VIVA Knowledge-Based Agent Programming

Gerd Wagner* (gw@informatik.uni-leipzig.de)
Institut für Informatik, Universität Leipzig,
Augustusplatz 10-11, D-04109 Leipzig, Germany

Preliminary Report   October 4, 1996

Abstract

We sketch VIVA, a new agent-oriented programming language based on the theory of VIVid Agents presented in [Wag96a]. VIVA follows the AOP paradigm of [Sho93] but is, in a sense, more conservative by adopting as many concepts as possible from SQL and Prolog. This concerns, e.g., the distinction between the schema and the state of an agent, or the use of facts and rules with negation-as-failure, logical variables and unification. The basic design principles of VIVA are conservativity, scalability, and versatility.

Contents

*http://www.informatik.uni-leipzig.de/~gwagner/home.html

1 Introduction

A *vivid agent* is a software-controlled system whose state is expressed in terms of mental components, such as beliefs and intentions, and whose behavior is represented by means of *action* and *reaction rules*. The basic functionality of a vivid agent comprises an update and an inference operation, and the possibility to represent and perform actions in order to be able to generate and execute plans. Since a vivid agent is 'situated' in an environment with which it has to be able to communicate, it also needs the ability to react in response to perception events, and in response to communication events created by the communication acts of other agents. In this paper, we propose a declarative programming language, called VIVA, which is based on the concept of vivid agents [Wag96a]. The language is intended for general purpose agent-oriented programming of software agents, embedded systems, and robots.

VIVA does not restrict the agent designer to a fixed logical system for the knowledge base of an agent. Rather, it allows to choose a suitable knowledge system for each agent individually according to its domain and its tasks. In simple cases, a relational database-like system (admiting only of atomic facts) will do the job, while in more involved cases one might need the ability to process, in addition to simple facts, uncertain, temporal or confidential information, or even such advanced capabilities as deductive query answering based on rule knowledge and inconsistency handling. The knowledge system of a vivid agent will be nonmonotonic, since one needs the Closed-World Assumption (CWA), and negation-as-failure, in any practical system (like in relational and deductive databases).

The combination of a structured knowledge base (representing the mental state of an agent) with action and reaction rules yields an *executable specification* of an agent, resp. of a multi-agent system. This is similar to the idea of PROgramming in LOGic where programs have both a procedural and a declarative reading. We consider VIVA as a multi-agent logic programming language, therefore. Notice, however, that there are three kinds of rules in VIVA: deduction, action and reaction rules, whereas a Prolog program only contains deduction rules. We could also characterize VIVA as a *knowledge-based* programming language, since the state of a VIVA program, refered to by rule conditions, does not just consist of the current values of program variables but is composed of several substates including the belief state of the programmed agent. Finally, VIVA can be viewed as a *coordination language* in the sense of [Weg96], since it separates computation (achieved by deduction rules) from real-time coordination (achieved by reaction rules and the underlying message passing mechanisms). According to Wegner, coordination is constrained interaction among communicating software components.

In this paper, we will not treat issues of open multi-agent systems. Rather, we will restrict our considerations to closed systems where an agent knows the other agents if there are any, and where we have a common language for inter-agent communication. It is an issue for future work to extend VIVA by adding features which address the specific requirements of open systems.

2 Basic Requirements for Knowledge-Based Agent Programming

We discuss three basic requirements for a knowledge-based agent programming language: versatility, scalability and conservativity. In addition, we consider a number of further desiderata below, among them extensibility, schema and behavior inheritance, and online modification.

2.1 Versatility

A general-purpose agent-oriented programming language should allow to program agents for all kinds of domains where different types of information, such as temporal, uncertain, or confidential information, may occur. The language must therefore provide appropriate constructs to define such *qualified* predicates in addition to simple predicates.

In order to enable the agent to reason with qualified information adequately the interpreter must support temporal, uncertain, and secure inference according to appropriate logics. Therefore, knowledge-based agent programming is also 'multi-logic programming' involving multiple logics. Since qualifications may be used in the expression of causal relations, generic laws, and conceptual subsumption, the programming language must allow to define deduction rules with qualified predicates, and the interpreter must be able to evaluate such rules.

2.2 Scalability

As with natural agents (such as humans), not every artificial agent needs to be able to do sophisticated reasoning. In many cases, inference will only consist of a Boolean combination of various table lookups as in SQL databases.

A programmer should not be required to know the more advanced features of an agent programming language if she only needs basic level constructs. The knowledge processing of a simple agent based on a relational database-like knowledge system should not be affected by the availability of more advanced features like, e.g., possibilistic rule evaluation. In other words, the simplicity and efficiency of a program for a simple agent should not be affected by the overall possibilities and complexity of the programming language.

2.3 Conservativity

A new declarative programming language should build upon the already existing ones and adopt as many of their concepts as possible. This concerns, in particular, SQL and Prolog both of which are declarative languages which have been successfully applied to a wide range of information and knowledge processing problems. For both of them the use of the CWA and negation-as-failure is essential. In SQL, a negated query such as $p(x) \wedge \neg q(x)$ is answered by forming the set difference $P - Q$ of the resp. tables P and Q which corresponds exactly to negation-as-failure, or, logically speaking, to preferential entailment based on minimal models. But VIVA also borrows from other database paradigms, such as historical and temporal databases for processing temporally qualified information, fuzzy databases for processing uncertain information, and multilevel secure databases for processing confidential information.

3 Specifying and Executing Agents in VIVA

A VIVA programming system requires platform-specific code to configure and execute a multiagent system, and platform-independent VIVA source code to specify the participating agents. An agent specified by a piece of VIVA source code can run on a number of hosts with the same or different hardware/software architectures provided they can handle the events and actions of the agent. The composition of a multiagent system, and the locations of the participating agents, are details that must be specified at link and execution time, before a VIVA application can run.

A VIVA specification of an agent consists of three parts: schema definition, behavior definition, and initialization.

```
agent-specification = AGENT agent-name
                      schema-def
                      behavior-def
                      initialization
```

The schema definition determines the language of the knowledge base, i.e. the kind of predicates an agent uses to represent knowledge, and to reason, about its domain. The behavior definition consists of action and reaction rules. Finally, the initialization of an agent assigns initial beliefs and tasks.

When a VIVA agent performs a physical action, it calls an external procedure which may be a function of the operating system in the case of a software agent, or a function of an effector control process in the case of a robot. Although a VIVA program performs the high-level processing of perception data, and the high-level control of physical actions, it does not provide the low-level programming of sensors and effectors. This has to be done by means of conventional programming languages which seem to be more appropriate for this purpose than a decalarative language such as VIVA.

3.1 Reagents

Conceptually, a vivid reagent consists of three components:

1. a knowledge base collecting the beliefs of the reagent (also called its *epistemic state*),

2. an *event queue*, i.e. a buffer receiving messages from other agents or from perception subsystems running as concurrent processes,[1]

3. a set of *reaction rules* encoding the reactive and communicative behavior.

Reaction rules are triggered by events. The agent interpreter continuously checks the event queue of the agent. If there is a new event message, it is matched with the event condition of all reaction rules, and the epistemic conditions of those rules matching the event are evaluated. If they are satisfiable in the current knowledge state, all free variables in the rules are instantiated accordingly resulting in a set of triggered actions with associated epistemic effects. All these actions are then executed, leading to physical actions and to communicative actions (i.e. sending messages to other agents), and their epistemic effects are assimilated into the current knowledge base.

Informally, the VIVA execution model of a reagent consists of the following steps:

1. Get the next message from the event queue, and check whether it triggers any reaction rules. If it does not, then repeat 1, else continue.

2. For each of the triggered reaction rules, check its epistemic condition. If it holds, assimilate the epistemic effect of the triggered action into the knowledge base, and if it is

   (a) a physical action, execute it by calling the associated procedure.

   (b) a communicative action, execute it by sending the corresponding message to the specified addressee.

3. Continue with step 1.

3.2 Agents

A vivid agent consists of four components:

1. a *mental state* including its *beliefs* (or knowledge base), its *tasks* and its *current intentions*,

2. an *event queue*, i.e. a buffer receiving messages from other agents or from perception subsystems running as concurrent processes,

[1]Such a perception subsystem may be a UNIX process monitoring some part of the file system and informing the agent about relevant changes in the case of a software agent, or it may be a neural network-based process evaluating sensor data in the case of a robot.

3. a set of *action rules* representing the available action types which, together with the task list, is the basis for planning and plan execution, and

4. a set of *reaction rules* encoding the reactive and communicative behavior.

The execution of vivid agents consists of interleaving their reactive behavior (where incoming messages trigger reaction rules) with their pro-active behavior (where unsolved tasks require further planning and plan execution). The execution model of VIVA is defined in [Wag96a, Wag96c], where a formal transition system semantics for vivid agents is presented.

4 Schema Definition

The schema of an agent reflects the domain it has to deal with and the environment it is situated in since it determines the language by means of which the agent represents the domain and communicates with its environment. Similar to the definition of the database schema in SQL, the schema of an agent has to be defined in VIVA before any knowledge content can be assigned to the agent, and before any behavior can be specified for it. As opposed to SQL databases where the schema essentially consists only of predicate definitions (i.e. relation schemas) by means of CREATE TABLE statements, we have a more complex schema definition in the case of an agent.

First, we have various kinds of predicates expressing different types of information: (in)complete, uncertain, temporal, bitemporal, reliable and confidential.[2] For each predicate it should be possible to specify whether it is completely represented, i.e. the Closed-World Assumption (CWA) is applicable, or not. We will assume the CWA for a predicate if nothing else is specified.

Second, we have to define the schemas of communication and perception events. Third, action schemas have to be defined. Syntactically, an event or action schema has the same structure as a relation schema consisting of a name and a list of argument definitions.

In summary, we obtain the following BNF clause for the *agent schema* definition:

schema-def	=	pred-def
		evt-def
		act-def

For simplicity, we will assume that all agents of a multi-agent system share a common knowledge base schema, i.e. *pred-def*, while the event and action schemas may be different (although they will usually overlap).[3]

[2] This list may grow in future versions of VIVA.

[3] In a non-homogeneous multi-agent system, where agents have different knowledge base schemas, it will be necessary that each agent knows the schemas of all others in order to avoid 'Babylonian miscommunication'.

4.1 Defining Predicates

The definition of predicates determines the schema of the agent's KB. VIVA associates a certain knowledge system[4] with the specified KB schema. For instance, if an uncertain predicate is specified, VIVA uses a knowledge system based on *possibilistic* logic.[5] The KB schema also determines the input and the query language of the agent. While the input language defines what the agent can be told (i.e. what it is able to assimilate into its knowledge base), the query language defines what the agent can be asked.

Simple agents will not need deductive query answering based on rule knowledge. It is therefore an option to include deduction rules in the knowledge base of an agent. Deduction rules are used to express *intensional* predicates which may represent conceptual subsumption relationships, causal relations, or generic laws.

pred-def	=	EXTENSIONAL PREDICATES ext-pred-def*
pred-def	=	EXTENSIONAL PREDICATES ext-pred-def*
		INTENSIONAL PREDICATES int-pred-def*

4.1.1 Defining Extensional Predicates

A simple predicate is defined as in SQL, except that it is possible to declare the predicate to be incomplete, i.e. not to be subject to the CWA. For instance, the following VIVA code would define two predicates for the schema of a hospital information system agent:

```
CREATE TABLE patient(
PatientNo    CHAR(10) NOT NULL,
Name         CHAR(30) NOT NULL
);

CREATE INCOMPLETE TABLE illness(
PatientNo    CHAR(10) NOT NULL,
Disease      CHAR(20) NOT NULL,
StartDate    DATE,
EndDate      DATE
);
```

If an extensional predicate is not completely represented, i.e. not subject to the CWA, it will be represented by means of a *split-table*: the upper part containing its positive extension, and the lower part containing its negative extension (i.e. all instances for which it is known that the predicate does not hold). For

[4] See [Wag96a, Wag95].

[5] See [DLP94]. In future versions of VIVA, there may be a choice among different knowledge systems.

instance, the split-table ILLNESS/NONILLNESS for the incomplete predicate *illness* could have the following content:

$$\frac{\text{ILLNESS}}{\text{NONILLNESS}} = \begin{array}{|cccc|} 007 & \text{hepatitis B} & 01.07.93 & 15.08.93 \\ 007 & \text{influenca} & 29.01.96 & 10.02.96 \\ 194 & \text{malaria} & 01.03.95 & 01.04.95 \\ \hline 194 & \text{hepatitis B} & 29.01.96 & 29.01.96 \\ 007 & \text{HIV} & 01.03.95 & 01.04.95 \\ \hline \end{array}$$

while the table for the complete predicate *patient* (the hospital has a complete record of all people having ever been patients) could look like

$$\text{PATIENT} = \begin{array}{|cc|} \hline 191 & \text{Yeltsin} \\ 194 & \text{Rexrodt} \\ 007 & \text{Bond} \\ \hline \end{array}$$

ILLNESS represents positive, while NONILLNESS represents negative diagnoses. The predicate *illness* is declared to be incomplete because it cannot be concluded that a patient did not have a certain disease if there is no corresponding positive entry in the table. For instance, while we can infer $\neg patient(Clinton)$ simply because there is no corresponding entry in PATIENT, we cannot infer $\neg illness(007, \mathit{influenca}, 01.02.94)$, though this is not recorded in ILLNESS. For an incomplete predicate, like *illness*, we can only infer a negated sentence if there is a corresponding entry in its negative extension, i.e. in NONILLNESS. If we want to record negative information for the predicate *illness* we can do this by inserting resp. tuples in NONILLNESS.

In summary, we obtain the follwoing BNF clause for the definition of tables (extensional predicates):

ext-pred-def	=	CREATE [INCOMPLETE] [pred-type] TABLE pred-name(attr-def*)
pred-type	=	FUZZY | TEMPORAL | BITEMPORAL | MLS
pred-name	=	... *(some practical naming convention)*
attr-def	=	... *(like in SQL)*

The definition of more advanced types of predicates – qualified predicates such as fuzzy, (bi)temporal and MLS predicates, and intensional predicates – is described below.

4.1.2 Schema-Dependent Types

VIVA derives five implicit agent-specific types from the knowledge base schema definition:

`INPUT` is the set of all literals formed with extensional predicates.

`IFQUERY` is the set of all variable-free queries formed with (extensional and intensional) predicates from the knowledge base schema.

`QUERY` is the set of all queries formed with (extensional and intensional) predicates from the knowledge base schema.

`IFANSWER` is the set of all possible answers to if-queries. It contains, e.g., the answers *yes*, *no*, and *unknown*.

`ANSWER` is the set of all (possibly qualified) answer substitutions for answering queries with variables.

4.1.3 Integrity Constraints

VIVA allows to specify integrity constraints (ICs) in the form of if-queries as part of the knowledge base schema. ICs must not be violated by updates. The simplest kind of ICs are functional dependencies such as expressed by the UNIQUE attribute clause of SQL. For instance, the UNIQUE declaration in the predicate definition

```
CREATE TABLE turn( Elevator  CHAR(1) UNIQUE);
```

is equivalent to the if-query $\forall x \forall y : turn(x) \wedge turn(y) \supset x = y$. It expresses the condition that, at any moment, there can be only one elevator whose turn is next. In the presence of this IC, it is not necessary, for instance, first to delete $turn(a)$ before adding $turn(b)$, like in

$$\mathsf{Upd}(\{turn(a)\}, \neg turn(a) \wedge turn(b)) = \{turn(b)\}$$

but it suffices to update by the new input which overrides the old information:

$$\mathsf{Upd}(\{turn(a)\},\ turn(b)) = \{turn(b)\}$$

Integrity constraints are used in VIVA to guide updates.

In addition to UNIQUE declarations, we will use exclusive disjunctions in our examples below in order to express that a system can only be in exactly one state of a number of possible states. For instance, the integrity constraint

$$\exists x : at(x) | up(x) | down(x)$$

leads to updates such as

$$\mathsf{Upd}(\{at(1)\}, up(1)) = \{up(1)\}$$

4.2 Defining Event Schemas

VIVA makes a distinction between perception and communiction events. There may be agents which need not communicate with other agents, but only observe their environment and react adequately in response to certain events. For instance, a typical perception event of a robot would be

```
DEFINE PERCEPTION EVENT obstacleAhead( Distance  FLOAT)
```

In a similar way, communication events representing the inter-agent communication language are defined. E.g., a robot might have to request the permission of another robot in order to approach it:

```
DEFINE COMMUNICATION EVENT reqPermAppr()
```

VIVA provides a number of predefined communication events based on *speech act* theory. For instance, the schema of the communication act *tell* is predefined as

```
DEFINE COMMUNICATION EVENT tell( What  INPUT)
```

Recall that INPUT is a special VIVA type specifying all admissible knowledge inputs, i.e. those logical sentences which an agent can assimilate into its knowledge base. In addition to *Tell*, there are further predefined communication events including *askIf*, *askOne*, *askAll*, *reply*, *reqDo*, and *reqStit* (see below).

Thus, events are defined according to the following BNF clause:

evt-def	=	DEFINE PERCEPTION EVENT evt-name(attr-def*)
evt-def	=	DEFINE COMMUNICATION EVENT evt-name(attr-def*)
evt-name	=	... *(some practical naming convention)*

4.3 Defining Action Schemas

Depending on the kind of agent (e.g. software agent, embedded agent, or robot) and on its domain, there will be a certain repertoire of actions available to the agent (e.g. delete file, open door, turn right). Actions are defined by a name, a list of arguments and an associated external procedure call which realizes the action when it is performed by means of a DO statement. This requires a standardized call interface. E.g., the definition of a software agent might include

```
DEFINE ACTION deleteFile(
Path     CHAR(40),
File     CHAR(40)
)
REALIZED BY <some UNIX function>
```

Thus, actions are defined according to:

act-def	=	DEFINE ACTION act-name(attr-def*)
		REALIZED BY external-procedure-call
act-name	=	... *(some practical naming convention)*
external-procedure-call	=	... *(according to some call interface)*

5 Behavior Definition

The behavior of a vivid agent is defined by means of action and reaction rules. In order to be able to refer to action and reaction rules in plans, VIVA assigns a name to each action and reaction rule. While action rules determine the pro-active behavior repertoire, reaction rules determine the reactive behavior of an agent. Agents without action rules (i.e. without any pro-active behavior) are also called *reagents* (see also the appendix).

behavior-def = REACTION RULES react-rule-def*
behavior-def = ACTION RULES act-rule-def*
REACTION RULES react-rule-def*

Before we go on to explain the definition of actions and reactions, we present a simple example of a reagent specification.

5.1 Example: An Elevator as a Reagent

In order to describe an elevator as a reagent we use three action types: *mvup, mvdown, halt*; and two perception event types: *reqTo(X), arrAt(X)*. Whenever the elevator, while moving up or down, arrives at floor X, this is signalled by the environment as the perception event $arrAt(X)$. Whenever the elevator is requested to serve floor Y, signalled by the event $reqTo(Y)$, this will be recorded in the KB by means of the fluent $req(Y)$. The state of the elevator is described by means of the fluents $at(X)$, $up(X)$, and $down(X)$, i.e. predicates representing the information that the elevator is currently halting at floor X, moving upwards from floor X, or moving downwards from floor X. The elevator can be specified in VIVA as in figure 1.

Reaction rules have the form
Reaction $\longleftarrow$ RECV(Message, From), Condition While *a1*, *a2*, and *a5* are physical reaction rules, *a3* and *a4* are epistemic reaction rules which update the knowledge state according to new perceptive input.

In the following example run, we assume that an incoming request from the 5th floor is received, i.e. RECV(req(5)) holds. We obtain the following execution history:

	transition	KB	Event Queue
0		$\{at(3)\}$	[]
1	req(5)	$\{at(3)\}$	[req(5)]
2	a_1	$\{up(3), req(5)\}$	[]
3	arrAt(4)	$\{up(3), req(5)\}$	[arrAt(4)]
4	a_3	$\{up(4), req(5)\}$	[]
5	arrAt(5)	$\{up(4), req(5)\}$	[arrAt(5)]
6	a_5	$\{at(5)\}$	[]

```
AGENT a    %  elevator "a"

EXTENSIONAL PREDICATES
   CREATE TABLE at( Floor  INTEGER UNIQUE);
   CREATE TABLE up( Floor  INTEGER UNIQUE);
   CREATE TABLE down( Floor  INTEGER UNIQUE);
   CREATE TABLE req( Floor  INTEGER UNIQUE);

INTEGRITY CONSTRAINTS
   exists X: at(X) | up(X) | down(X);

DEFINE PERCEPTION EVENT reqTo( Floor  INTEGER);
DEFINE PERCEPTION EVENT arrAt( Floor  INTEGER);

DEFINE ACTION mvup() REALIZED BY 'call( go_up)';
DEFINE ACTION mvdown() REALIZED BY 'call( go_down)';
DEFINE ACTION halt() REALIZED BY 'call( halt)';

REACTION RULES
a1: DO( mvup), req(X) AND up(Y)
    <-  RECV( reqTo(X)), at(Y) AND X>Y AND X<topfloor.
a2: DO( mvdown), req(X) AND down(Y)
    <-  RECV( reqTo(X)), at(Y) AND X<Y AND X>0.
a3: up(X) <- RECV( arrAt(X)), req(Y) AND X<Y AND X<topfloor.
a4: down(X) <- RECV( arrAt(X)), req(Y) AND X>Y AND X>0.
a5: DO( halt), at(X) AND not req(X)
    <- RECV( arrAt(X)), req(X).

INITIAL BELIEFS at(3).
```

5.2 Defining Actions

The possible actions available to an agent have to be represented in the agent specification. When the agent generates a plan for solving a certain task, it can only use actions from its action repertoire. A plan consists of a sequence of actions. After a plan has been found, it is stored together with the task to be solved by it in the set of *current intentions*. Whenever the agent has nothing more important to do, it executes its current intentions. The interleaving of this proactive behavior with the reactive behavior is described in [Wag96a].

In many cases, an action can be defined by a single action rule specifying the mental precondition of the action, and its mental effects. An action may, however, have context-dependent effects in which case it is defined by a number of action rules with different mental conditions and effects.

In VIVA, there is a distinction between mental, physical and communicative actions. A mental action does not affect the external environment of an agent but only its internal state. A mental action rule has the following form:

act-rule-def	=	rule-name: mental-eff `<-` mental-cond
mental-eff	=	literal | NOT literal
mental-eff	=	[NOT] TASK query
mental-eff	=	NOT INTEND query
mental-eff	=	mental-eff AND mental-eff
mental-cond	=	query | TASK query | INTEND query
literal	=	atom | NON atom
atom	=	pred-name(arg*)
query	=	literal
query	=	NOT query | query AND query | query OR query

A mental effect formula *TASK F* leads to the insertion of F into the task list of the agent while *NOT TASK F* leads to the deletion of F from the task list. An effect formula *NOT INTEND G* leads to the deletion of the goal/plan pair G/P from the agent's list of current intentions.

A communicative action involves sending a message to another agent. The *send message* command is expressed by SEND. An example for a physical action rule used by an embedded agent system controlling an elevator is the following.

```
startmvup(X):
DO( mvup), up(X) AND not at(X)
<- at(X) AND X < topfloor.
```

This rule represents the elevator action of starting to move upwards from floor X when currently halting at floor X required to be below the topfloor.

In general, action rules are defined according to

act-rule-def	=	rule-name: act-expr <- mental-cond
act-expr	=	mental-eff
act-expr	=	DO(act-name(par*)), mental-eff
act-expr	=	SEND(cevt-name(par*), receiver), mental-eff
receiver	=	... *(an ID for another agent)*

where *act-name* refers to an action, and *cevt-name* to a communication event, defined in the schema, and *par** is a list of suitable parameters.

5.3 Defining Reactions

A reaction consists of a (possibly compound) action linked to a triggering event. Events are checked by means of being popped (i.e. querying and consuming the incoming messages) from the event queue. This check is expressed by *RECV(evt-name(par***), sender)*, where *evt-name* denotes either a perception or a communication event type, and *sender* denotes either a perception subsystem (i.e. a process evaluating sensor data) or another agent.

react-rule-def	=	rule-name: react-expr <- RECV(evt-name(par*), sender), mental-cond
react-expr	=	act-expr
react-expr	=	react-expr; act-expr
sender	=	... *(an ID for another agent or a perception subsystem)*

A reaction rule in VIVA is similar to what has been called 'production rule' in expert systems, or 'active rule' in active database systems. It could also be written in a more explicit syntax like, e.g.

```
ON RECV( event, sender)
IF mental-condition
THEN action1, mental-eff1; action2, mental-eff2;...
```

6 Agent Initialization

A VIVA program assigns an initial mental state to an agent. This concerns the agent's extensional beliefs, i.e. all the tables defined in the agent's knowledge base schema, and also its tasks.

initialization	=	INITIAL BELIEFS belief-statement* INITIAL TASKS if-query*
belief-statement	=	INSERT INTO pred-name(arg-tuple*)
belief-statement	=	pred-name(arg-tuple)

VIVA will start the agent by selecting an initial task and generate a plan for it on the basis of its initial beliefs. The resulting goal/plan pair will be added to the set of current intentions, and the agent will successively execute the plan if possible, and at the same time be responsive to perception and communication events.

7 Advanced Knowledge Systems

7.1 Defining Qualified Extensional Predicates

Since diagnoses may be uncertain, the predicate *illness* could as well be defined as an uncertain predicate in the following way:

```
CREATE INCOMPLETE FUZZY TABLE illness(
PatientNo    CHAR(10) NOT NULL,
Disease      CHAR(20) NOT NULL,
StartDate    DATE,
EndDate      DATE
);
```

This table would have an additional implicit column containing uncertainty values:

007	influenca	17.02.91	23.02.91	0.9
007	hepatitis B	01.07.93	15.08.93	0.7
007	influenca	29.01.96	10.02.96	0.9
008	malaria	01.03.95	01.04.95	0.5
007	hepatitis B	29.01.96	29.01.96	-0.9
008	HIV	01.03.95	01.04.95	-0.5

VIVA handles uncertain information according to the possibilistic logic of [DLP94, Wag96b]. Negative values denote the certainty of falsity while positive values denote the certainty of truth.

Temporal and bitemporal predicates can be defined in a similar way. For instance, *illness* could also be defined as a temporal predicate:

```
CREATE TEMPORAL TABLE illness(
PatientNo    CHAR(10) NOT NULL,
Disease      CHAR(20) NOT NULL
);
```

This table would have an additional implicit column containing time stamps representing *valid time*:

007	influenca	17.02.91–23.02.91, 29.01.96–10.02.96
007	hepatitis B	01.07.93–15.08.93
008	malaria	01.03.95–01.04.95

A bitemporal predicate would have an additional time stamp column representing *belief time* in addition to valid time. By distinguishing between temporal and bitemporal predicates VIVA follows the models proposed in the database literature (see, e.g., [SA85], and the recent definition of TSQL2).

Finally, VIVA allows to define *multilevel secure* (MLS)[6] predicates for protecting confidential information from unauthorized access. If not otherwise specified, the four security levels *unclassified* (0), *confidential* (1), *secret* (2), and *top secret* (3) are used to classify entries in a table.

```
CREATE MLS TABLE patient(
PatientNo     CHAR(10) NOT NULL,
Name          CHAR(30) NOT NULL
);
```

Depending on the respective person it may be sensitive information to know whether he is a patient in the hospital. For instance, in the case of a politician, such infomation would be publicly available. But not so in the case of a shy pop star, or a secret service agent:

091	Boris Yeltsin	0
094	Michael Jackson	1
007	James Bond	3

7.2 Defining Intensional Predicates

Intensional predicates are not represented by tables but by Prolog-style deduction rules. For instance, the special diet of patients may be defined by rules such as

```
specialDiet( PatientNo, dietA)
:- illness( PatientNo, diabetes); illness( PatientNo, hepatitis).
```

As in Prolog, logical variables begin with a capital letter whereas constants and names begin with a lowercase letter, or are enclosed in quotes. Instead of the Prolog operators ':–', ';' and ',' VIVA also allows 'IF', 'OR', and 'AND'. An example of an intensional uncertain predicate is

```
infection( PatientNo, pseudomonas_aeruginosa) : 0.6
:- culture( PatientNo, CultNo),
   organism_morphology( CultNo, rod),
   not gram_stain_test_pos( CultNo).
```

Before an intensional predicate can be defined by deduction rules, its schema has to be defined in the same way as for extensional predicates:

[6]See, e.g., [JS91].

int-pred-def = DEFINE [INCOMPLETE] [pred-type] PREDICATE
pred-name(attr-def*)
BY DEDUCTION RULES ded-rule*

The general format of a deduction rule in VIVA is

ded-rule = head :– body
head = literal | NOT literal
body = query

where in addition to simple predicates also qualified predicates are allowed in the formation of atoms:

atom = pred-name(arg*)
atom = pred-name(arg*) : uncertainty-value
atom = pred-name(arg*) @ timestamp
atom = pred-name(arg*) / security-level

Deduction rules must be range-restricted, i.e. all free variables in the head must also occur in the body of a rule, and the body must be an *evaluable* query formula in the sense of [vGT91]. It is also required in VIVA that all predicates in the body of a rule are defined in the knowledge base schema (no matter whether extensional or intensional). The semantics of deduction rules is determined by stable models (see [GL88, HW96a, HW96b]).

8 Pre-Defined Communication Events

In VIVA, there are a number of pre-defined communication events based on 'speech act' theory: *tell*, three forms of *ask*, two forms of *reply*, and two forms of *request*. Their pre-defined operational semantics is based on the assumption that all agents involved in the communication are truthful, benevolent, and competent, implying that they normally provide correct information. These definitions can be overridden by programmer-defined communication rules if necessary.

8.1 Tell, Ask, and Reply

The simplest communication act is *informing* someone about something which is expressed by

```
DEFINE COMMUNICATION EVENT tell( What  INPUT)
```

The piece of information conveyed by a *tell* act is assimilated into the beliefs of the receiver, expressed by the following pre-defined reaction rule:

```
F <- RECV( tell(F), Sender).
```

where F is a formula representing an input.

Like in KQML, three forms of asking are distinguished in VIVA:

```
DEFINE COMMUNICATION EVENT askIf(
QueryID     QHANDLE,
IfQuery     IFQUERY
);
DEFINE COMMUNICATION EVENT askOne(
QueryID     QHANDLE,
Query       QUERY
);
DEFINE COMMUNICATION EVENT askAll(
QueryID     QHANDLE,
Query       QUERY
);
```

Correspondingly, there are two forms of reply:

```
DEFINE COMMUNICATION EVENT replyIf(
QueryID     QHANDLE,
Answer      IFANSWER
);
DEFINE COMMUNICATION EVENT reply(
QueryID     QHANDLE,
Answer      ANSWER
);
```

IFQUERY, QUERY, IFANSWER, and ANSWER are special VIVA types refering to the agent's knowledge base schema (see 4.1.3). IFQUERY defines the set of queries that can be answered by *yes*, *no*, or *unknown*. QUERY is the set of all queries to be answered by answer substitutions (either nondeterministically by just one, like in Prolog, or by all, like in SQL where an answer table is returned).

For practical reasons, each query is associated with a system-generated ID, called *query handle*. This ID is used to store queries (in the pre-defined table *query*) until the answers are received, or until they are timed out.

8.1.1 Handling If-Queries

There are three reaction rules handling the three cases of answering an if-query:

```
SEND( replyIf( QID, yes), S) <- RECV( askIf( QID, F), S), F.
SEND( replyIf( QID, no), S) <- RECV( askIf( QID, F), S), non F.
SEND( replyIf( QID, unknown), S)
<- RECV( askIf( QID, F), S), not F and not non F.
```

Notice that only by employing the two negations of extended logic programs (resp. partial logic), we can express that a sentence is unknown.

```
F <- RECV( replyIf( QID, yes), S), query( QID, F).
non F <- RECV( replyIf( QID, no), S), query( QID, F).
```

8.1.2 Handling General Queries

In our definition of replying to *askOne* and *askAll*, we assume that there are two system predicates, *oneans* and *allans*, providing the respective answer to a query with free variables.

```
SEND( reply( QID, Answer), S)
<- RECV( askOne( QID, F), S), oneans( F, Answer).
SEND( reply( QID, Answer), S)
<- RECV( askAll( QID, F), S), allans( F, Answer).
```

For assimilating an answer table into the knowledge base, a respective set operation is needed. It may look like

```
INSERT INTO QID( Answer)
<- RECV( reply( QID, Answer), S), query( QID, F) and nonempty( Answer).
```

Notice that this is a preliminary formulation which needs further elaboration.

8.2 Requests to Act

In VIVA, there are two kinds of requests to act: an agent may be requested to perform a specific action from its action repertoire (and to confirm the performance), or it may be requested to *see to it that* some goal is achieved by performing appropriate actions.

```
DEFINE COMMUNICATION EVENT reqDo( Act  ACTION);
DEFINE COMMUNICATION EVENT confDo( Act  ACTION);

DEFINE COMMUNICATION EVENT reqStit( What  IFQUERY);
```

In order to define the standard reaction to these requests we assume that there is a system table *action* which contains all available actions together with their preconditions and effects.

```
DO( Act), Eff; SEND( confDo( Act), S)
<- RECV( reqDo( Act), S), action( AR, Act, Cond, Eff) AND Cond

TASK F <- RECV( reqStit(F), S)
```

The latter rule causes the communicated task F to be inserted into the agent's task list leading to the subsequent generation of a plan, and the formation of a corresponding intention.

9 Further Desiderata

We briefly discuss three further desiderata for VIVA.

9.1 Extensibility

It would be desirable that VIVA is extensible in the sense that a programmer can define the knowledge system (resp. logic) she wants to assign to an agent. For instance, instead of handling uncertainty on the basis of possibilistic logic, one might want to use Bayesian inference, or some other uncertainty formalism.

Likewise, it should be possible to define special planning procedures instead of the built-in one. Thus, VIVA should provide extensible libraries of knowledge systems, planning procedures, and communication protocols.

9.2 Schema and Behavior Inheritance

Since in many cases agents will share all or large parts of their schema and behavior definition (which does not mean that they will behave the same) it seems natural to define *agent types*. A specific agent can then be defined by assgining an agent type to it, and by extending or overriding the inherited schema and behavior through the addition of specific schema and behavior definitions.

9.3 Online Modification

It may be useful to be able to change the schema of an agent during its operation, as possible for SQL databases. The same holds for the behavior definition. In the case of autonomous robots such online modification could be achieved by communicating corresponding change requests to the robot.

10 Related Work

There are a number of languages aimed at providing models for *coordination*. One of the best known is *Linda* [CD89] which achieves coordination through a shared variable space (based on a blackboard architecture). Several concurrent logic programming languages, such as *Strand* [FT90], or *Shared Prolog* [BC91], also deal with coordination. However, since these languages are not based on the agent metaphor, they are not as high-level as VIVA. Thus, they do not include the concept of proactive behavior based on run-time task assignments, plan generation and execution. Also, concurrent logic programming languages realize the idea of knowledge-based programming only in a very restricted way, since they are not able to handle qualified (temporal, uncertain, etc.) predicates.

In [Tho95], an extension of Shoham's AGENT0, called PLACA, is described where agents can not only be requested to perform some action but also to see to it that some condition holds which requires planning and plan execution.

The same interaction can be programmed in VIVA by means of the pre-defined communication acts *reqDo* and *reqStit*. Thus, VIVA seems to be more general than PLACA, since 1) message types (i.e. communication events) which are fixed in PLACA are programmer-definable in VIVA, 2) perception is distinguished from communication, 3) actions are represented with their preconditions and effects in the agent program whereas PLACA only lists the names of available actions in the form of *capabilities* and leaves the full representation of actions to the planner.

In [Rao96], a logic programming-like language, called *AgentSpeak(L)* is proposed. While AgentSpeak(L) contains a more complex form of action rules (used to represent predefined plans), it does not provide for online planning. Beliefs in PLACA and in AgentSpeak(L) consist, like in AGENT0, only of literals formed with simple predicates upon which classical inference is performed, i.e. the CWA and negation-as-failure are not used. In VIVA however, we take into account that, based on the experience with SQL and Prolog, the CWA and negation-as-failure are essential for information and knowledge processing. Also, the restriction to simple predicates in AgentSpeak(L), and to temporally qualified predicates in AGENT0 and PLACA exludes many forms of relevant domain knowledge, such as uncertain or confidential information. Finally, neither of AGENT0, PLACA, and AgentSpeak(L) allows intensional predicates for expressing causal relations and generic laws in the form of possibly nonmonotonic deduction rules.

A prototype interpreter for VIVA on the basis of PVM is currently under development [SW96, SW97]. In a first case study [SW97], we used it to implement a distributed diagnosis simulation where several diagnosis agents monitor and test a target system and try to coordinate their diagnostic findings. For this application, knowledge-based programming was essential since the model-based diagnosis method depends on knowledge and assumptions about the target system which can only be expressed in a sufficiently expressive formalism allowing, e.g., for nonmonotonic deduction rules.

In [SW96], we hope to solve the mental ressource conflict between reactive and proactive behavior ('bounded rationality') by running two processes for them in parallel.

Acknowledgements: Thanks to Michael Schröder for stimulating discussions and fruitful cooperation.

References

[BC91] A. Borgi and P. Ciancarini: The Concurrent Language Shared Prolog, *ACM Trans. on Programming Languages and Systems* **13**/1 (1991), 99-123.

[CD89] N. Carriero and D. Gelernter: LINDA in Context, *Comm. of the ACM* **32** (1989), 444-458.

[DLP94] D. Dubois, J. Lang and H. Prade: Possibilistic Logic, in D.M. Gabbay et al. (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Pogramming*, Clarendon Press, Oxford, 1994, 439–513.

[FT90] I. Foster and S. Taylor: *Strand – New Concepts in Parallel Programming*, Prentice-Hall, 1990.

[vGT91] A. van Gelder and R.W. Topor: Safety and Translation of Relational Calculus Queries, *ACM Transactions on Database Systems* **16**:2 (91), 235–278.

[GL88] M. Gelfond and V. Lifschitz: The Stable Model Semantics for Logic Programming, *Proc. Int. Conf. on Logic Programming ICLP'88*, MIT Press, 1988.

[HJW96] H. Herre, J. Jaspars and G. Wagner: Partial Logics with Two Kinds of Negation as a Basis for Knowledge-Based Reasoning, in D. Gabbay and H. Wansing (Eds.), *What is Negation ?*, Oxford University Press, 1996/1997.

[HW96a] H. Herre and G. Wagner: Stable Models are Generated by a Stable Chain, to appear in the *J. of Logic Programming.*

[HW96b] H. Herre and G. Wagner: Stable Semantics for Temporal Deductive Databases, Technical Report, available from the author's home page.

[JS91] S. Jajodia and R. Sandhu: Towards a Multilevel Secure Relational Data Model, *Proc. ACM SIGMOD*, ACM, New York, 1991, 50–59.

[MC95] R.F.P. Marques and J.A. Cunha: Using PVM with a logic programming interface, technical report, F.C.T., Univ. Nova de Lisboa.

[Rao96] A.S. Rao: 'AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language', in [MAAMAW96], 42–55.

[Sho93] Y. Shoham: Agent-Oriented Programming, *Artificial Intelligence* **60** (93), 51–92.

[SA85] R. Snodgrass and I. Ahn: A Taxonomy Of Time in Databases, in *Proc. of ACM SIGMOD*, ACM, New York, 1985, 236–246.

[SW96] M. Schröder and G. Wagner: Concurrent Action and Planning – Towards an Implementation of VIVA, position paper for *Foundations of Multi-Agent Systems (FOMAS'96).*

[SW97] M. Schröder and G. Wagner: Distributed Diagnosis by Vivid Agents, *Proc. Int. Conf. on Autonomous Agents*, ACM Press, 1997.

[Tho95] S.R. Thomas: The PLACA Agent Programming Language, in M. Wooldridge and N.R. Jennings (Eds.), *Intelligent Agents*, Proc. of ATAL'94, Springer-Verlag, 1995.

[MAAMAW96] W. Van de Velde and J.W Perram (Eds.), *Agents Breaking Away*, Proc. of MAAMAW'96, Springer Lecture Notes in AI **1038** (1996).

[Wag95] G. Wagner: From Information Systems to Knowledge Systems, in E.D. Falkenberg et al. (Eds.), *Information System Concepts*, Chapman & Hall, London, 1995; also available from the author's home page.

[Wag96a] G. Wagner: A Logical and Operational Model of Scalable Knowledge- And Perception-Based Agents, in [MAAMAW96], 26–41.

[Wag96b] G. Wagner: Possibilistic and Extended Logic Programs, Technical Report, available from the author's home page.

[Wag96c] G. Wagner: 'Vivid Agents – How They Deliberate, How They React, How They Are Verified', technical report, available from the author's home page.

[Weg96] P. Wegner: Coordination as Constrained Interaction, in P. Ciancarini and C. Hankin (Eds.), *Coordination Languages and Models*, Springer LNCS **1061** (1996), 28–33.

A Programming Examples

A.1 Communicating Elevators

In the communicating elevator scenario, we assume that two elevators a and b operate in the same shaft and must not collide. For simplicity, we shall consider the case with three floors. Elevator a serves floor 1 and 2, while elevator b serves floor 2 and 3, and hence the critical zone is floor 2. The states of the elevators are described by means of the indexical fluents $at(x)$, $up(x)$, and $down(x)$, i.e. indexical predicates representing the information that the resp. elevator is currently halting at floor x, moving upwards from floor x, or moving downwards from floor x.

In addition to perception events, $L_{\mathrm{PEvt}} = \{\mathrm{reqTo}(x), \mathrm{arrAt}(x)\}$, we now also have communication events in order to avoid collisions and deadlocks. The elevators use two communication acts for coordination: requesting the permission to approach, and granting the permisssion, expressed by $L_{\mathrm{CEvt}} = \{reqPerm,\ grPerm\}$. A special fluent $turn(x)$ controls whose turn it is when both want to go to floor 2 at the same time. We assume that initially $turn(a)$ holds for both elevators. This leads to the following specification, including a coordination protocol.

The behavior of elevator a is specified by the rules $a_1, \ldots, a_9$, while that of elevator b is specified by the dual rules $b_1, \ldots, b_9$.

```
AGENT a    %  elevator "a" serving floor 1 and 2

EXTENSIONAL PREDICATES
   CREATE TABLE at( Floor  INTEGER UNIQUE);
   CREATE TABLE up( Floor  INTEGER UNIQUE);
   CREATE TABLE down( Floor  INTEGER UNIQUE);
   CREATE TABLE turn( Elevator  CHAR(1) UNIQUE);
```

```
   CREATE TABLE req( Floor  INTEGER UNIQUE);
   CREATE TABLE permreq( Elevator  CHAR(1) UNIQUE);

INTEGRITY CONSTRAINTS
   exists(X, at(X) | up(X) | down(X));

DEFINE PERCEPTION EVENT reqTo( Floor  INTEGER);
DEFINE PERCEPTION EVENT arrAt( Floor  INTEGER);
DEFINE COMMUNICATION EVENT reqPerm();
DEFINE COMMUNICATION EVENT grPerm();

DEFINE ACTION mvup() REALIZED BY 'call( go_up)';
DEFINE ACTION mvdown() REALIZED BY 'call( go_down)';
DEFINE ACTION halt() REALIZED BY 'call( halt)';

REACTION RULES
a1: SEND(reqPerm,b), req(2)  <-  RECV( reqTo(2)), at(1).
a2: SEND(grPerm,b)
    <- RECV(reqPerm,b), (at(1) and not req(2)) or down(2).
a3: SEND(grPerm,b), turn(a)
    <- RECV(reqPerm,b), at(1) and req(2) and turn(b).
a4: permreq(b) and turn(b)
    <- RECV(reqPerm,b), at(1) and req(2) and turn(a).
a5: permreq(b)  <-  RECV(reqPerm,b), at(2) or up(1).
a6: DO(mvup), up(1)  <-  RECV(grPerm,b), at(1) and req(2).
a7: DO(mvdown), req(1) and down(2)  <-  RECV(reqTo(1)), at(2).
a8: SEND(grPerm,b), not permreq(b)
    <- RECV(reqTo(1)), at(2) and permreq(b).
a9: DO(halt), at(X) and not req(X)  <-  RECV(arrAt(x)), req(X)

INITIAL BELIEFS at(1), turn(a).
```

Elevator b has the same schema definition as elevator a, only its reaction rules are the duals of those of a:

```
REACTION RULES
b1: SEND(reqPerm,a), req(2)  <-  RECV(reqTo(2)), at(3).
b2: SEND(grPerm,a)
    <- RECV(reqPerm,a), (at(3) and not req(2)) or up(2).
b3: SEND(grPerm,a), turn(b)
    <- RECV(reqPerm,a), at(3) and req(2) and turn(a).
b4: permreq(a) and turn(a)
    <- RECV(reqPerm,a), at(3) and req(2) and turn(b).
b5: permreq(a)  <-  RECV(reqPerm,a), at(2) or down(3).
b6: DO(mvdown), down(3)  <-  RECV(grPerm,a), at(3) and req(2).
b7: DO(mvup), req(3) and up(2)  <-  RECV(reqTo(3)), at(2).
b8: SEND(grPerm,a), not permreq(a)
```

```
    <- RECV(reqTo(3)), at(2) and permreq(a).
b9: DO(halt), at(X) and not req(X)  <-  RECV(arrAt(x)), req(X)

INITIAL BELIEFS at(3), turn(a).
```

This program was proved to be correct in [Wag96c]. In particular, the correctness proof guarantees thate there will be no collisions between the two elevators.

A.2 A Two-Database System

A cooperative knowledge base is a particular type of reagent sharing and exchanging its information with other knowledge bases. A multi-knowledge-base system is a collection of cooperative knowledge bases communicating with each other by means of *tell*, *ask* and *reply*.

In our example, we assume that two relational databases a and b (e.g. of two different cities) cooperate in query answering. Each of them knows about which predicates it has complete information by means of a catalog table *COMPLETE* containing all relation names for which the Closed-World Assumption holds locally. For simplicity, we consider only two predicates: *resident* and *married*. Since each city has complete information about its residents, in both databases holds that $resident \in COMPLETE$. However, this does not hold for *married*, since anyone may marry anyone in any city, and this may be recorded only there.[7]

We assume that there is a meta-predicate *anscomp*, stating for which if-queries the database is answer competent, according to the following definition:

```
DEFINE PREDICATE anscomp( Q  IFQUERY)
BY DEDUCTION RULES
  anscomp( Q1 and Q2) :- anscomp(Q1), anscomp(Q2).
  anscomp( Q1 or Q2) :- anscomp(Q1); anscomp(Q2).
  anscomp(not Q) :- anscomp(Q).
  anscomp(exists(X, Q)) :- anscomp(Q).
  anscomp(Q) :-
    Q =.. [Predicate | Arguments],
    complete( CL), member( Predicate, CL).
```

There are two possible originators of a query/message: a local user, or the other database. The communication protocol consists of 5 interaction rules:

```
AGENT a

EXTENSIONAL PREDICATES
   CREATE TABLE resident( Name  STRING);
   CREATE INCOMPLETE TABLE married(
```

[7]Notice that, as a one-place predicate, *resident* is spatially *indexical*, i.e. it has different extensions at different locations, while *married* is temporally indexical, having different extensions at different timepoints.

```
      Husband  STRING,
      Wife     STRING );

DEFINE COMMUNICATION EVENT askIf( Query  IFQUERY);
DEFINE COMMUNICATION EVENT replyIf( Answer  IFANSWER);

REACTION RULES
a1: SEND(replyIf(Q,yes),S) <- RECV(askIf(Q),S), Q.
a2: SEND(replyIf(Q,no),S) <- RECV(askIf(Q),S), not Q and anscomp(Q).
a3: SEND(askIf(Q),b) <- RECV(askIf(Q),user), not Q and not anscomp(Q).
a4: SEND(replyIf(Q,unknown),b)
    <- RECV(askIf(Q),b), not Q and not anscomp(Q).
a5: SEND(replyIf(Q,A),user) <- RECV(replyIf(Q,A),b).

INITIAL BELIEFS
   complete([resident]),
   resident(susan), resident(peter),
   married(peter,linda), married(tom,susan).
```

The database b has the same schema as a, and corresponding interaction rules:

```
AGENT b

EXTENSIONAL PREDICATES
   CREATE TABLE resident( Name  STRING);
   CREATE INCOMPLETE TABLE married(
      Husband  STRING,
      Wife     STRING );

DEFINE COMMUNICATION EVENT askIf( Query  IFQUERY);
DEFINE COMMUNICATION EVENT replyIf( Answer  IFANSWER);

REACTION RULES
b1: SEND(replyIf(Q,yes),S) <- RECV(askIf(Q),S), Q.
b2: SEND(replyIf(Q,no),S) <- RECV(askIf(Q),S), not Q and anscomp(Q).
b3: SEND(askIf(Q),a) <- RECV(askIf(Q),user), not Q and not anscomp(Q).
b4: SEND(replyIf(Q,unknown),a)
    <- RECV(askIf(Q),a), not Q and not anscomp(Q).
b5: SEND(replyIf(Q,A),user) <- RECV(replyIf(Q,A),a).

INITIAL BELIEFS
   resident(mary), resident(tom), married(peter,susan).
```

Notice that, since there are no global integrity constraints in this example, it cannot be prevented that someone is (of course, illegaly) involved in multiple marriages, such as Peter and Susan.

In order to demonstrate a run of the system, we assume that a database user at b asks if *Tom is married*, i.e. b receives the message

askif(exists(X, married(tom,X)))

from *user*. This event triggers the rule set $\{b_1, b_2, b_3\}$, from which b_3 is the only reaction to be executed since the conditions of b_1 and b_2 are not satisfied. So, b reacts by applying its reaction rule b_3: it forwards the query to a since it is not able to answer it. Then, a finds a positive answer, *married(tom,susan)*, and replies *yes* to b by applying a_1. Finally, b passes the answer to the user by applying b_5.

The following sections are from [SW97].

B Vivid Knowledge Systems

The knowledge system of a vivid agent is based on three specific languages: L_{KB} is the set of all admissible knowledge bases, L_{Query} is the query language, and L_{Input} is the set of all admissible inputs, i.e. those formulas representing new information a KB may be updated with. In a diagnosis setting L_{Input} may be $\{test(\_,\_), diagnoses(\_,\_)\}$, where *test* is used to update other agents' test results and *diagnoses* to update the agents' diagnosis results. While the input language defines what the agent can be told (i.e. what it is able to assimilate into its KB), the query language defines what the agent can be asked. Where L is a set of formulas, L^0 denotes its restriction to closed formulas (sentences). Elements of L^0_{Query}, i.e. closed query formulas, are also called *if-queries*.

Definition 1 (Knowledge System) *An abstract knowledge system*[8] $\boldsymbol{K}$ *consists of three languages and two operations: a knowledge representation language* L_{KB}*, a query language* L_{Query}*, an input language* L_{Input}*, an inference relation* $\vdash$*, such that* $X \vdash F$ *holds if* $F \in L^0_{\mathrm{Query}}$ *can be inferred from* $X \in L_{\mathrm{KB}}$*, and an update operation* Upd*, such that the result of updating* $X \in L_{\mathrm{KB}}$ *with* $F \in L^0_{\mathrm{Input}}$ *is the knowledge base* $\mathsf{Upd}(X, F)$*.*

We now present two examples of basic knowledge systems.

B.1 Relational Databases

A finite set of ground atoms corresponds to a relational database. For instance, in diagnosis a relational database may contain observations and connections of the system to be diagnosed: $X_1 = \{hi(s_1), conn(s_1, s_2)\}$, may represent the information that switch s_1 is high and that it is connected to switch s_2. As a

[8]See also [Wag95].

kind of natural deduction from positive facts an inference relation $\vdash$ between a database X and an if-query is defined in the following way:

$$\begin{array}{lll} (\vdash a) & X \vdash a & \text{if} \quad a \in X \\ (\vdash \neg a) & X \vdash \neg a & \text{if} \quad a \notin X \end{array}$$

Notice the non-monotonicity of $\neg$: negation in relational databases corresponds to *negation-as-failure*. Compound if-queries, involving conjunction and disjunction, are handled in the standard way. Negated compound if-queries are treated by simplification according to the DeMorgan rules and double negation elimination.[9] For example, $X_1 \vdash hi(s_1) \wedge \neg hi(s_2)$. Because of its built-in general Closed-World Assumption, a relational database X answers an if-query F by either yes or no: the answer is yes if $X \vdash F$, and no otherwise.

Updates are insertions, $\mathsf{Upd}(X, a) := X \cup \{a\}$, and deletions, $\mathsf{Upd}(X, \neg a) := X - \{a\}$, where a is an atom. For a consistent set of literals E, we have $\mathsf{Upd}(X, E) = X \cup E^+ - E^-$, where E^+ contains the positive, and E^- contains the negative literals of E. For instance,

$$\mathsf{Upd}(X_1, \neg hi(s_1) \wedge hi(s_2)) = \{conn(s_1, s_2), hi(s_2)\}$$

describes a possible transaction. The knowledge system of relational databases is denoted by $\boldsymbol{A}$.[10] Knowledge systems extending $\boldsymbol{A}$ conservatively are called *vivid*. Positive vivid knowledge systems use a general Closed-World Assumption, whereas general vivid knowledge systems employ specific Closed-World Assumptions (and possibly two kinds of negation). For instance, $\boldsymbol{A}$ can be extended to a general vivid knowledge system, called *factbase*, by allowing for literals instead of atoms as information units (see below). Further important examples of positive vivid knowledge systems are temporal, uncertain and disjunctive databases. All these kinds of knowledge bases can be extended to *deductive knowledge bases* by adding deduction rules of the form $F \leftarrow G$ [Wag95].

B.2 Factbases and Extended Logic Programs

A knowledge base consisting of a consistent set of ground literals (viewed as positive and negative facts) is called a *factbase*. In a factbase, the CWA does not in general apply to all predicates, and therefore in the case of a non-CWA predicate, negative information is stored along with positive. This allows to represent predicates for which the KB does not have complete information.

The schema of a factbase stipulates for which predicates the CWA applies by means of a special set *CWRel* of relation symbols. Explicit negative information is represented by means of a *strong* negation $\sim$. For instance, in the factbase

$$\begin{array}{rcl} CWRel & = & \{conn\} \\ X_2 & = & \{conn(s_1, s_2), \sim hi(s_1)\} \end{array}$$

[9] Notice that inference in relational databases corresponds to preferential entailment based on minimal models.

[10] $\boldsymbol{A}$ stands for $\boldsymbol{A}$tomic.

the CWA applies only to the predicate *conn* representing the connection of components, i.e. if it is not positively confirmed that two components are connected we assume that they are not. In contrast to this, the CWA does not apply to *hi* anymore. Now we can distinguish the two cases that we have explicitly observed that a switch is not high and that we do not have information about the switch. I.e. $X_2 \vdash \sim hi(s_1)$ means that switch s_1 is observed to be not-high (i.e. low), whereas $X_2 \vdash \neg hi(s_2)$ only expresses that we cannot infer s_2 to be high, which means that it is either not high or that there is no information. As a kind of natural deduction from positive and negative facts an inference relation $\vdash$ between a factbase X and an if-query is defined in the following way:

$$\begin{array}{lll} (\vdash \neg) & X \vdash \neg p(c) & \text{if} \quad p(c) \notin X \\ (\vdash \sim) & X \vdash \sim p(c) & \text{if} \quad \sim p(c) \in X \\ (\vdash \sim_{\mathrm{CWA}}) & X \vdash \sim p(c) & \text{if} \quad p \in \mathit{CWRel} \;\; \& \;\; X \vdash \neg p(c) \end{array}$$

where $p(c)$ stands for an atomic sentence with predicate p and constant (tuple) c. $\sim$ and $\neg$ are also called *strong* and *weak* negation. Note that, since X is consistent, the strong negation implies the weak negation:

$$X \vdash \sim F \text{ implies } X \vdash \neg F$$

Compound formulas are treated according to DeMorgan and double negation rules.[11] A factbase X answers an if-query F by yes if $X \vdash F$, by no if $X \vdash \sim F$, and by unknown otherwise. Updates are recency-prefering revisions:

$$\begin{array}{lll} \mathsf{Upd}(X, p(c)) & := & \left\{ \begin{array}{l} X \cup \{p(c)\} \text{ if } p \in \mathit{CWRel} \\ X - \{\sim p(c)\} \cup \{p(c)\} \text{ else} \end{array} \right. \\ \mathsf{Upd}(X, \sim p(c)) & := & \left\{ \begin{array}{l} X - \{p(c)\} \text{ if } p \in \mathit{CWRel} \\ X - \{p(c)\} \cup \{\sim p(c)\} \text{ else} \end{array} \right. \end{array}$$

The knowledge system of factbases is denoted by $\boldsymbol{F}$. The extension of $\boldsymbol{F}$ by adding deduction rules leads to *extended logic programs* with two kinds of negation.

Definition 2 *Extended Logic Program*
An extended logic program consists of a factbase and a set of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_m, \neg L_{m+1}, \ldots, \neg L_n \;\; (0 \leq m \leq n)$$

where each L_i is a positive or negative fact ($L_i = A | \sim A$, $0 \leq i \leq n$).

Inference in extended logic programs can be defined model-theoretically as preferential entailment based on stable coherent partial models [HJW96] corresponding to the *answer sets* of Gelfond and Lifschitz. In the next section we briefly describe how to employ extended logic programs to solve model-based diagnosis problems.

[11] Inference in factbases corresponds to predicate circumscription in partial logic, i.e. to preferential entailment based on minimal coherent partial models.

C A PVM-Prolog-Based Interpreter for Vivid Agents

PVM (parallel virtual machine) is a standard software in distributed and parallel computing that permits a heterogeneous collection of UNIX computers (ranging from a personal computer or workstation to massively parallel processors) to be viewed as a single parallel computer. PVM coordinates the different hardware architectures and data formats of the networked machines. The PVM computing model is simple, yet very general: The main principle are tasks which are spawned in the network and which access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization of the tasks.

Besides the standard interfaces to C, C++ and Fortran, PVM is available in Lisp, Ada and Prolog. Its widespread use and generality makes PVM an excellent candidate for implementing interpreters for multi-agent programming languages. A general scheme for such implementations contains PVM as a layer on top of low-level network protocols. The next layer contains programming languages with full PVM access. On top of them, agent interpreters can be implemented, allowing to run high-level agent programs as the topmost layer. Alternatively, agents can be programmed directly in the lower level of standard programming languages with PVM access. Notice, however, that we rather recommend to use a high-level declarative agent programming language such as our vivid agent specification language in order to achieve readable and compact source code.

C.1 PVM and PVM-Prolog

PVM realizes communication of spawned tasks by message-passing. The message-passing primitives are tailored for heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum. There are no limits for messages: A task can send an unlimited number of messages of unlimited size, though the size is, of course, limited by the available memory of the host.

An essential feature for multi-agent applications is asynchronous communication. Tasks can perform an asynchronous blocking send and receive. A *blocking* send or receive waits until the message is sent or an incoming message arrives. Blocking is often useful to let the agent idle and safe computational power. Besides the primitive send, a multi-cast to a set of tasks and broadcast to user-defined groups is supported. In general, the message order is preserved in point-to-point communication. Additionally, PVM provides dynamic hardware configuration, i.e. addition and deletion of hosts at run-time and signaling, such that on events such as exit, addition or deletion of tasks other tasks are informed.

All these PVM-primitives are included in PVM-Prolog, a Prolog-core extended by a PVM-interface [MC95]. In addition to PVM's coarse grain parallelism, PVM-Prolog provides process-internal threads. While PVM tasks run physically distributed the threads are process internal and therefore have less overhead.

For the diagnosis agent implementation we use PVM's coarse grain parallelism to spawn agents on the network's machine and to provide message passing facilities among the agents. In on-going work we use threads to run a planner concurrently with the perception-reaction-cycle. We investigate the possibility to use the low-level concurrency of threads which is executed as time-sliced interleaving to solve the problem of bounded rationality. Depending on the application different priorities and computational resources can be assigned to the threads for planning and reaction.

C.2 An Interpreter for Vivid Agents

To set up the multi agent system a corresponding number of processes is spawned at different locations each loading a file containing its behavior specification. Once the processes are set up they can communicate through the PVM-primitives `pvm_send` and `pvm_recv`.

```
cycle :-
  newEvt( Evt ),
  findall(ActEff, (reaction(ActEff,Evt,Cond),
                   demo(Cond)), ActEffs),
  perform( ActEffs ),
  cycle.

newEvt( Msg/From ) :-
  pvm_nrecv( -1,1,Msg/From ).
newEvt( noEvt ).
```

Figure 1: Perception-Reaction-Cycle.

The execution of a vivid reagent is specified by a *perception-reaction cycle* similar to the 'observation-thought-action' cycle proposed by Kowalski and the interpreter cycle of AGENT0 [Sho93]. First, we check if there is a message (`newEvt`, see Figure 1). If there is one, the predicate `pvm_nrecv`, which realizes a non-blocking receive, succeeds and the message is popped from the event queue. Next, the reaction rules are evaluated. For all reaction rules matching the event expression the precondition `Cond` is tested and in case it holds, the triggered actions are collected in the set `ActEffs`. Subsequently, they are performed and the cycle is closed.

```
perform( [noAct/Effs|ActEffs] ) :-
  assimilate( Effs ),
  perform( ActEffs ).
perform( [send(Msg,To)/Effs|ActEffs] ) :-
  name2id( To,Id ),
  i_am( Self ),
  pvm_send( Id,1,Msg/Self ),
  assimilate( Effs ),
  perform( ActEffs ).
perform( [do(Act)/Effs|ActEffs] ) :-
  call( Act ),
  assimilate( Effs ),
  perform( ActEffs ).
perform( [halt/_|ActEffs] ) :-
  perform( ActEffs ),
  pvm_exit,
  fail.

assimilate( [not(L)|Effs] ) :-
  retractall( fact(L) ),
  assimilate( Effs ).
assimilate( [L|Effs] ) :-
  assert( fact(L) ),
  assimilate( Effs ).
```

Figure 2: Executing Reactions.

Triggered actions are performed as follows (see Figure 2): Communicative actions are translated into PVM-Prolog's `pvm_send`, whereas a physical action causes a call to a procedure with the same name. The epistemic effects of actions are assimilated into the knowledge base by means of the Prolog operations *assert* and *retract*.