions for built-in *RAD* commands and funtions,

   `~carnot/semantic/dca/release/base/src/radlib/*`

4. a library of example knowledge bases, some of which are described in the appendix to this document

   `~carnot/lib/dca/*`

10.3 The MAKE Process

Installation of the Distributed Communicating Agent software is automated by Makefiles. However, you need to define several MAKE variables for your particular environment. The `carnot/config` directory contains configuration Makefiles for the supported architectures. Examine the appropriate file and modify it for your environment. In particular, set the value of the variable `RADLISP` to the path where your lisp is located. Then run the carnot/Makefile. For example, to compile and install the software on a Sun4, do the following from `CARNOTDIR`:

```
% cmake semantic-svcs
```

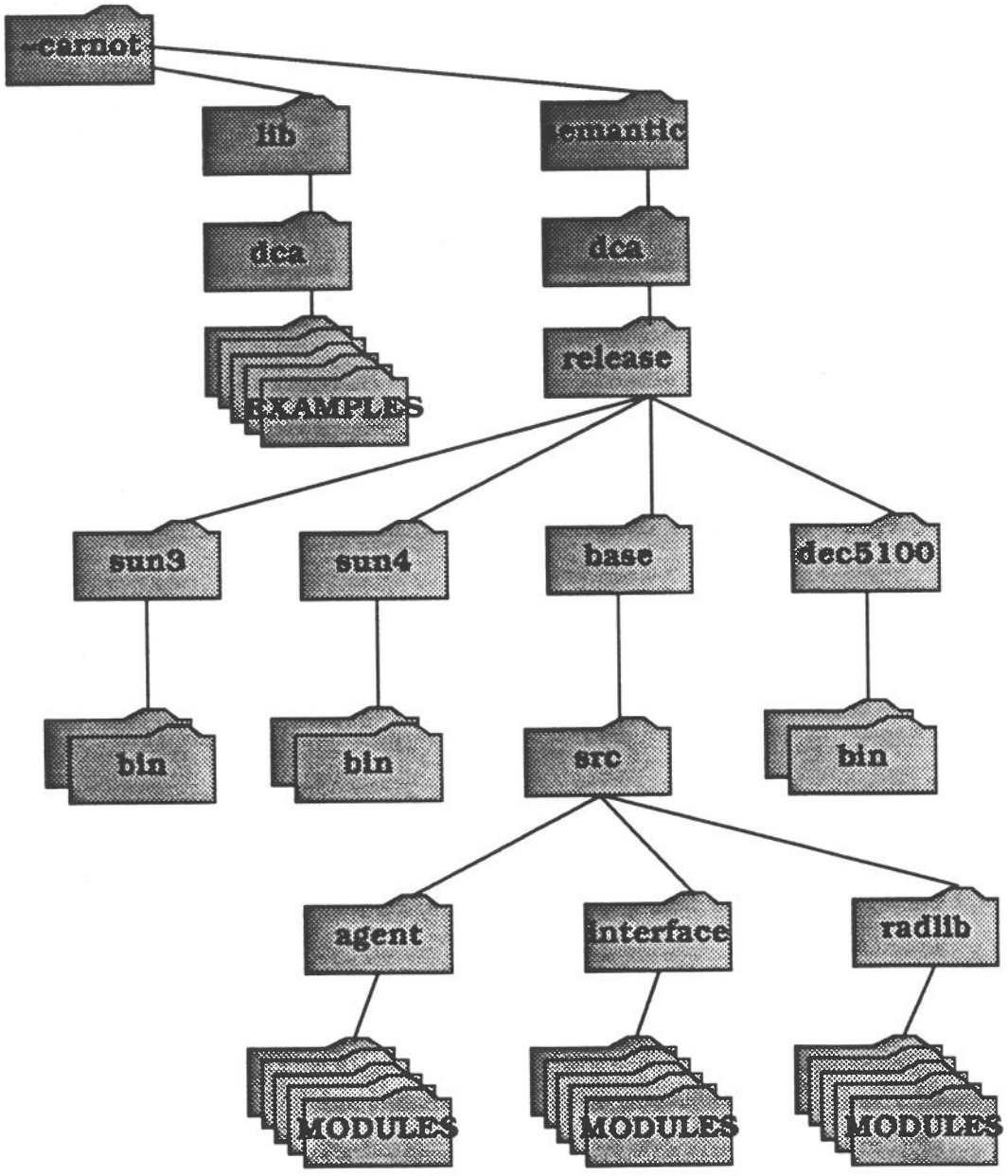


Figure 10.1: Organization of the Distributed Communicating Agent software

Appendix A

Multiagent Tic-Tac-Toe

This section of the document describes a *RAD* implementation of tic-tac-toe involving four agents: a referee, two players, and a human observer. The human observer communicates with the referee to initiate the game. The referee 1) determines if the two players are ready to play the game, 2) finds out if the players have a preference for "X" or "O" and assigns marks accordingly, 3) "flips a coin" to determine which player goes first, 4) successively asks each player for his next move, 5) checks each move for legality, 6) determines when a game is over, and 7) declares a winner (if any). The players each have knowledge about tic-tac-toe and a strategy for playing it.

A.1 Knowledge Base for Referee

```
;;*** Class and Instance Declarations ***

assert-instance(class player)
assert-subclass(agent player)
assert-instance(player PlayerA PlayerB)
assert-instance(agent referee)
assert-instance(class game)
assert-instance(class nought-or-cross)
assert-instance(nought-or-cross O X)
assert-instance(class index)
```

```
assert-instance(index zero one two)
assert-instance(class agent-capability)
assert-instance(agent-capability tic-tac-toe-playing)
assert-instance(game game1)

;;*** Relation Declarations ***

assert-instance(relation mod1)
argument-types(mod1 (index index))

assert-instance(relation mod2)
argument-types(mod2 (index index))

assert-instance(relation expertise) ; multiple-valued
argument-types(expertise (agent agent-capability))
specificity-ordered(expertise)

assert-instance(relation awake)
argument-types(awake (agent symbol))
single-valued(awake)
specificity-ordered(awake)

assert-instance(relation abnormal)
argument-types(abnormal (agent))

assert-instance(relation play-tic-tac-toe)
argument-types(play-tic-tac-toe (player player game))

assert-instance(relation tic-tac-toe)
argument-types(tic-tac-toe (player player game))

assert-instance(relation mark)
argument-types(mark (player game nought-or-cross))
single-valued(mark)
specificity-ordered(mark)

assert-instance(relation prefer-mark)
```

```
argument-types(prefer-mark (player nought-or-cross))
single-valued(prefer-mark)
specificity-ordered(prefer-mark)

assert-instance(relation opposite-mark)
argument-types(opposite-mark (nought-or-cross nought-or-cross))
single-valued(opposite-mark)
specificity-ordered(opposite-mark)

assert-instance(relation coin-toss-result)
argument-types(coin-toss-result (game number))
single-valued(coin-toss-result)
specificity-ordered(coin-toss-result)

assert-instance(relation goes-next)
argument-types(goes-next (game fixnum player))
specificity-ordered(goes-next)

assert-instance(relation other-player)
argument-types(other-player (player player game))
single-valued(other-player)
specificity-ordered(other-player)

assert-instance(relation board)
argument-types(board (game index index nought-or-cross))
specificity-ordered(board)

assert-instance(relation game-over)
argument-types(game-over (game))
specificity-ordered(game-over)

assert-instance(relation tie-game)
argument-types(tie-game (game))
specificity-ordered(tie-game)

assert-instance(relation wins-game)
argument-types(wins-game (player game))
```

```
specificity-ordered(wins-game)

assert-instance(relation move)
argument-types(move (player index index game))
specificity-ordered(move)

assert-instance(relation okay-move)
        argument-types(okay-move (game player index index))
specificity-ordered(okay-move)

;;*** End of Relation Declarations ***

        mod1(zero one)
        mod1(one two)
        mod1(two zero)

        mod2(zero two)
        mod2(one zero)
        mod2(two one)

expertise(referee tic-tac-toe-playing)
opposite-mark(X O)
opposite-mark(O X)

Infer awake(?Agent Yes)
from unless(abnormal(?Agent))

;; Begin a game of tic-tac-toe between two Players.
;; Make sure the Players are ready to play...

Begin-tic-tac-toe:
  If  (play-tic-tac-toe(?Player1 ?Player2 ?Game)
       query-once( ?Player1 awake(?Player1 Yes))
       query-once( ?Player2 awake(?Player2 Yes)))
  then (tic-tac-toe(?Player1 ?Player2 ?Game)
        other-player(?Player1 ?Player2 ?Game)
        other-player(?Player2 ?Player1 ?Game)
```

```
        tell( ?Player1 other-player(?Player1 ?Player2 ?Game))
        tell( ?Player2 other-player(?Player2 ?Player1 ?Game)))

Trigger-mark-choice:
  If  (tic-tac-toe(?Player1 ?Player2 ?Game)
       excuse(unless( marks-chosen(?Game))))
  then (preference-rule-enabled(?Game)    ;Order of consequents
        no-preference-rule-enabled(?Game));determines order of
                                          ;rule firing.


Choose-marks-1: ;give Player1 his preference
  If  (tic-tac-toe(?Player1 ?Player2 ?Game)
       excuse(unless(marks-chosen(?Game)))
       preference-rule-enabled(?Game)
       query-once( ?Player1 prefer-mark(?Player1 ?Mark))
       opposite-mark(?Mark ?Other-mark))
  then (mark(?Player1 ?Game ?Mark)
        mark(?Player2 ?Game ?Other-mark)
        tell( ?Player1 mark(?Player1 ?Game ?Mark))
        tell( ?Player1 mark(?Player2 ?Game ?Other-mark))
        tell( ?Player2 mark(?Player1 ?Game ?Mark))
        tell( ?Player2 mark(?Player2  ?Game ?Other-mark))
        marks-chosen(?Game))

Choose-marks-2: ;give Player2 his preference
  If  (tic-tac-toe(?Player1 ?Player2 ?Game)
       excuse(unless( marks-chosen(?Game)))
       preference-rule-enabled(?Game)
       query-once( ?Player2 prefer-mark(?Player2 ?Mark))
       opposite-mark(?Mark ?Other-mark))
  then (mark(?Player2 ?Game ?Mark)
        mark(?Player1  ?Game ?Other-mark)
        tell( ?Player1 mark(?Player2 ?Game ?Mark))
        tell( ?Player1 mark(?Player1  ?Game ?Other-mark))
        tell( ?Player2 mark(?Player2 ?Game ?Mark))
        tell( ?Player2 mark(?Player1  ?Game ?Other-mark))
```

```
        marks-chosen(?Game))

Choose-marks-3: ;assign marks if no preferences
  If  (tic-tac-toe(?Player1 ?Player2 ?Game)
       excuse(unless( marks-chosen(?Game)))
       no-preference-rule-enabled(?Game))
  then (mark(?Player1 ?Game X)
        mark(?Player2 ?Game O)
        tell( ?Player1 mark(?Player1 ?Game X))
        tell( ?Player1 mark(?Player2 ?Game O))
        tell( ?Player2 mark(?Player1 ?Game X))
        tell( ?Player2 mark(?Player2 ?Game O))
        marks-chosen(?Game))

Decide-who-goes-first: ;the referee "flips a coin"
  If  (tic-tac-toe(?Player1 ?Player2 ?Game)
       marks-chosen(?Game)
       excuse(unless(coin-toss-result(?Game ?Number))))
  then coin-toss-result(?Game ,random(1.0)) ;0 <= ?X < 1.0

  If  (tic-tac-toe(?Player1 ?Player2 ?Game)
       coin-toss-result(?Game ?X)
       ,>=(?X 0.5)) ;Player1 goes first
  then goes-next(?Game 1 ?Player1)

  If  (tic-tac-toe(?Player1 ?Player2 ?Game)
       coin-toss-result(?Game ?X)
       ,<(?X 0.5)) ;Player2 goes first
  then goes-next(?Game 1 ?Player2)

Incorporate-next-player-move:
  If (goes-next(?Game ?Move-number ?Player1) ;Next player moves
      excuse(unless(goes-next(?Game ,1+(?Move-number) ?Anyone)))
      excuse(unless(game-over(?Game)))       ;unless game over
      ,<(?Move-number 10)
      okay-move(?Game ?Player1 ?RMove ?CMove);Get next move
      mark(?Player1 ?Game ?Mark)
```

```
      other-player(?Player1 ?Player2 ?Game))
  then (board(?Game ?RMove ?CMove ?Mark)
        tell( ?Player1 board(?Game ?RMove ?CMove ?Mark))
        tell( ?Player2 board(?Game ?RMove ?CMove ?Mark))
        print("~%Move ~A: square ~A ~A is occupied by ~A"
       (?Move-number ?RMove ?CMove ?Player1))
        goes-next(?Game ,1+(?Move-number) ?Player2))
                                    ;Other player is next

Find-good-move:
  Infer okay-move(?Game ?Player ?RMove ?CMove)
  from (query-once(?Player move(?Player ?RMove ?CMove ?Game))
        excuse(unless(board(?Game ?RMove ?CMove ?Any-mark))))

Test-row-1:
  If (mark(?Player ?Game ?Mark)
      board(?Game zero zero ?Mark)
      board(?Game zero one ?Mark)
      board(?Game zero two ?Mark))
  then (wins-game(?Player ?Game)
        game-over(?Game))

Test-row-2:
  If (mark(?Player ?Game ?Mark)
      board(?Game one zero ?Mark)
      board(?Game one one ?Mark)
      board(?Game one two ?Mark))
  then (wins-game(?Player ?Game)
        game-over(?Game))

Test-row-3:
  If (mark(?Player ?Game ?Mark)
      board(?Game two zero ?Mark)
      board(?Game two one ?Mark)
      board(?Game two two ?Mark))
  then (wins-game(?Player ?Game)
        game-over(?Game))
```

```
Test-column-1:
  If (mark(?Player ?Game ?Mark)
      board(?Game zero zero ?Mark)
      board(?Game one zero ?Mark)
      board(?Game two zero ?Mark))
  then (wins-game(?Player ?Game)
        game-over(?Game))

Test-column-2:
  If (mark(?Player ?Game ?Mark)
      board(?Game zero one ?Mark)
      board(?Game one one ?Mark)
      board(?Game two one ?Mark))
  then (wins-game(?Player ?Game)
        game-over(?Game))

Test-column-3:
  If (mark(?Player ?Game ?Mark)
      board(?Game zero two ?Mark)
      board(?Game one two ?Mark)
      board(?Game two two ?Mark))
  then (wins-game(?Player ?Game)
        game-over(?Game))

Test-diagonal-1:
  If (mark(?Player ?Game ?Mark)
      board(?Game zero zero ?Mark)
      board(?Game one one ?Mark)
      board(?Game two two ?Mark))
  then (wins-game(?Player ?Game)
        game-over(?Game))

Test-diagonal-2:
  If (mark(?Player ?Game ?Mark)
      board(?Game zero two ?Mark)
      board(?Game one one ?Mark)
```

```
      board(?Game two zero ?Mark))
  then (wins-game(?Player ?Game)
        game-over(?Game))

Test-tie-game:
  If (board(?Game zero zero ?Any-mark1)
      board(?Game zero one ?Any-mark2)
      board(?Game zero two ?Any-mark3)
      board(?Game one zero ?Any-mark4)
      board(?Game one one ?Any-mark5)
      board(?Game one two ?Any-mark6)
      board(?Game two zero ?Any-mark7)
      board(?Game two one ?Any-mark8)
      board(?Game two two ?Any-mark9))
  then (tie-game(?Game)
        game-over(?Game))

Print-message-tie:
  If tie-game(?Game)
  then (print("~2%Game ~A ends in a tie." (?Game)))

Print-message-win:
  If wins-game(?Player ?Game)
  then (print("~2%Game ~A is won by ~A." (?Game ?Player)))

Print-message-over:
  If game-over(?Game)
  then (print("~2%Game ~A is over." (?Game)))
```

A.2 Knowledge Base for Player A

```
;;*** Class and Instance Declarations ***

assert-instance(class player)
assert-subclass(agent player)
assert-instance(player PlayerA PlayerB)
```

```
assert-instance(agent referee)
assert-instance(class game)
assert-instance(class nought-or-cross)
assert-instance(nought-or-cross O X)
assert-instance(class index)
assert-instance(index zero one two)
assert-instance(class agent-capability)
assert-instance(agent-capability tic-tac-toe-playing)

;;*** Relation Declarations ***

assert-instance(relation mod1)
argument-types(mod1 (index index))

assert-instance(relation mod2)
argument-types(mod2 (index index))

assert-instance(relation expertise)
argument-types(expertise (agent agent-capability))
specificity-ordered(expertise)

assert-instance(relation awake)
argument-types(awake (agent symbol))
single-valued(awake)
specificity-ordered(awake)

assert-instance(relation abnormal)
argument-types(abnormal (agent))

assert-instance(relation play-tic-tac-toe)
argument-types(play-tic-tac-toe (player player game))

assert-instance(relation tic-tac-toe)
argument-types(tic-tac-toe (player player game))

assert-instance(relation mark)
argument-types(mark (player game nought-or-cross))
```

```
single-valued(mark)
specificity-ordered(mark)

assert-instance(relation prefer-mark)
argument-types(prefer-mark (player nought-or-cross))
single-valued(prefer-mark)
specificity-ordered(prefer-mark)

assert-instance(relation opposite-mark)
argument-types(opposite-mark (nought-or-cross nought-or-cross))
single-valued(opposite-mark)
specificity-ordered(opposite-mark)

assert-instance(relation other-player)
argument-types(other-player (player player game))
single-valued(other-player)
specificity-ordered(other-player)

assert-instance(relation board)
argument-types(board (game index index nought-or-cross))
specificity-ordered(board)

assert-instance(relation game-over)
argument-types(game-over (game))
specificity-ordered(game-over)

assert-instance(relation tie-game)
argument-types(tie-game (game))
specificity-ordered(tie-game)

assert-instance(relation wins-game)
argument-types(wins-game (player game))
specificity-ordered(wins-game)

assert-instance(relation move)
argument-types(move (player index index game))
specificity-ordered(move)
```

```
;;*** End of Relation Declarations ***

        mod1(zero one)
        mod1(one two)
        mod1(two zero)

        mod2(zero two)
        mod2(one zero)
        mod2(two one)

expertise(referee tic-tac-toe-playing)
opposite-mark(X O)
opposite-mark(O X)

Infer awake(?Agent Yes)
from unless(abnormal(?Agent))

;;;;;;;;;;;;;; Rules that Generate Suggested Moves ;;;;

Block-row:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent ?Game ?Mark)
        board(?Game ?Row ?Col1 ?Mark)
        board(?Game ?Row ?Col2 ?Mark)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Block-column:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent ?Game ?Mark)
        board(?Game ?Row1 ?Col ?Mark)
        board(?Game ?Row2 ?Col ?Mark)
        mod1(?Row ?Row1)
```

```
        mod2(?Row ?Row2)
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Block-principal-diagonal:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent ?Game ?Mark)
        board(?Game ?Row1 ?Col1 ?Mark)
        board(?Game ?Row2 ?Col2 ?Mark)
        mod1(?Row ?Row1)
        mod2(?Row ?Row2)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Block-anti-principal-diagonal:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent ?Game ?Mark)
        board(?Game ?Row1 ?Col2 ?Mark)
        board(?Game ?Row2 ?Col1 ?Mark)
        mod1(?Row ?Row1)
        mod2(?Row ?Row2)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Try-row-next-own:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent  ?Game ?Opponent-mark)
        mark(?Player ?Game ?Mark)
        board(?Game ?Row ?Col1 ?Mark)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col2 ?Opponent-mark)))
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))
```

```
Try-row-next-to-next-own:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent  ?Game ?Opponent-mark)
        mark(?Player ?Game ?Mark)
        board(?Game ?Row ?Col2 ?Mark)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col1 ?Opponent-mark)))
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Try-column-next-own:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent  ?Game ?Opponent-mark)
        mark(?Player ?Game ?Mark)
        board(?Game ?Row1 ?Col ?Mark)
        mod1(?Row ?Row1)
        mod2(?Row ?Row2)
        excuse(unless(board(?Game ?Row2 ?Col ?Opponent-mark)))
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Try-column-next-to-next-own:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent  ?Game ?Opponent-mark)
        mark(?Player ?Game ?Mark)
        board(?Game ?Row2 ?Col ?Mark)
        mod1(?Row ?Row1)
        mod2(?Row ?Row2)
        excuse(unless(board(?Game ?Row1 ?Col ?Opponent-mark)))
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Select-center:
  Infer move(?Player one one ?Game)
  from (excuse(unless(board(?Game one one ?Any-mark))))
```

```
Select-corner-00:
  Infer move(?Player zero zero ?Game)
  from (excuse(unless(board(?Game zero zero ?Any-mark))))

Select-corner-02:
  Infer move(?Player zero two ?Game)
  from (excuse(unless(board(?Game zero two ?Any-mark))))

Select-corner-20:
  Infer move(?Player two zero ?Game)
  from (excuse(unless(board(?Game two zero ?Any-mark))))

Select-corner-22:
  Infer move(?Player two two ?Game)
  from (excuse(unless(board(?Game two two ?Any-mark))))

Select-middle-01:
  Infer move(?Player zero one ?Game)
  from (excuse(unless(board(?Game zero one ?Any-mark))))

Select-middle-10:
  Infer move(?Player one zero ?Game)
  from (excuse(unless(board(?Game one zero ?Any-mark))))

Select-middle-12:
  Infer move(?Player one two ?Game)
  from (excuse(unless(board(?Game one two ?Any-mark))))

Select-middle-21:
  Infer move(?Player two one ?Game)
  from (excuse(unless(board(?Game two one ?Any-mark))))

assert-instance(game game1)
prefer-mark(PlayerA X)
```

A.3 Knowledge Base for Player B

```
;;*** Class and Instance Declarations ***

assert-instance(class player)
assert-subclass(agent player)
assert-instance(player PlayerA PlayerB)
assert-instance(agent referee)
assert-instance(class game)
assert-instance(class nought-or-cross)
assert-instance(nought-or-cross O X)
assert-instance(class index)
assert-instance(index zero one two)
assert-instance(class agent-capability)
assert-instance(agent-capability tic-tac-toe-playing)

;;*** Relation Declarations ***

assert-instance(relation mod1)
argument-types(mod1 (index index))

assert-instance(relation mod2)
argument-types(mod2 (index index))

assert-instance(relation expertise)
argument-types(expertise (agent agent-capability))
specificity-ordered(expertise)

assert-instance(relation awake)
argument-types(awake (agent symbol))
single-valued(awake)
specificity-ordered(awake)

assert-instance(relation abnormal)
argument-types(abnormal (agent))

assert-instance(relation play-tic-tac-toe)
```

```
argument-types(play-tic-tac-toe (player player game))

assert-instance(relation tic-tac-toe)
argument-types(tic-tac-toe (player player game))

assert-instance(relation mark)
argument-types(mark (player game nought-or-cross))
single-valued(mark)
specificity-ordered(mark)

assert-instance(relation prefer-mark)
argument-types(prefer-mark (player nought-or-cross))
single-valued(prefer-mark)
specificity-ordered(prefer-mark)

assert-instance(relation opposite-mark)
argument-types(opposite-mark (nought-or-cross nought-or-cross))
single-valued(opposite-mark)
specificity-ordered(opposite-mark)

assert-instance(relation other-player)
argument-types(other-player (player player game))
single-valued(other-player)
specificity-ordered(other-player)

assert-instance(relation board)
argument-types(board (game index index nought-or-cross))
specificity-ordered(board)

assert-instance(relation game-over)
argument-types(game-over (game))
specificity-ordered(game-over)

assert-instance(relation tie-game)
argument-types(tie-game (game))
specificity-ordered(tie-game)
```

```
assert-instance(relation wins-game)
argument-types(wins-game (player game))
specificity-ordered(wins-game)

assert-instance(relation move)
argument-types(move (player index index game))
specificity-ordered(move)

;;*** End of Relation Declarations ***

        mod1(zero one)
        mod1(one two)
        mod1(two zero)

        mod2(zero two)
        mod2(one zero)
        mod2(two one)

expertise(referee tic-tac-toe-playing)
opposite-mark(X O)
opposite-mark(O X)

Infer awake(?Agent Yes)
from unless(abnormal(?Agent))

;;;;;;;;;;;;;; Rules that Generate Suggested Moves ;;;;

Block-row:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent ?Game ?Mark)
        board(?Game ?Row ?Col1 ?Mark)
        board(?Game ?Row ?Col2 ?Mark)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))
```

```
Block-column:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent ?Game ?Mark)
        board(?Game ?Row1 ?Col ?Mark)
        board(?Game ?Row2 ?Col ?Mark)
        mod1(?Row ?Row1)
        mod2(?Row ?Row2)
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Block-principal-diagonal:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent ?Game ?Mark)
        board(?Game ?Row1 ?Col1 ?Mark)
        board(?Game ?Row2 ?Col2 ?Mark)
        mod1(?Row ?Row1)
        mod2(?Row ?Row2)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Block-anti-principal-diagonal:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent ?Game ?Mark)
        board(?Game ?Row1 ?Col2 ?Mark)
        board(?Game ?Row2 ?Col1 ?Mark)
        mod1(?Row ?Row1)
        mod2(?Row ?Row2)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Try-row-next-own:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
```

```
        mark(?Opponent  ?Game ?Opponent-mark)
        mark(?Player ?Game ?Mark)
        board(?Game ?Row ?Col1 ?Mark)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col2 ?Opponent-mark)))
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Try-row-next-to-next-own:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent  ?Game ?Opponent-mark)
        mark(?Player ?Game ?Mark)
        board(?Game ?Row ?Col2 ?Mark)
        mod1(?Col ?Col1)
        mod2(?Col ?Col2)
        excuse(unless(board(?Game ?Row ?Col1 ?Opponent-mark)))
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Try-column-next-own:
  Infer move(?Player ?Row ?Col ?Game)
  from (other-player(?Player ?Opponent ?Game)
        mark(?Opponent  ?Game ?Opponent-mark)
        mark(?Player ?Game ?Mark)
        board(?Game ?Row1 ?Col ?Mark)
        mod1(?Row ?Row1)
        mod2(?Row ?Row2)
        excuse(unless(board(?Game ?Row2 ?Col ?Opponent-mark)))
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Try-column-next-to-next-own:
   Infer move(?Player ?Row ?Col ?Game)
   from (other-player(?Player ?Opponent ?Game)
         mark(?Opponent  ?Game ?Opponent-mark)
         mark(?Player ?Game ?Mark)
         board(?Game ?Row2 ?Col ?Mark)
         mod1(?Row ?Row1)
```

```
        mod2(?Row ?Row2)
        excuse(unless(board(?Game ?Row1 ?Col ?Opponent-mark)))
        excuse(unless(board(?Game ?Row ?Col ?Any-mark))))

Select-center:
  Infer move(?Player one one ?Game)
  from (excuse(unless(board(?Game one one ?Any-mark))))

Select-corner-00:
  Infer move(?Player zero zero ?Game)
  from (excuse(unless(board(?Game zero zero ?Any-mark))))

Select-corner-02:
  Infer move(?Player zero two ?Game)
  from (excuse(unless(board(?Game zero two ?Any-mark))))

Select-corner-20:
  Infer move(?Player two zero ?Game)
  from (excuse(unless(board(?Game two zero ?Any-mark))))

Select-corner-22:
  Infer move(?Player two two ?Game)
  from (excuse(unless(board(?Game two two ?Any-mark))))

Select-middle-01:
  Infer move(?Player zero one ?Game)
  from (excuse(unless(board(?Game zero one ?Any-mark))))

Select-middle-10:
  Infer move(?Player one zero ?Game)
  from (excuse(unless(board(?Game one zero ?Any-mark))))

Select-middle-12:
  Infer move(?Player one two ?Game)
  from (excuse(unless(board(?Game one two ?Any-mark))))

Select-middle-21:
```

```
  Infer move(?Player two one ?Game)
  from (excuse(unless(board(?Game two one ?Any-mark))))

assert-instance(game game1)
prefer-mark(PlayerB X)
```

A.4 Playing a Game

The game is played by creating three *RAD* agents, named *Referee*, *PlayerA*, and *PlayerB*, (arbitrarily) all connected to the same ESS. An interface for the human observer is created by

```
raduser -a referee -e delphi -u observer
```

which causes the response

```
                RAD User Interface

  Copyright 1992 by Microelectronics and Computer
              Technology Corporation
           Proprietary and Confidential
               MCC Carnot Project

Connecting to ESS on Delphi+17001...
ESS: Aide observer-aide is ready.
Interface: pinging agent referee...
referee: READY
observer:
```

The appropriate knowledge bases should now be loaded into each agent by contacting them and issuing consult commands:

```
observer: consult "referee-kb.rad"
referee:  Consulting referee-kb.rad
Finished consulting referee-kb.rad

observer: contact playera
playera: READY
```

```
observer: consult "playera-kb.rad"
playera: Consulting playera-kb.rad
playera: Finished consulting playera-kb.rad

observer: contact playerb
playerb: READY

observer: consult "playerb-kb.rad"
playerb: Consulting playerb-kb.rad
playerb: Finished consulting playerb-kb.rad
```

The *Referee* is now instructed to begin the game, and the results are shown to the *Observer*.

```
observer: contact referee
referee: READY

observer: play-tic-tac-toe(playera playerb game1)
referee: Assertion PLAY-TIC-TAC-TOE-1 accepted.
Move 1:  square 1 1 is occupied by PLAYERA
Move 2:  square 1 1 is occupied by PLAYERB
Move 3:  square 1 1 is occupied by PLAYERA
Move 4:  square 1 1 is occupied by PLAYERB
Move 5:  square 1 1 is occupied by PLAYERA
Move 6:  square 1 1 is occupied by PLAYERB
Move 7:  square 1 1 is occupied by PLAYERA
Move 8:  square 1 1 is occupied by PLAYERB
Move 9:  square 1 1 is occupied by PLAYERA

Game GAME1 ends in a tie.
Game GAME1 is over.
```

Appendix B

Resolving Contradictions

B.1 Simple Example of CRM

The following is the contents of a file that *RAD* can consult for a simple example of contradiction resolution:

```
;;; Create class of elephants
assert-instance (class elephant)

;;; Make an attribute of color:
assert-instance (relation color)          ; Make color a relation
argument-types(color (elephant symbol)) ; Specify domain, range
specificity-ordered(color)                ; Enable inheritance
single-valued(color)                      ; Make it single-valued

assert color(?X:elephant grey)            ; Provide default value

;;; Domain Rules, Assertions

;; Forward rule about albinoism
if albino(?X) then color(?X white)

if (says(?X ?Y) unless(liar(?X))) then assert(?Y)

Prefer(liar(?X) contradiction())
```

```
if (color(?X:elephant ?Y) bad-color(?X ?Y)) then contradiction()

;;; Situation-Specific Knowledge

assert-instance (elephant clyde)

says(joe albino(clyde))
```

RAD will exhibit the following behavior when the knowledge in this file is consulted by an agent named advisor and accessed by a user named doctor:

```
doctor: consult "clyde.rad"
advisor: Consulting clyde.rad
advisor: Finished consulting clyde.rad

doctor: why color-3
advisor: Instance Slot Value COLOR-3    (IN)
  COLOR(CLYDE WHITE)
was derived from the following:

Forward Rule FRULE-0    (IN)
  If
   ALBINO(?X)
  then
   COLOR(?X WHITE)

Assertion ALBINO-1    (IN)
  ALBINO(CLYDE)

doctor: why albino-1
advisor: Assertion ALBINO-1    (IN)
  ALBINO(CLYDE)
was derived from the following:

Forward Rule FRULE-1    (IN)
  If
```

```
  (
   SAYS(?X ?Y)
   /UNLESS/0(?X PI::*ARG-END-SYM* LIAR)
  )
  then
   ASSERT(?Y)

Assertion SAYS-1     (IN)
  SAYS(JOE ALBINO(CLYDE))

Failed Goal LIAR-1     (OUT)
  LIAR(JOE)

doctor: contradict color-3
advisor: CONTRADICTION-0

RESOLVED by justifying the following:
LIAR-2

doctor: why liar-2
advisor: Assertion LIAR-2     (IN)
  LIAR(JOE)
resolves contradictions:
CONTRADICTION-0
when it is justified by the following:

Forward Rule FRULE-1     (IN)
  If
  (
   SAYS(?X ?Y)
   /UNLESS/0(?X PI::*ARG-END-SYM* LIAR)
  )
  then
   ASSERT(?Y)

Assertion SAYS-1     (IN)
  SAYS(JOE ALBINO(CLYDE))
```

```
Forward Rule FRULE-0     (IN)
  If
   ALBINO(?X)
  then
   COLOR(?X WHITE)

doctor:
```

B.2 Complex Example of CRM

B.2.1 Knowledge Base for Example of CRM

The following is the consult file for the example discussed in Chapter 8:

```
;;; Domain Rules, Assertions
assert-instance(relation
                test-error mistaken-observation result)

If Symptom(?Patient dehydrated)
then Conclusion(?Patient low amt H2O)

If (Conclusion(?Patient low amt H2O)
    Remember(Normal(?Patient amt Na)))
then Conclusion(?Patient high conc Na)

Infer Normal(?Patient amt Na)
from Unless(Result(?Patient low amt Na))

If (Lab-test(?Patient low ?Meas ?Chem)
    Conclusion(?Patient high ?Meas ?Chem))
then contradiction()

;;;  Metaknowledge
Fix(Lab-test(?Patient ?Degree ?Meas ?Chem)
    Lab-test(?Patient ?Degree ?Meas ?Chem)
    Test-error(?Patient ?Degree ?Meas ?Chem))
```

```
Fix(Conclusion(?Patient ?Degree ?Meas ?Chem)
    Symptom(?Patient ?Symptom)
    Mistaken-observation(?Patient ?Symptom))

Prefer(Mistaken-observation(?Patient ?Symptom)
       Test-error(?Patient ?Degree ?Meas ?Chem))

Defeat(Lab-test(?Patient ?Degree ?Meas ?Chem)
      none
      Test-error(?Patient ?Degree ?Meas ?Chem))

Defeat(Symptom(?Patient ?Symptom)
       none
       Mistaken-observation(?Patient ?Symptom))

Prefer(Mistaken-observation(?Patient ?Symptom) contradiction())
Prefer(Test-error(?Patient ?Degree ?Meas ?Chem) contradiction())
Prefer(Result(?Patient low amt ?Chem) contradiction())
```

B.2.2 Protocol for Use of CRM

The following is the protocol using the consult file above. There are three agents. Two are *radusers*: the doctor and the lab. Agent "advisor" is a *radagent*. The consult file name is "sodium.rad".

```
doctor: contact advisor
advisor: READY
doctor: consult "sodium.rad"
advisor: Consulting sodium.rad
advisor: Finished consulting sodium.rad
doctor: symptom(Jane dehydrated)
doctor: show conclusion
advisor: CONCLUSION
   Instance of:          RELATION

   ARGUMENT-TYPES:       none
```

```
   SPECIFICITY-ORDERED:  no
   SINGLE-VALUED:        no
   PRIMITIVE:            no
   SYSTEM:               no
   PERSISTENT:           no
   UNIDIRECTIONAL:       none
   AXIOMATIC:            no
   SPECIALIZATION:       none
User Relation CONCLUSION (quaternary)
Current assertions:
CONCLUSION-1:  CONCLUSION(JANE LOW AMT H2O)
CONCLUSION-2:  CONCLUSION(JANE HIGH CONC NA)

doctor: why conclusion-2
advisor: Assertion CONCLUSION-2    (IN)
  CONCLUSION(JANE HIGH CONC NA)
was derived from the following:

Forward Rule FRULE-1    (IN)
  If
  (
   CONCLUSION(?PATIENT LOW AMT H2O)
   REMEMBER(NORMAL(?PATIENT AMT NA))
  )
  then
   CONCLUSION(?PATIENT HIGH CONC NA)

Assertion NORMAL-2    (IN)
  NORMAL(JANE AMT NA)

Assertion CONCLUSION-1    (IN)
  CONCLUSION(JANE LOW AMT H2O)
```

At this point, the doctor has used the advisor as a simple expert system. By asserting a symptom, the rules have drawn conclusions which can be examined by the doctor. However, since the system is distributed, the lab may add its own data about the case.

```
lab: lab-test(Jane low conc Na)
advisor: CONTRADICTION-0
RESOLVED by justifying the following:
MISTAKEN-OBSERVATION-1
```

When the lab added conflicting information, a contradiction was produced. Because the consult file used did not include query schemata, no one was asked about this possibility. But the metaknowledge included was sufficient for the advisor to treat this as the most likely possibility and justify it.

However, this now comes to the attention of the doctor who disagrees that he may not have properly observed Jane's symptoms. This ability to disagree is an important part of *RAD*'s functionality.[1]

```
doctor: show mistaken-observation-1

advisor: Assertion MISTAKEN-OBSERVATION-1     (IN)
  MISTAKEN-OBSERVATION(JANE DEHYDRATED)

doctor: contradict mistaken-observation-1
advisor: CONTRADICTION-1
RESOLVED by justifying the following:
TEST-ERROR-1
```

The doctor finds this resolution likely. However, the lab does not believe this is a possibility and disagrees...

```
lab: show test-error-1
advisor: Assertion TEST-ERROR-1     (IN)
  TEST-ERROR(JANE LOW CONC NA)

lab: contradict test-error-1
advisor: CONTRADICTION-2
RESOLVED by justifying the following:
RESULT-2
       -------
doctor: show result-2
```

[1]The doctor may also have disagreed by commanding **wrong** after a query about mistaken observations.

```
advisor: Assertion RESULT-2     (IN)
  RESULT(JANE LOW AMT NA)
```

The advisor has now offered another possibility: namely that Jane has an unusually low amount of sodium in her blood. Only this would explain why she was dehydrated but has a low blood concentration of sodium. This was a somewhat uncommon possiblity discovered by backtracking. (The rule writer encoded this possibility as an assumption of normality.)

However, this result is still itself only an assumption. The doctor must still find out why Jane's sodium level is low. (It turns out she was radiation poisoned, which also accounts for the dehydration symptoms.) While pursuing this possibility, the doctor may want to remember the reasoning behind it. In the current version of *RAD*, this is done by a full replay of the justification for the assertion.

```
doctor: why result-2
advisor: Assertion RESULT-2     (IN)
  RESULT(JANE LOW AMT NA)

 resolves contradictions:
CONTRADICTION-2
CONTRADICTION-1
CONTRADICTION-0

when it is justified by the following:

Backward Rule NORMAL-1     (IN)
  Infer
   NORMAL(?PATIENT AMT NA)
  from
   /UNLESS/0(?PATIENT LOW AMT NA PI::*ARG-END-SYM* RESULT)
```

```
Forward Rule FRULE-0    (IN)
  If
   SYMPTOM(?PATIENT DEHYDRATED)
  then
   CONCLUSION(?PATIENT LOW AMT H2O)

Forward Rule FRULE-1    (IN)
  If
  (
   CONCLUSION(?PATIENT LOW AMT H2O)
   REMEMBER(NORMAL(?PATIENT AMT NA))
  )
  then
   CONCLUSION(?PATIENT HIGH CONC NA)

Forward Rule FRULE-2    (IN)
  If
  (
   LAB-TEST(?PATIENT LOW ?MEAS ?CHEM)
   CONCLUSION(?PATIENT HIGH ?MEAS ?CHEM)
  )
  then
   CONTRADICTION()

doctor: quit
```

The information in this explanation amounts to the fact that two other possibilities, explaining conflicting data, were rejected and this was the remaining explanation. The doctor can see the other possibilities by commanding **why** on the contradictions listed. Eventually, a new justification, based on the discovery of radiation poisoning, can be established for this assertion.

Bibliography

[Proteus 1989] Natraj Arni, et al., "Proteus 3: A System Description," MCC Technical Report No. ACT-AI-226-89-Q, Microelectronics and Computer Technology Corporation, Austin, TX, June 1989.

[Aït-kaci, *et al.* 1985] Hassan Aït-kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr, "Efficient Implementation of Lattice Operations," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, January 1989, pp. 115–146.

[Bond and Gasser 1988] Alan H. Bond and Les Gasser, *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[Bridgeland 1989] David M. Bridgeland, "Extending the Warren Abstract Machine for an Expert System Shell," presented at *AAAI Spring Symposium on Automated Theorem Proving*, Stanford, CA, March 1989.

[Charniak *et al.* 1980] E. Charniak, C. K. Riesbeck, and D. V. McDermott, *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1980.

[Clocksin and Mellish 1981] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin Heidelberg, 1981.

[Davis and Smith 1983] Randall Davis and Reid G. Smith, "Negotiation as a Metaphor for Distributed Problem solving," *Artificial Intelligence*, vol. 20, no. 1, January 1983, pp. 63–109.

[Doyle 1979] Jon Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, no. 3, 1979, pp. 231–272.

[Feigenbaum 1988] Edward A. Feigenbaum, Pamela McCorduck, and H. Penny Nii, *The Rise of the Expert Company*, Times Books, New York, 1988.

[Gasser *et al.* 1987] Les Gasser, Carl Braganza, and Nava Herman, "Implementing Distributed Artificial Intelligence Systems Using MACE," *Proceedings of the Third IEEE Conference on Artificial Intelligence Applications*, 1987, pp. 315–320.

[Gasser and Huhns 1989] Les Gasser and Michael N. Huhns, eds., *Distributed Artificial Intelligence, Volume II*, Pitman Publishing, London, 1989.

[Goodwin 1984] J. W. Goodwin, "WATSON: A Dependency Directed Inference System," Research Report LiTH-IDA-R-84-10, Computer and Information Science Department, Linkoping University, 1984.

[Harp and Sederberg 1988] Steven A. Harp and John C. Sederberg, "Using Truth Maintenance to Do Configuration," *Proc. Fourth IEEE Conference on Artificial Intelligence Applications*, San Diego, CA, March 1988, pp. 393–394.

[Hsu *et al.* 1987] Ching-Chi Hsu, Shao-Ming Wu, and Jan-Jan Wu, "A Distributed Approach for Inferring Production Systems," *Proceedings IJCAI-87*, Milan, Italy, August 1987, pp. 62–67.

[Huhns and Bridgeland 1991] Michael N. Huhns and David M. Bridgeland, "Multiagent Truth Maintenance," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 6, December 1991.

[Huhns and Acosta 1988] Michael N. Huhns and Ramon D. Acosta, "Argo: A System for Design by Analogy," *IEEE Expert*, Fall 1988, pp. 53–68.

[Huhns 1987] Michael N. Huhns, ed., *Distributed Artificial Intelligence*, Pitman Publishing, London, 1987.

[Kodak 1988] M. R. Hoffmann, "KIMS System Configurer V1.0—Knowledge Documentation," Kodak Technical Report, December 1988.

[Kirchen 1989] Daniel Kirchen, "The Power Restoration Advisor," CDC Technical Report (unpublished), 1989.

[Mason and Johnson 1989] Cindy L. Mason and R. R. Johnson, "DATMS: A Framework for Distributed Assumption Based Reasoning," in [Gasser and Huhns 1989], pp. 293–317.

[Pierce] C. S. Pierce, *Scientific Metaphysics*, Vol. VI, p. 358.

[Petrie *et al.* 1986] Charles J. Petrie, David M. Russinoff, and Donald D. Steiner, "Proteus: A Default Reasoning Perspective," *Proceedings 5th Generation Conference*, National Institute for Software, October 1986.

[Petrie 1986] Charles J. Petrie, "Extended Contradiction Resolution," MCC Technical Report No. ACA-AI-102-86, Microelectronics and Computer Technology Corporation, Austin, TX, March 1986.

[Petrie 1987] Charles J. Petrie, "Revised Dependency-Directed Backtracking for Default Reasoning," *Proceedings of AAAI-87*, Seattle, WA, July 1987, pp. 167–172.

[Petrie 1989] Charles J. Petrie, "Reason Maintenance in Expert Systems," MCC Technical Report No. ACA-AI-021-89, Microelectronics and Computer Technology Corporation, Austin, TX, February 1989.

[Russinoff 1985] David M. Russinoff, "An Algorithm for Truth Maintenance," MCC Technical Report No. AI-062-85, Microelectronics and Computer Technology Corporation, Austin, TX, April 1985.

[Smith 1980] Reid G. Smith, "The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver," *IEEE Transactions on Computers*, vol. C-29, no. 12, December 1980, pp. 1104–1113.

[Smith and Davis 1981] Reid G. Smith and Randall Davis, "Frameworks for Cooperation in Distributed Problem Solving," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-11, no. 1, January 1981, pp. 61–70.

[Steele 1989] Robin L. Steele, Scott A. Richardson, and Michael A. Winchell, "Design Advisor: A Knowledge-Based Integrated Circuit Design Critic," in Herb Schorr and Alain Rappaport, eds., *Innovative Applications of Artificial Intelligence*, AAAI Press, 1989.

[Tomlinson *et al.* 1991] Chris Tomlinson, Mark Scheevel, and Vineet Singh, "Report on Rosette 1.1," MCC Technical Report No. ACT-OODS-275-91, Microelectronics and Computer Technology Corporation, Austin, TX, July 1991.

[UNIX] "UNIX Programmers Manual," University of California, Berkeley, CA, 1984.

[Virdhagriswaran *et al.* 1987] Sankar Virdhagriswaran, Sam Levine, Scott Fast, and Susan Pitts, "PLEX: A Knowledge-Based Placement Program for Printed Wire Boards," *Proc. Third IEEE Conference on Artificial Intelligence Applications*, February 1987, pp. 302–305.

[Virdhagriswaran and Pitts 1987] Sankar Virdhagriswaran and Susan Pitts, "MINC: A Deniable Expert System that Reasons with Simplifying Assumptions," *Proc. SPIE Applications of Artificial Intelligence V*, Vol. 786, 1987, pp. 38–40.

[Warren 1983] David H. D. Warren, "An Abstract Prolog Instruction Set," SRI Technical Note 309, SRI International, October 1983.

Index