Agents with Complex Plans: Design and Implementation of CASA*

Stephan Flake, Christian Geiger
C-LAB
Fuerstenallee 11
33102 Paderborn
Germany
email:{flake, chris}@c-lab.de

Abstract

We describe the design of CASA, an agent specification language that builds on the formal agent specification approach AgentSpeak(L) and extends it by concepts from concurrent logic programming. With CASA it is possible to design agents with complex behavior patterns like speculative computations and parallel executed strategies. The design of multi agent systems composed of CASA agents is supported by providing predefined message structures and integrating an existing agent communication framework.

1 Introduction

Agent-based technology in the sense of distributed computing is growing dramatically in many directions. We are mainly concerned with the structured design and rapid prototyping of complex concurrent systems and from our design point of view agents provide a natural metaphor for conceptualizing and building a wide range of these systems including flexible manufacturing systems, mobile robots, or interactive 3D graphics [Geiger, 1998; Geiger *et al.*, 1999].

From an implementation point of view the existence of special libraries or dedicated programming languages that provide data and control structures for describing and manipulating agent specific properties allows a straight forward implementation of the designed models. Also, there is considerable work on formalizing notions in multi agent systems (MAS), e.g. commitments, capabilities, and know how.

A major criticism of much of the formal work is that only little advice is given on how to apply theoretical results to practical realizations. Summarizing recent work on the design of agent-based systems (e.g. [Nwana and Ndumu, 1999]), the following principles should be regarded: (1) transform theoretical definition/specification concepts to a practical realization, (2) regard and, if possible, reuse well-known concepts from related fields and existing approaches, (3) develop an extendible and flexible design that is efficiently realized.

The work presented in this paper describes CASA, a multi agent specification language, and its efficient implementation. Based on our experiences with concurrent logic programming we identified the need for a multi agent system that allows rapid prototyping of system elements similar to our previous work [Geiger and Lehrenfeld, 1994]. Moreover, we wanted to exploit the structured design view the MAS approach (and particular the BDI architecture) provides for. Lastly, we decided to base our new system on an existing solid framework for describing BDI agents. We found Rao's work on AgentSpeak(L) particularly useful [Rao, 1996], because it demonstrates a successful reverse engineering approach of an implemented MAS that is now given a formal specification [d'Inverno and Luck, 1998]. By extending Rao's specification with our new features for complex plans (e.g. different types of plans, parallel plans and plan elements, speculative computations) and refining the abstract interpreter it was possible for us to directly derive an efficient implementation that supports these new features. We defined the following properties as essential: *Speculative Computations:* Plans are only triggered if the plans' preconditions hold. The reduction of such conditions can itself result in a complex computation requiring the reduction of new sub-goals and the evaluation of new plans. CASA allows such "deep guards" (similar to Concurrent Prolog) by providing special data structures for different plan types that are accessed during the interpreter cycle. *Concurrency in plans and between plans:* CASA allows the concurrent execution of multiple plans as well as the parallel processing of elements within a plan. This is achieved by providing special data structures (multi stack for intentions) that are used to store and access currently executed plans. *Hierarchical plan structure:* Based on the type of preconditions in plans, CASA sets precedences between different applicable plans. Priorities allow to select between plans of the same type. *Design using an abstract interpreter:* The operational semantic in CASA was specified using an abstract interpreter. For prototyping the interpreter was divided in separate functional blocks that were implemented using Java. A textual specification format describing

*This work has been partly funded by a german research grant (DFG, SPP 1064: Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen)

the agents is parsed and executed by the CASA interpreter at runtime.

2 BDI agents, AgentSpeak and Guarded Horn Clauses

Perhaps the most compelling reason for modern agent approaches is that these approaches combine a respectable philosophical model of human practical reasoning [Bratman, 1987], a number of well designed implemented systems and several successful applications (e.g. factory process control, business process, fault diagnosis systems). Recent agent architectures that provide significantly support for mentalistic notions include JAM [Huber, 1999], Agent0 [Shoham, 1993], or PLACA [Thomas, 1995]. In addition to these monolithic architectures, a number of layered approaches such as TouringMachines [Ferguson, 1992] or InteRRaP [Mueller, 1996] exist, that provide different modules for reactive behavior, planning, or scheduling. Many of these systems are built on the foundations of BDI logics.

Within the MAS community, the BDI model [Rao and Georgeff, 1991] has come to be possibly the best known and best studied model of agency. However, it is generally concerned that there is a gap between those powerful BDI logics and practical system implementations. Following [Mora *et al.*, 1998], one reason is that the logical formalisms used to define the models do not have an operational model to support them. Recent work to overcome this problem can be separated in two major directions. One is to define BDI models using a suitable logical formalism that allows to represent mental states and has operational procedures to use the logic as knowledge representation formalism. Mora et al., for example, defined the notions of beliefs, desires, and intentions using extended logic programming (ELP), an extension of logic programming with a second, explicit negation. This allows the explicit representation of negative information based on a well founded semantics.

Another approach is to extend existing BDI logics with appropriate operational models so that agent theories become computational. Schild's representation of a BDI logic follows this approach [Schild, 1998]. Rao also took this approach in [Rao, 1996] where he defines a proof procedure for the propositional version of his BDI logic. He presented AgentSpeak(L), a formal agent description language for an already implemented MAS. AgentSpeak(L) allows the specification of an agent in a declarative notation similar to horn clauses used in logic programming languages like PROLOG. An agent's behavior (i.e., plans) and its knowledge are completely and clearly described using this notation. Plans in AgentSpeak(L) refer to horn clauses that are triggered by an event.

We found the description of AgentSpeak(L) similar to the semantics of flat guarded horn clauses used in concurrent logic programming languages like Flat Concurrent Prolog (FCP). FCP is a concurrent logic programming language [Shapiro, 1989] and was developed by E. Shapiro during 1985-1989. It provides a process oriented semantics and was implemented on multi processor architectures. The language can easily handle infinite computations typical for the modeling of reactive systems. The language incorporates guarded-command indeterminism, dataflow like synchronization, and a commitment mechanism. A Flat Concurrent Prolog program is a finite set of *Guarded Horn Clauses* (GHC), denoted as $H \leftarrow G_1, G_2, ..., G_n \mid B_1, B_2, ..., B_m$. The operator '|' separates the guard from the body and is called *commit operator*. The head and body elements define parallel processes with arguments. The components of the guard define test conditions related to constraints that the head's arguments should satisfy. Declaratively, the commit operator is read just like a conjunction. The abstract computation model of Flat Concurrent Prolog is established by process interpretation of the goals forming the resolvent. The goals are regarded as an asynchronous *process network*. The concurrent processes communicate and synchronize via shared logical variables according to an asynchronous communication model. When several clauses are applicable at once, the commit operator '|' acts as a control primitive ensuring that clause selection is carried out in a mutual exclusive manner. The potential for parallelism offered by clause selection might be considered as some form of restricted OR-parallelism.

The concepts of concurrent logic languages received significant attention in the 80's and several parallel programming languages based on these ideas were implemented, e.g. FCP or Strand. Concurrent constraint programming languages and multi-paradigm programming languages like DFKI Oz/Mozart or AKL inherited many ideas from these approaches. Such languages often build the implementation base for multi agent systems.

3 Specification of CASA Agents

The specification of CASA agents is based on AgentSpeak(L), Guarded Horn Clauses and several extensions. The complete specification covers the definition of beliefs, goals, actions, messages, events, plans, and agents [Geiger, 1998]. In the following we focus on the definition of plans and agents and will describe the other elements only briefly and informally.

Beliefs compare to facts in logic programming. If two facts $a(t)$ and $b(s)$ are processed sequentially this is written as $a(t); b(s)$. A concurrent execution is denoted as $a(t), b(s)$. A **goal** $g(s)$ represents the states an agent wants to achieve in the future. There are two different types of goals. A test $!g(s)$ simply looks for a belief in the agent's knowledge base that unifies with the test. A deep goal $?g(s)$ is a goal in the PROLOG sense and needs the reduction of one or more strategies. An **action** $a(t)$ refers to a basic behavior block the agent can perform to modify its environment. **Messages** are special actions with a given structure including sender, receiver, type, and content. If a message from other agents, the system, or the user can be perceived by an agent, this message is denoted as an **event**. The behavior of an agent is

basically defined by specifying **plans**. A plan P is formally defined as an *extended guarded horn clause* of form $P: H \leftarrow G_1...G_n \mid B_1...B_m(p)$, in which H is the head, G_i are guards, B_j are body predicates, and p defines a priority. The head H of a CASA agent strategy P describes the event the agent must perceive in order to execute the plan. The guard elements may consist of any number of tests, goals and messages which are processed sequentially (separated by ';') or in parallel (separated by ','). The priority (p) of a plan denotes its importance for the agent. The evaluation of a guard G_i can result in a complex computation if G_i is not unifiable with a fact but with the head of another plan. Such *deep guards* relate to PROLOG's backtracking mechanism but can be seamlessly integrated in the semantics of concurrent logic programming languages using guarded commands.

Based on the type of the guards different strategies are distinguished at run time. If all guards are simple tests, the strategy is considered as *reactive*. If there are also deep goals in the guards, the strategy is *deliberative*, because the evaluation of the guards requires a speculative computation which evaluates other strategies in order to reduce the goal (multi level plans). If the context test also requires the communication between agents, the strategy is described as *communicative*. Other actions than messages are not allowed in guards. If multiple strategies can be applied, reactive strategies take precedence over deliberative strategies. Communicative strategies have lowest priority. During execution an agent can suspend executing strategies and resume suspended ones by using special operations for suspending/resuming.

Execution Cycle of a CASA Agent

A CASA agent is described by a tuple *(Bel, Plans, Msg, Act, RunTime(Evt, Int, AMsg, AAct, S_e, S_p, S_i))*, where $Bel, Plans, Msg, Act$ are sets with elements representing the agent's beliefs, plans, messages, and actions. The structure $RunTime(...)$ describes a component that defines the state of execution in each agent at run time. This structure consists of a set of currently perceived events (Evt), a set of intentions (Int, i.e., strategies that are currently pursued by the agent), a subset of messages/ actions ($AMsg, AAct$), that are processed in the current execution cycle, and a number of selection functions (S_e, S_p, S_i).

The operational semantics of a CASA agent can be best described by means of an abstract interpreter as it is visually given in Figure 1. A formal version of this interpreter can be found in [Flake, 1999]. The interpreter manages the execution of all agent activities in an interpretation loop which is controlled by three functions that control event selection, plan selection, and intention selection. By modifying their implementation, the system can be easily tailored for different operational semantics in other applications.

The interpretation starts with the selection of an incoming event. All perceived events are stored and the selection function S_e selects a suitable element. In a second step, a set of relevant plans which are appropri-

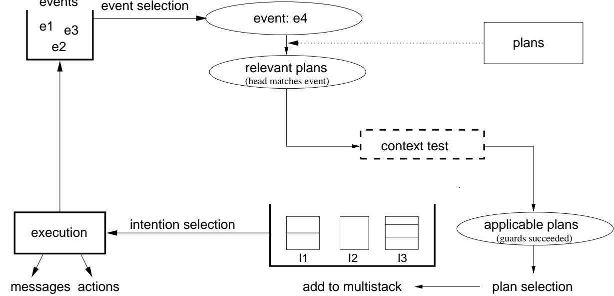

Figure 1: CASA Execution Cycle

ate for processing the selected event are identified. A *relevant plan* is defined by a plan which head matches the selected event. The preconditions of all relevant plans are checked against the facts/plans stored in the agent's belief base. This test can include the evaluation of other strategies, e.g., if a precondition is a goal of another strategy. This is realized by generating a corresponding event that is processed in one of the next execution cycles. A relevant plan which preconditions are all satisfiable is called an *applicable plan*. The agent uses the selection function S_p to select one applicable plan from the set of all applicable plans. This plan is processed as the pursued strategy and becomes instantiated as an intention on the multistack. If the event that triggered the plan was generated by the agent itself (as a consequence of a former execution cycle) the plan is pushed onto the intention stack that generated the event. If the event was generated from other agents, the environment, or the user, the selected applicable plan is stored as a new intention on the multistack. This concept allows each agent to investigate several plans in parallel and to instantiate new (sub)intentions. Finally, the interpreter selects an intention by means of the selection function S_i from the multistack and executes the next body element. Execution can result in either an action, a message, or the generation of one or more (sub)intentions as new events. Thereafter, the interpreter advances to process the next event or continues to process the existing intentions on the multistack.

4 Modeling CASA

The CASA language for modeling agents is syntactically defined similarly to the agent language JAM [Huber, 1999], but the semantics differ with respect to the underlying CASA interpreter model. Expressions in the CASA specification language refer to first-order terms of the formal CASA specification. An expression can be a variable, a constant (a string or a number), a function call (e.g. calculation or comparative operations), or a relation. A relation simply associates a list of data values with an identifier. **Facts** correspond to the beliefs of the formal CASA specification. A fact is defined with the keyword FACT followed by a relation. The two different types of **goals** are handled in the following ways: Syntactically a test is introduced by the keywords EXISTS FACT followed

by a relation. The agent's knowledge base is checked if the specified fact with the given argument values exists and returns TRUE if that fact could be found. Deep goals are introduced by the keyword ACHIEVE followed by a relation and a priority value. The relation name specifies the goal to achieve and the relation arguments represent call-by-value-and-result parameters for relevant plans. There are a number of pre-defined CASA **actions** available, e.g. for manipulating the agent's knowledge base or assigning values to variables. For any other action which is not predefined in CASA additional functions can easily be declared. Additional functions have to be introduced by the keyword EXECUTE followed by an identifier representing the action and a list of arguments. **Messages** are special actions; they are inter-agent operations and therefore concern both the agent's micro and macro view. CASA has some pre-defined message types to specify messages like REQUEST, INFORM, or REPLY. These keywords are followed by a list of arguments declaring a message receiver, a message label, and the content of the message. CASA **plans** correspond to extended guarded horn clauses described in the formal CASA specification. They are defined along the lines of the following patterns:

```
PLAN: {
   NAME:         <string>;
   DESCRIPTION:  <string>;
   GOAL:         ACHIEVE <relation>;
   TYPE:         <plan type>;
   PRECONDITION: <list of conditions>;
   BODY:         <list of actions>;
   FAILURE:      <list of actions>;
   PRIORITY:     <numeric value>;
}
```

In addition to the five plan sections which directly refer to the components of extended guarded horn clauses we have defined sections for informally describing the plan, specifying a plan type, and a failure section, which is executed when errors occur during run time execution of the plan body. Only some actions are allowed in this section, e.g. assignments to variables in order to achieve a consistent state before dropping the plan. Due to a better readability the elements to execute in parallel are explicitly declared in a PARALLEL section (instead of using commas), while sequences are still simply separated by semicolons. Some additional pre-defined structures are available in plan bodies to support easy specification development, e.g. IF-THEN-ELSE or WHILE-loops. With these language elements it is possible to specify a complete initial state of CASA agents. Each agent specification is composed of the four main sections *functions, goals, facts* and *plans*:

```
FUNCTIONS: EVENTSELECT:     <selection function>;
           PLANSELECT:      <selection function>;
           INTENTIONSELECT: <selection function>;

GOALS:     ACHIEVE <relation> : <numeric value>;
           ...
FACTS:     FACT <relation>;
           ...
PLANS:     PLAN: {...}
           ...
```

Three selection functions have to be declared in the first section of an agent specification. The selection functions depend on the agent application, i.e., agent developers provide these functions to their agents in a function library. Initial deep goals are defined in the second section. These goals will be instantiated – together with an applicable plan – in the multistack as separate intentions. Initial facts are simply listed in the third section and added to the agent's knowledge base. Lastly a set of plans has to be defined in order to achieve the goals which are perceived as events during run time execution. The plans are stored in the agent's plan library.

Refined Operational Semantics

We have refined the abstract interpreter, as it is desirable to distinguish between different types of perceived events. In the following we focus on the run time execution of new goals in CASA agents and present central aspects of modeling speculative computations, which appear when a subgoal in a precondition of a plan has to be tested (deep guards).

Suppose there is a (sub)goal selected from the set of perceived events. When there is no applicable reactive plan found for this goal (this can be checked immediately), relevant deliberative and/or communicative plans have to be checked. These speculative computations take place in an additional module called *ContextTest* in order to explicitly separate them from the interpreter cycle. The ContextTest module consists of the two components *DelibStructure* and *CommStructure*. Each element in the *DelibStructure* is composed of a goal event and all relevant deliberative plans to check. Similarly, an element in the *CommStructure* is composed of a goal event and all relevant communicative plans. In the following we briefly illustrate how the two additional components interfere with the interpreter cycle. When no applicable reactive plan is found for a goal g, a new element – composed of g and all relevant deliberative plans for g – is added to the DelibStructure. An element in the DelibStructure

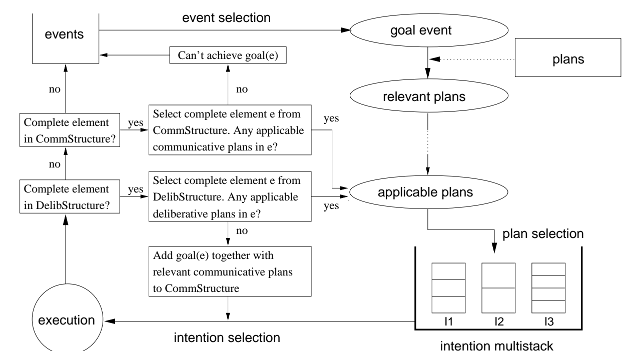

Figure 2: Check Points in Refined Interpreter Cycle

(resp. CommStructure) is said to be complete, when all its relevant plans are checked, i.e., the according plans' preconditions have been tested. As complete elements have to get back into the basic interpreter cy-

cle, we added check points into the original interpreter cycle. The operational semantics are best described by means of figure 2. When there is a complete elment in the DelibStructure, it is checked whether there is at least one applicable plan. If this is true, the interpreter cycle is continued with plan selection and intention instantiation. Otherwise all relevant communicative plans for this goal have to be checked in the CommStructure. Finally, when even no applicable communicative plan can be found, the new goal can't be achieved and an exception has to be raised. It is important to notice that the DelibStructure, the CommStructure and the interpreter cycle operate independently and elements in the two additional components are usually not completed before several rounds of the interpreter cycle have passed.

5 Prototype Implementation

A parser written in JavaCC reads CASA specification programs, sets the initial state of CASA agents and starts the execution on the CASA interpreter. The CASA interpreter is implemented in Java (JDK 1.1.8) integrating modules from the JAM library [Huber, 1999]; JAM's classes for representation and manipulation of facts, actions, simple control structures and external functions were easily adapted to our requirements. CASA agents communicate with the environment using the MECCA framework [Bauer and Steiner, 1998], an agent management system that implements the FIPA ACL (Agent Communication Language) standard. A CASA agent writes messages through a specific communication adaptor to the internal message transport channel of the MECCA system. This adaptor is also reading incoming messages from the transport channel, converting messages into appropriate CASA events and forwards them to the CASA agent. The adaptor allows CASA agents to communicate with any FIPA compliant agent using the MECCA framework. Further details can be found in [Flake, 1999].

6 A Sample CASA Agent

In this section we present a sample CASA agent which was built as part of a MAS that is simulating a color sorting assembly buffer (CSAB) for coloring car bodies. Car bodies are buffered in sortlines before they are transported further to coloring stations. The sortlines shall help to reduce the number of color changes in the coloring stations, thus reducing time and costs of the whole coloring process. Each transport robot, sortline and coloring station of the CSAB is represented by an agent. In this paper we focus on the tasks of a transport robot. In the corresponding specification file we have listed the robot's strategies to achieve certain goals, e.g. getting orders, moving to the car body depot, or determining the most appropriate destination sortline. The following CASA code fragment is describing the central part of the agent's specification:

```
FUNCTIONS: EVENTSELECT:      Event_OldestFirst;
           PLANSELECT:       Plan_HighestPriority;
           INTENTIONSELECT: Int_HighestPriority;
```

```
GOALS:     ACHIEVE busy : 1.0;
FACTS:     ...
           FACT orderAgent "orderManager";
PLANS:

PLAN: {
  NAME:     "Master Plan";
  GOAL:     ACHIEVE busy;
  TYPE:     REACTIVE;
  BODY:     WHILE TEST (TRUE) {
              ...
              PARALLEL
              { ACHIEVE getInfos : 1.0;
                ACHIEVE optimize $number $color
                        $bodyType $sortline : 1.0;
              }
              { ACHIEVE moveToDepot : 1.0;
              }
              IF TEST (!= $number "none") {
                ACHIEVE loadRobot $bodyType
                        $bodyNumber : 1.0;
                ACHIEVE transport $sortline : 1.0;
              }
              ...
            }
  PRIORITY: 1.0;
}

PLAN: {
  NAME:     "get information from other agents";
  GOAL:     ACHIEVE getInfos;
  TYPE:     COMMUNICATIVE;
  PRECONDITION:
            PARALLEL
            { REQUEST $orderAgent "getOrderCount" : 1.0;
              GETREPLY $orderCount;
            }
            { REQUEST $sortline1 "getDesire" : 1.0;
              GETREPLY $color1 $blockLength1 $fillState1;
            }
            ... // request desire from other sortlines
  BODY:     IF TEST (> $orderCount 0) {
              ACHIEVE "getOrders" : 1.0;
            }
            IF TEST (!= $color1 "none") {
              ADD sortline 1 "desire" $color1
                          $blockLength1 $fillState1;
            }
            ... // add desire of other sortlines to KB
  PRIORITY: 2.0;
}
```

The specification defines an initial goal *busy* and an appropriate plan to achieve this goal. When this plan's body is executed, several subgoals have to be achieved in each cycle of the while-loop, some of them even in parallel. Each time after the parallel subgoals are achieved the value of variable *$number* is checked in order to find out whether an order is to be transported or not. In this example different types of plans are needed to achieve the subgoals, e.g. a communicative subgoal to get information from other agents as well as reactive and deliberative plans to achieve the subgoal *optimize*. The CASA code above (right side) denotes a fragment of one possible plan to achieve the subgoal *getInfos*: When the communicative plan's precondition is checked, the agent is requesting information from different other agents: On the one hand the robot agent is asking the order agent about the number of waiting orders. On the other hand all sortlines are asked about their current "desire" to accept new orders. This is described by the color of the last accepted order, the block length (i.e. the number of orders already taken with this color) and the percentage filling state. Finally, we present parts of the different plans to achieve the subgoal *optimize*:

```
PLAN: {
  NAME:         "optimize - 1";
  GOAL:         ACHIEVE optimize $num $color $type $sortline;
  TYPE:         REACTIVE;
  PRECONDITION: GETFACT order $num $color $type "urgent";
  BODY:         ADD currentOrder $num $color $type "urgent";
                ... // determine best destination sortline
  PRIORITY:     2.0;
}
PLAN: {
  NAME:         "optimize - 2";
  GOAL:         ACHIEVE optimize $num $color $type $sortline;
  TYPE:         REACTIVE;
  PRECONDITION: GETFACT sortline $lineNum $color $blockLen 0;
  BODY:         ADD currentDestination "sortline" $lineNum;
                ...// determine appropriate order to transport
  PRIORITY:     1.0;
}
PLAN: {
  NAME:         "optimize - 3";
  GOAL:         ACHIEVE optimize $num $color $type $sortline;
  TYPE:         DELIBERATIVE;;
  PRECONDITION: ACHIEVE investigateEnvironment : 1.0;
  BODY:         EXECUTE findBestCombination $num $color
                                            $type $sortline;
  PRIORITY:     2.0;
}
```

According to the CASA interpreter model the reactive plans are checked first. Plan *optimize-1* is checking whether a fact with an urgent order is in the agent's knowledge base. Plan *optimize-2* is checking whether a fact with an empty sortline exists. Suppose both of the preconditions are true, then because of the plan selection function "highestPriority" the first plan is chosen. Only if both plans' preconditions are false, the third relevant plan *optimize-3* is considered, as it is deliberative: Before the best combination of an order and one of the sortlines can be computed, the robot has to investigate the environment, e.g. find out where the other robots are or determine the distances to sortlines and calculate the estimated time needed to get there. Once it has checked all this, the best combination of order and sortline can be calculated (using a Java function *findBestCombination*). After execution of one of the three plans the four variables *$number, $color, $bodyType, $sortline* are automatically updated in the invoking intention, as they are interpreted as call-by-value-and-result parameters.

7 Summary and Outlook

We have described the design of the agent specification language CASA and the development of an abstract interpreter for CASA agents. CASA combines the BDI agent approach with concepts from concurrent logic programming and allows the design of complex agent strategies on a high level of abstraction. Main differences to other approaches are the concurrent execution of typed, priorized plans and plan elements (preconditions, actions) and the support for speculative computations similar to deep guards in Concurrent Prolog. The presented work also provides a successful case study of how to derive an efficient agent implementation from a formal specification that was taken from another autor's work and extended towards our requirements. First examples include simple applications taken from holonic manufacturing [Flake *et al.*, 1999] and the design of 3D actors in virtual worlds [Geiger and Latzel, 2000]. Future work will concentrate on the development of visual tools for the design of CASA agents and the application in the area of manufacturing and smart 3D graphics.

References

[Bauer and Steiner, 1998] B. Bauer and D. Steiner. MECCA - System Reference Manual. *Internal Documentation*, Siemens AG, Munich, Germany, 1998.

[Bratman, 1987] M. E. Bratman. Intentions, Plans, and Practical Reason. *Harvard University Press*, Cambridge, USA, 1987.

[Ferguson, 1992] A. Ferguson. TouringMachines: An Architecture for Dynamic, Rational Mobile Agents. *PhD Thesis*, University of Cambridge, Cambridge, USA, 1992.

[Flake, 1999] S. Flake. Design and Prototypical Realization of a Specification Language for Intelligent Software Agents. *Diploma Thesis*, University of Paderborn, May 1999 (in German).

[Flake *et al.*, 1999] S. Flake, C. Geiger, G. Lehrenfeld, W. Mueller, and V. Paelke. Agent-Based Modeling for Holonic Manufacturing Systems with Fuzzy Control. *18th International Conference of the North American Fuzzy Information Processing Society, NAFIPS'99*, New York, USA, June 10-12, 1999.

[Geiger and Lehrenfeld, 1994] C. Geiger and G. Lehrenfeld. The Application of Concurrent Fuzzy Prolog in the Field of Modelling Flexible Manufacturing Systems. In L. Sterling, editor: *The Second International Conference on the Practical Application of PROLOG, PAP 94*, London, England, April 1994.

[Geiger, 1998] C. Geiger. Rapid Prototyping of Interactive 3D Animations. *PhD Thesis*, University of Paderborn, September 1998 (in German).

[Geiger *et al.*, 1999] C. Geiger, G. Lehrenfeld, and W. Mueller. Visual Specification, Animation and Illustration of Complex Dynamical Systems. In: *Proc. of the 32nd Hawaiian International Conference on Computer Systems, HICCS*, Maui, Hawaii, January 1999.

[Geiger and Latzel, 2000] C. Geiger and M. Latzel. Prototyping of Complex Plan Based Behavior for 3d Actors. In: *Proc. of the 4th Int. Conference on Autonomous Agents, Agents2000*, Barcelona, Spain, June 2000. (to appear)

[Huber, 1999] J.M. Huber. JAM - A BDI-theoretic Mobile Agent Architecture. In: *Proc. of the Third International Conference on Autonomous Agents*, Seattle, Washington, USA, May 1-5, 1999.

[d'Inverno and Luck, 1998] M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A Formal Computational Model. In: *Journal of Logic and Computation*, 8(3), p.233-260, 1998.

[Mora *et al.*, 1998] M. Mora et al. BDI Models and Systems: Bridging the GAP. In: *Intelligent Agents V. Agent Theories, Architectures, and Languages. 5th International Workshop, ATAL 98*, Paris, France, July 1998. Springer LNAI 1555.

[Mueller, 1996] J. Mueller. The Design of Intelligent Agents: A Layered Approach. Springer-Verlag, 1996.

[Nwana and Ndumu, 1999] H. Nwana and D. Ndumu. A Perspective on Software Agents Research. *Knowledge Engineering Review*, Cambridge University Press, Cambridge, USA, January 1999.

[Rao and Georgeff, 1991] A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI-Architecture. *Tech. Report 64*, Australian Artificial Intelligence Institute, Melbourne, Australia, February 1996.

[Rao, 1996] A.S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. *7th European Workshop on Modeling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

[Schild, 1998] K. Schild. On the Relationship between BDI Logics and Standard Logics of Concurrency. In: *Intelligent Agents V. Agent Theories, Architectures, and Languages.* 5th International Workshop, ATAL 98, Paris, France, July 1998. Springer LNAI 1555.

[Shapiro, 1989] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Surveys*, 21(3) 1989.

[Shoham, 1993] Y. Shoham. Agent-oriented Programming. *Artificial Intelligence*, 60(1), p.52-92, 1993.

[Thomas, 1995] R.S. Thomas. The PLACA Agent Programming Language. In: *Intelligent Agents II - Theory, Architectures, and Languages.* Springer (ATAL 95), 1995.