Running head: RULE CONEPTS FOR BUILDING AGENT ARCHITECTURES

Using Rule-Based Concepts as Foundation for Higher-Level Agent Architectures

Lars Braubach, Distributed Systems and Information Systems Group, Department of Informatics, University of Hamburg, Germany

Alexander Pokahr, Distributed Systems and Information Systems Group, Department of Informatics, University of Hamburg, Germany

Adrian Paschke, AG Corporate Semantic Web, Institut für Informatik, Freie Universität Berlin, Germany

ABSTRACT

Declarative programming using rules has advantages in certain application domains and has been successfully applied in many real world software projects. Besides building rule-based applications, rule concepts also provide a proven basis for the development of higher-level architectures, which enrich the existing production rule metaphor with further abstractions. One especially interesting application domain for this technology is the behavior specification of autonomous software agents, because rule bases help fulfilling key characteristics of agents such as reactivity and proactivity. In this chapter it will be detailed, which motivations promote the usage of rule bases for agent behavior control and what kinds of approaches exist. Concretely, these approaches will be explained in the context of four existing agent architectures (pure rule-based, AOP, Soar, BDI) and their implementations (Rule Responder, Agent-0 and successors, Soar, and Jadex). In particular, it will be emphasized in which respect these agent architectures make use of rules and with what mechanisms they extend the base functionality. Finally, the approaches will be generalized by summarizing their core assumptions and extension mechanisms and possible further application domains besides agent architectures will be presented.

INTRODUCTION AND MOTIVATION

Software agents are computational entities that can act autonomously without user intervention and interact with each other in order to fulfill given tasks. In the recent years, agent technology has evolved in to a large and highly active research field. Therefore, nowadays many different forms of agent applications exist, such as information agents, interface agents, mobile agents, and agents for problem solving (Nwana 1995), many of which also have been put successfully into practice (Jennings and Wooldridge 1998). Besides the application domain, implemented agent systems can also be distinguished by the employed ‘agent architecture’, i.e. the control

structures that facilitate the specification and execution of agent behavior. Often these agent architectures build upon or have been influenced by rule-based technology.

In this chapter, rule-based technology is seen as a declarative approach to programming. On a conceptual level, a rule-based approach enforces a separation of the state of a system (i.e. the working memory) from the behavior (i.e. state transition rules). Different types of rules exist for implementing different types of systems. E.g. in production systems so called forward chaining rules are used, which are composed of a condition and an action part, such that the system will evaluate the condition of a rule and execute the corresponding action, when the condition holds. On the other hand, backward chaining rules (containing antecedence and consequence parts) are used in expert systems and allow deriving new knowledge (consequence) from existing facts (antecedence). For implementing such systems, sophisticated mechanisms have been developed for e.g. representing knowledge in the state, efficiently evaluating rules, deciding which rule to execute when multiple rules match at the same time (conflict resolution) and dealing with knowledge created by rules, which are no longer activated (truth maintenance).

The general approach of this chapter is to present and discuss agent architectures as one very interesting application domain for rule concepts. Therefore, the chapter shows the current state of the art with respect to rules for describing behavior of intelligent agents. One specific objective of this chapter is to show why rule concepts are important for the specification of agent architectures and in what different ways they can be used as basis for such architectures. On a more generic level, the objective of this chapter is to explain how higher-level concepts can be built on rule concepts and what advantages can be drawn from such developments.

The outline of the chapter is as follows. In the next section a short overview of the field of agents and multi-agent systems will be given, with a special focus on the role of agent architectures. Section 3 will provide a description scheme for and a categorization of existing agent architectures with regard to their incorporation of rule-based technology. An in-depth discussion of each of the four categories, including a detailed analysis of one representative in each case, follows in Sections 4-7. Section 8 will provide an outlook on interesting areas of future research before Section 9 closes the chapter with a summary and conclusion.

BACKGROUND ON AGENTS AND MULTI-AGENT SYSTEMS

The field of agent technology emerged during the Nineties of the last century and has its roots in different areas of computer science such as artificial intelligence (AI), software engineering (SE), and distributed computing (Luck et al. 2005). In agent technology, an agent is seen as an independent software entity situated in an environment that is capable of controlling its own behavior (i.e. an agent can act without user intervention). Although agent technology is a very

diverse field with many sometimes quite unrelated sub areas, general consensus exists, that agents can be ascribed the following set of properties (Wooldridge 2001):

- *Autonomy*. An agent decides on its own, how to accomplish given tasks. This is also the case for agents that act on behalf of a user.
- *Reactivity*. An agent continually monitors its environment and automatically reacts to changes in a timely manner, if necessary.
- *Proactivity*. An agent does not only react to stimuli from the outside, but also starts new actions on its own in order to pursue its own (given) objectives.
- *Social ability*. Agents are commonly situated in an environment composed of other (software or human) agents. For accomplishing their tasks, agents can engage in dialogs with other agents and interact in cooperative or competitive ways.

Despite the diversity of the field, two broad strands can be identified, focusing on individual agents and multi-agent issues, respectively. With regard to individual agents, different approaches have been developed for allowing agent programmers to control the practical reasoning processes inside the agents. Hereby, *practical reasoning* means the process of deciding about the action(s) to perform, given current knowledge and (e.g. sensory) input. Multi-agent systems on the other hand, highlight the aspect of *interaction*, that is, communication between agents to coordinate their actions.
Among the prime advantages of the agent metaphor over other approaches are the natural and intuitively understandable concepts. In this respect, agent technology was and continues to be influenced by numerous other disciplines like philosophy or biology (for models of decision processes) as well as sociology or economics (for models of agent interaction). The concepts and technologies that have been developed in the agent field have already successfully been deployed in many different application areas, such as electronic negotiations, shop-floor management, transportation logistics, business process management, computer games and military decision support (Jennings and Wooldridge 1998).

Agent Architectures, Frameworks and Platforms

One important question of multi-agent system construction concerns the mechanism that agents apply for conducting their practical reasoning. For this purpose generic *internal agent architectures* have been devised, which on the one hand describe the building blocks agents consist of and on the other hand an interpreter that works on these building blocks.

As a broad range of different internal agent architectures have been developed, several classification schemes have been proposed in the literature. In the well-known scheme of Wooldridge and Jennings (1995) a distinction is made between, *reactive*, *deliberative* and *hybrid* agent architectures. Reactive agent architectures emphasize the importance of fast reactions to changes within dynamic environments. In the strictest interpretation reactive agents are not allowed to posses a symbolic knowledge representation and act in direct consequence to the perceived percepts. In contrast to reactive agent architectures, deliberative architectures rely on a symbol-based reasoning process, which requires an agent to posses local knowledge and are therefore typically processing-intensive and time consuming. In order to alleviate the weaknesses, that each of the aforementioned architecture types exhibits, *hybrid architectures*

have been devised. They try to unify aspects from both approaches and therefore combine timely reactions with well-planned behavior hybrid architectures have gained high attention in practice and nearly all internal architectures, which are supported by agent frameworks, build on the balanced reactive as well as deliberative actions.

On a technical level, the individual and multi-agent concepts have been concretized into various software frameworks and agent platforms. A detailed overview can be found in (Braubach et al. 2006). In the following special attention will be paid to agent architectures and platforms that incorporate or are based upon some kind of rule mechanism.

RULE-BASED AGENT ARCHITECTURES

In this section the foundations for the following architecture and system descriptions will be laid. The selection of architectures is based on a rule-based categorization, which aims at a distinction of agent architecture with regard to the way they employ rule-based mechanisms. Moreover, a general scheme is introduced, which will be used for the explanation of the architectures. This scheme consists of different steps and as it is applied to all architecture in the same way, the evaluations will be directly comparable.

Architecture and System Categorization

In the literature a multitude of different agent architectures and systems have been proposed. In order to describe these artifacts with respect to rule properties a new classification scheme has been devised (see Figure 1), which allows assigning the representatives to the different categories. The categorization is done along two axes termed *deliberation cycle* and *programming concepts*. The deliberation cycle dimension describes the way the agent interpreter is constructed. In this respect it is distinguished between a controlled rule engine (1), which extends a rule engine with agent concepts, a normal rule engine (2) and an agenda-based interpreter (3), which does not rely on rules at all. Regarding the programming concepts dimension it is differentiated between procedural behavior description (A) based on mainstream programming languages such as Java, rule approaches without (B) and with mentalistic notions (C). Several well-known agent platforms have been assigned to one of the resulting nine categories. From these nine categories only those four will be presented in greater detail, which rely on rule concepts (framed categories). Therefore, agent platforms such as JADE (Bellifemine et al. 2005), Jason (Bordini et al. 2005), and JACK (Winikoff 2005) are not considered in this chapter.

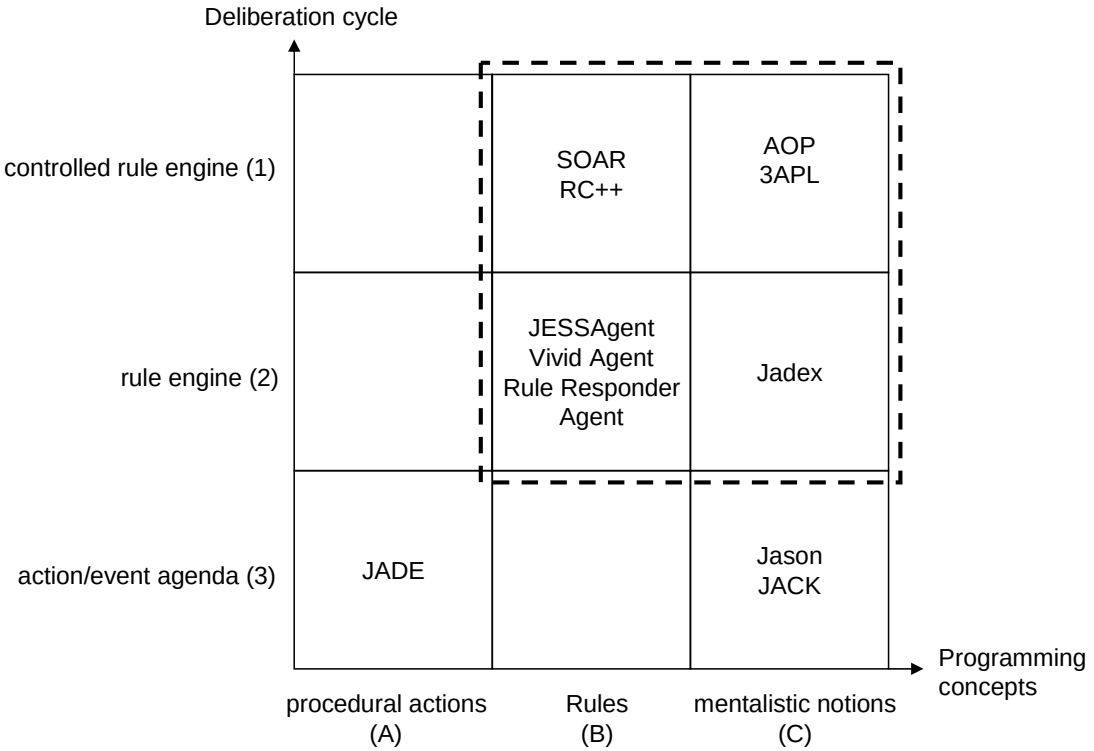

Figure 1: Agent Architecture Classification

The Description Scheme

Each of the architecture descriptions follows the coarse scheme that first the category is introduced, then a specific representative of that category is described in detail and finally a category generalization and conclusion are given. The explanation of the concrete architecture representative is done according the following sections:

- **Architecture and system description.** This section provides detailed information about the selected architecture and system. Therefore, the underlying agent architecture is explained with respect to its components and their interplay. Furthermore, a closer look will be taken at the agent's deliberation cycle, which determines its basic functioning.
- **Agent properties.** In this section, an agent-based perspective will be used for discussing how essential agent properties are supported by the selected architecture. The essential agent properties that are used in this chapter are those defined in the strong notion of agency (Wooldridge 2001). In this definition it is assumed that an agent, in addition of being autonomous, should be capable of *reactive*, *proactive* and *communicative* behavior. Additionally, it is important if *mentalistic notions* can be used for describing the agent behavior in an abstract and intuitive way. For each of these properties it is especially described, which architecture concepts are used for their realization and which aspect rules play in this context.
- **Rule-based mechanisms.** In contrast to the preceding section, this section takes up a rule-based perspective and elaborates on specific rule mechanisms employed within the architecture. The following topics will be discussed: *rule types*, *state representation*, *rule*

evaluation mechanism, *conflict resolution* and *truth maintenance*. With regard to rule types it is discussed, which kinds of rules are used (e.g. production, ECA or deduction rules). The state representation considers the working memory assembly including aspects like fact representation, type system and garbage collection. The rule evaluation mechanism determines in which way the rules are processed, i.e. which algorithm is used and conflict resolution refers to the way how the system decides which activation to fire if more than one is present. Finally, truth maintenance relates to the usage of logical facts and rules.

- **Rule incorporation.** In this section it will be discussed how the rule mechanisms have been integrated into the architecture. This is done on two different levels. On the programming language level it will be discussed, how rules are incorporated into the programming concepts, i.e. in which way a developer gets in contact with rule specifications. On the system level it is considered, which part rules play for the interpreter architecture, i.e. in what way the execution depends on rule mechanisms.

After setting the scene and scheme for discussion, the following sections will deal with the identified categories, respectively.

RULE ENGINES AND RULES (B2)

The rule-based approach can directly be used as basis for agent behavior control. Using this kind of architecture basically requires that the rule base is properly connected with the agent's sensors and effectors in order to allow an agent to receive percepts and execute actions. To be able to exhibit reactive behavior and process incoming messages it is necessary that 1) incoming messages will trigger the execution of processing rules and 2) the external knowledge representation fits to the internal one or is mapped accordingly. Examples of this domain are e.g. JADE/Jess agent (Cardoso 2007), Vivid Agents (Schroeder and Wagner 1998) and Rule Responder (Paschke et al. 2007).

In JADE/Jess an agent is constructed in a way that allows access to the production rule engine (JESS). JADE/Jess agents have e.g. been used in the application domain of agent based scheduling in hospitals (Krempels and Panchenko 2003).Vivid agents are controlled systems whose state comprises the mental components of knowledge, perceptions, tasks, and intentions. Their behavior is represented by means of action and reaction rules, which follow the event-condition-action (ECA) paradigm and are underpinned by formal transition semantics for concurrent action planning. Applications are e.g. the distributed system diagnosis and the monitoring of communication protocols (Schroeder and Wagner 1998).

Rule Responder (Paschke et al. 2007) builds upon existing technologies in multi-agent systems, in particular the Prova Agent Architecture (Dietrich et al. 2003) which evolves from the earlier works in the field, e.g. Vivid Agents (Schroeder and Wagner 1998). It blends and tightly combines them, with modern distributed Web rule and Semantic Web technologies, and service oriented and event driven architectures. Application domains of Rule Responder (http://responder.ruleml.org) include, e.g., virtual organizations (rule-based collaborative agents), Event-driven BPM, Rule Based Service Level Agreements (RBSLA), Health Care and Life Science eScience Service Infrastructure.

In the following Rule Responder architecture will be detailed as representative of this category "B2". Its architecture has evolved from earlier agent architectures, integrates existing agent vocabularies (e.g. FIPA ACL) and uses RuleML / Reaction RuleML (see chapter XXX of this book) for rule interchange and serialization.

Rule Responder

The Rule Responder middleware extends the Semantic Web towards a Pragmatic Web infrastructure for collaborative agent networks where independent agents engage in conversations by exchanging messages and cooperating to achieve collaborative goals (Paschke et al. 2007). Rule Responder utilizes modern enterprise service technologies and the Semantic Web with intelligent agent services that access data and ontologies, receive and detect events (complex event processing), and make rule-based inferences and autonomous pro-active decisions and reactions based on these representations. For a description of the syntax, semantics and implementation of the underlying logical formalisms see (Paschke 2007).

Architecture components

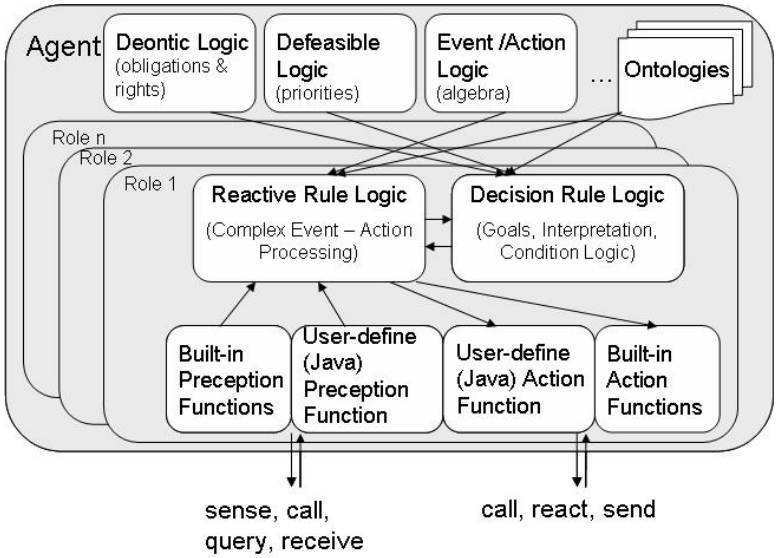

Figure 2: Rule Responder Agent Architecture

The core of a Rule Responder agent is a Prova rule engine (http://sourceforge.net/project/showfiles.php?group_id=50817) which has a modular knowledge base (module = set of facts and rules) to implement several different roles an agent might play. Each role has its own set of reaction rules to react on detected situations (complex events) and its own set of decision rules to interpret goals and derive decisions according to conditional proofs. To sense the environment and trigger actions, query data from external sources such as databases, call external procedural code such as Enterprise Java Beans, and receive / send messages from / to other agents or external services the Prova engine provides a set of built-in functions and additionally can dynamically instantiate any Java objects and call their API methods. Additional libraries can be imported, e.g., to represent rights and obligations of agents,

implement conflict handling rules, or describe complex events and actions. The agent might use vocabularies defined as Semantic Web ontologies (e.g. RDFS or OWL based) or Java class hierarchies to give its' rules a domain-specific meaning. The vocabularies can be used within the conversation with other agents to enable a semantic and pragmatic interpretation.

For the deployment of agents on the Web and for the communication in agent networks the Rule Responder enterprise service middleware is used. The three core parts of the Rule Responder middleware are (1) Reaction RuleML (http://ibis.in.tum.de/research/ReactionRuleML/, see also RuleML chapter XXX of this book) as a common platform-independent rule interchange format to interchange rules, events and data between agent services and other Semantic Web tools, (2) a highly scalable and efficient enterprise service bus (ESB) as agent/service-broker and communication middleware (Paschke et al. 2007a), and (3) Prova rule engines as platform-specific (Java) rule-based agent execution environments.

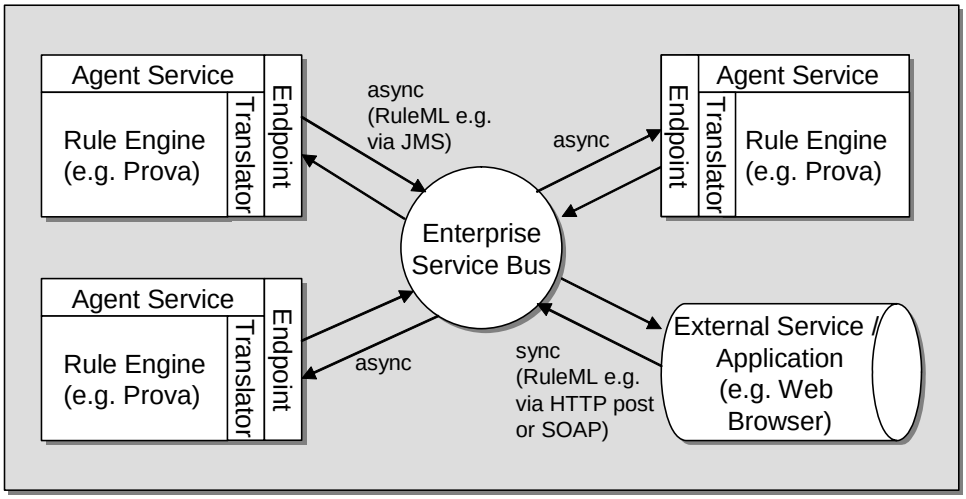

Figure 3: Rule Responder Middleware

The agents are deployed as distributed web-based services on an ESB. They have local autonomy and are decoupled via exchanging Reaction RuleML messages either asynchronously or synchronously over the ESB, or in an ad-hoc conversation directly between two agents using agent protocols such as Jade-HTTP. Arbitrary transport protocols can be used to transport rule sets, queries and answers as payload of Reaction RuleML event messages between the agent services. Each agent dynamically imports or pre-compiles and loads a possibly distributed and modular rule base (represented as one or more Reaction RuleML scripts published on the web or stored locally, e.g., in a (XML) database or file), which implements the decision and (behavioral) reaction logic of the agent.

Deliberation Cycle

Basically, a Rule Responder agent follows the sense-reason-act pattern. However, Rule Responder does not define one particular deliberation cycle, but allows configuring a Rule Responder agent with user-defined conversation-based negotiation and coordination protocols or workflow patterns. Via constructs for asynchronously sending and receiving event messages an

agent interacts with the environment. The payload of incoming event messages is interpreted with respect to the local conversation state, denoted by the conversation id, and the pragmatic context which is defined by a possibly complex pragmatic performative(s). Depending on the pragmatic context the message payload is used e.g., to update the internal knowledge of the agent (e.g. adds new facts or rule sets), add new tasks (goals), or detect a complex event pattern (from the internal event instance sequence). Several reasoning processes might run in parallel local to their conversation flows. Inactive reactions (conversation partitions) are removed from the system. Self activations by sending a message to the receiver "self" are possible. Several expressive logic formalisms are provided for updating the knowledge base (transactional update logic), defining and detecting complex events (complex event algebra) (Paschke et al. 2007a), handling situations (states) (event calculus), and for reasoning (e.g. deontic logic for normative reasoning on permissions, prohibitions, obligations) (Paschke 2007) and planning (abductive reasoning on plans and goals (Paschke and Bichler 2005).

Agent properties

Reactive behavior. External events (changes in the environment) and data from data sources such as Web resources, relational databases and external tools, web services or (Java) object representations can be dynamically accessed and integrated during runtime via the expressive homogenous and heterogeneous integration interfaces of Prova, or via publish-subscribe or reactive monitoring processes using the ESB, event / action messages and reaction rules (Paschke et al. 2007a). The logic for reactivity of the agent is implemented in terms of reaction rules and conversation-based messaging reaction rules. Complex event patterns can be defined in terms of an event algebra (based on an interval-based event calculus variant) and in terms of messaging workflow-like conversations

Proactive behavior. The decision and planning logic for (semi-) autonomous and pro-active decisions and actions, e.g. starting a new sub-conversation, is implemented in terms of derivation rules and messaging reaction rules. An interval-based event calculus axiomatisation (Paschke et al. 2007a) allows representing the agent's states as changeable fluents which are initiated or terminated by occurred (complex) events. For a set of believed (planned) goal states a set of hypotheses (~a plan represented as a sequence of planned events that need to occur) can be determined by an abductive extension of the event calculus meta program. (Paschke and Bichler 2005). Contradictions are detected with respect to predefined (application-specific) integrity constraints and test suites (set of test cases) from which violation free (hard integrity constraints) and covered (level of coverage according to the covered test cases) plans are selected (using a priority based defeasible logic) (Paschke 2007).

Communication. Typically, Rule Responder agents form collaborative networks where the agents interact in conversation dialogs which might follow a pre-defined negotiation or coordination protocol (see Paschke et al. 2006). This might also include negotiation and discussion about the meaning of ontological concepts, since agents might use their own micro-ontologies / vocabularies and must agree on relevant shared concepts to enable an efficient communication and knowledge interchange between the agent nodes.

Mentalistic notions. Rule Responder does not define a specific set of mentalistic notions as first-class programming constructs. Instead, interchanged messages beside the conversation's meta data and payload also carry the pragmatic context of the conversation such as communicative situations / acts, mentalistic notions, organizational and individual norms, purposes or individual goals and values. For instance, a standard nomenclature of pragmatic performatives which can be integrated as external (Semantic) vocabulary/ontology is defined by the Knowledge Query Manipulation Language (KQML) (Finin et al. 1993), the FIPA Agent Communication Language (ACL) which gives several speech act theory-based communicative acts or the normative concepts on obligations, permissions and prohibitions of Standard Deontic Logic (SDL).

Rule-based Mechanisms

Rule types. Rule Responder on the basis of the RuleML language family (see chapter XXX) and the Prova rule engine supports different rule types:

- *Derivation rules* to describe the agent's decision logic
- *Integrity rules* to describe constraints and potential conflicts
- *Normative rules* to represent the agent's permissions, prohibitions and obligation policies
- *Global reaction rules* to define global reaction logic which are triggered on the basis of detected complex events (event patterns defined by an event algebra)
- *Messaging reaction rules* to define conversation-based workflow reaction and behavioral logic based on complex event processing

State representation. States, denoting changeable agent's properties, such as its obligations, are represented as explicit event calculus fluents which are initiated and terminated by events. Updates to the internal knowledge base of the agent are treated as modules which transit the knowledge base to an extended knowledge state, where a set of rules or facts are added or removed to the previous knowledge state. Each module in the knowledge base (add rule/fact set) is identified by a unique module object id (oid). A complete sequence of state transitions, e.g. a sequence of updates in an execution path of firing reaction rules, might be rolled-back by backtracking the update operators on the remembered sequence module oids (Paschke et al. 2007a).

Conflict resolution. To overcome arising conflicts, preserve integrity of the knowledge base in each state and ensure an unique declarative outcome of update sequences (active rules) or complex updates (complex update actions) Rule Responder supports integrity constraints (ICs) and test cases (TCs). ICs and TCs are used to verify and validate the actual or any hypothetically future knowledge state (Paschke 2007).

Truth maintenance. Truth is defined locally with respect to an explicitly closed scope (constructive view) on the agent's knowledge base (KB). Scoped queries apply on the constructive views which are defined over module(s) or rule sets, e.g. all rules, which are written by a particular author (defined in the rule meta data). By default, un-scoped queries (goals) apply globally on the complete knowledge base of the agent, i.e. the current knowledge state. Hypothetical updates are treated as uncommitted modules in the KB which are tested against the

defined integrity constraints and might be rolled-back if they violate the integrity of the KB (Paschke 2007).

Rule Incorporation

In Rule Responder rules of various types (derivation, integrity, reaction, messaging) are first class citizens to program agent behavior and agent decision logic. The core Prova rule language and engine is not a specific agent language but a highly expressive, hybrid, declarative and compact rule programming language which combines the declarative programming paradigm with object-oriented programming (in particular Java) and the Semantic Web approach.
Due to the natural integration of Prova with Java, it offers a syntactically economic and compact way of specifying agents' behavior while allowing for efficient Java-based extensions to improve performance of critical operations. Via its capabilities to integrate external vocabularies such as Semantic Web ontologies and Java type systems the rule language can be custom-tailored to a specific agent vocabulary giving the used rule constructs a specific meaning with a clearly defined semantics, e.g., based on the FIPA vocabulary. In addition to backward-reasoning derivation rules, which for instance are used to implement decision or planning logic, Prova supports messaging reaction rules specifying the sequences or branches of actions that an agent responds with upon detection of a matching pattern in the inbound messages which the agent receives from other agents or the environment (other tools, services). Several meta-programs written as Prova scripts exist (Paschke 2007), e.g., for deontic reasoning about obligations, prohibitions and permissions of (agent) actions. These scripts can be added selectively as modules to the rule based of an agent hence providing the needed (meta) reasoning, planning and reaction capabilities to formalize the agent behavior in terms of rules.
On the platform-independent and web-based rule interchange level, Reaction RuleML is used. Via translator services the interchanged RuleML messages are translated into the platform-specific execution syntaxes of the agents' rule execution environments such as Prova.

Conclusion

In this section rules are directly employed to implement the agents' decision and reaction logic. Expressive declarative rule languages in combination with external data models and vocabularies are used to formalize coordination mechanisms and agent behavior. The main design philosophy behind these approaches is the minimalism and simplicity of the additionally introduced agent-specific syntax extensions to the core rule language.

Below, advantages and disadvantages of the B2 category are presented:
+ Highly-expressive rule languages allow declarative programming of complex agent coordination mechanisms and agent behavior; representation restrictions are only due the well-known trade-off between expressiveness and computational complexity
+ Rules can be serialized in common rule interchange formats such as RuleML / Reaction RuleML, published on the Web and interchanged between different platform-specific rule engines (given these execution environments, such as Prova, provide adequate expressiveness)
- Long learning curve for users / agent engineers, due to the highly flexible and expressive rule languages

- High-level conceptual abstractions which should be provided as specific agent programming constructs in the language need to be explicitly introduced either by meta programming techniques or via integration of external functionalities and vocabularies.

Generalization

The approaches in this category directly employ general rule languages (e.g. Reaction RuleML) to implement agent functionalities. Extensibility of the general rule language is achieved by integrating external vocabularies and external data and functionalities, e.g. object-oriented procedure calls. Rule languages are primary used in the expert system field. In this context various application domains such configuration, diagnosis and interpretation of data have been considered. For an extensive survey see Giarratano and Riley (2005).

CONTROLLED RULE ENGINES AND MENTALISTIC NOTIONS (C1)

This category encompasses approaches that aim at introducing abstract mentalistic notions as programming language constructs. Each of these approaches proposes a specific concerted set of mental state components and introduces a language with specific types of rules to operate on these components (e.g. commitment rules, which operate on commitments). As a natural way of controlling the relation between different types of mental components and rules respectively, these approaches are not implemented as a simple rule base, but use the specific rules only as part of an extended interpreter architecture.

All approaches in this category are intended to be general purpose agent programming languages. Nevertheless, due to the rules operating directly on the mental state of the agent, these approaches are best suited for agents operating in dynamic environments, where quick reactions to environmental changes are advantageous. Such environments are common, e.g. for agents operating on mobile devices such as PDAs or cell phones and for controlling autonomous robots.

Prominent representatives of this category are the AOP and the 3APL/2APL language families. 3APL (“An Abstract Agent Programming Language”) and its successor 2APL (“A Practical Agent Programming Language”) are developed at the University of Utrecht (Hindriks et al. 1999, Dastani et al. 2007). In addition to implementing agent applications, these languages are also used for teaching purposes. In contrast to 3APL/2APL, AOP languages are developed by different groups for different purposes including academic and commercial projects. Because of the greater diversity, in the following, the AOP approach will be presented in more detail.

AOP

Agent Oriented Programming (AOP, cf. Shoham 1993) is one of the first approaches being motivated by the desire of using agent-oriented concepts as part of a high-level programming

language. In the AOP paradigm an agent is cast as a special type of object (in the object-oriented sense). The state of an agent (unlike an arbitrary object) has a fixed and predefined structure, which only incorporates mental attitudes, such as beliefs and obligations. The approach is backed by the assumption that taking an *intentional stance* (Dennett 1971) towards a software system under development is advantageous in certain scenarios (McCarthy 1979). Therefore, the representation of mental attitudes as first-class programming constructs is seen as a desirable extension of traditional software engineering practices.

Three basic attitudes are proposed by Shoham for representing an agent's mental state: beliefs, obligations and capabilities. These are represented using a formal temporal language that contains the modal operators $BEL^{t}_{a}\varphi$ ("agent *a* currently believes that sentence φ is or will be true at time *t*"), $OBL^{t}_{a,b}\varphi$ ("agent *a* is obliged to agent *b* to do or bring about φ at time *t*"), and $CAN^{t}_{a}\varphi$ ("agent *a* is able to do or bring about φ at time *t*"). Sentences (φ) are defined in a recursive way, such that e.g. beliefs about obligations of other agents can be expressed. Although some other common attitudes can be represented by combinations of the basic attitudes (e.g. commitments can be seen as obligations of the agent to itself), Shoham sees this basic model only as a starting point and therefore does not claim completeness.

Besides such a *formal language* for representing the mental state of an agent as stated above a concrete *programming language* is needed, which follows the semantics of the formal language and allows to specify the agent behavior using additional constructs. Shoham concretizes his initial AOP approach by introducing a simple programming language called Agent-0 (Shoham 1993). Based on Agent-0 other AOP languages (cf. Figure 4) have been conceived that introduce additional constructs for making programming more convenient. In the second generation of AOP, PLACA (Thomas 1995) extends Agent-0 with planning features, while Agent-K (Davies and Edwards 1994) supports KQML (Finin et al. 1993) as a standardized agent communication language. Agent-K has been further extended to GOAL (Byrne and Edwards 1996), which incorporates planning similar to PLACA, but also on a multi-agent level (e.g. group goals, which comprise commitments of several agents). Successors of PLACA are RADL, which is the language of the commercial AgentBuilder toolkit (http://www.agentbuilder.com/) and AF-APL, belonging to the open source AgentFactory framework (Ross et al. 2005). Both languages try to advance the practical usability of the language, e.g., by providing convenient mechanisms for integration with Java. Due to its simplicity, we will use Agent-0 to discuss the basic principles of the AOP architecture and refer to the advanced features of the newer languages as necessary.

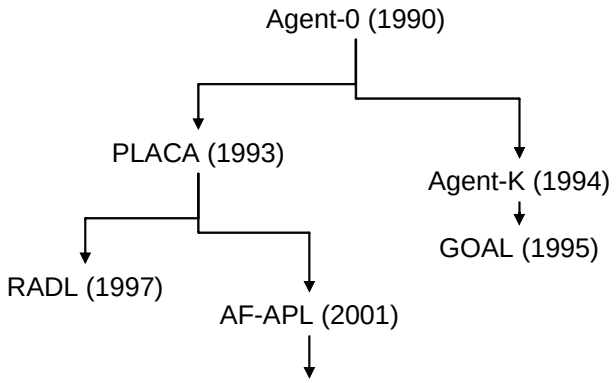

Figure 4: AOP language family

Architecture components

The separation between a (more or less formal) state representation and programming constructs acting upon that state quite nicely fits to the idea of a rule-based system. Not surprisingly, all proposed AOP languages employ a rule-based specification of agent behavior. In Agent-0, only one rule type is supported: so called *commitment rules*. These rules are a special form of event-condition-action (ECA) rules. The event part basically describes patterns of messages that may be received from other agents. The condition part refers to the current mental state (e.g. the beliefs) of the agent. The action part is quite restricted compared to traditional ECA-rules: instead of directly executing actions, a rule may only create new commitments towards actions, associated to some specific time point. As commitments are just added to the mental state of the agent, the triggering of a rule does not lead directly to the execution of actions. Instead actions are queued as commitments in the mental state of the agent and automatically executed, when their time is due.

As an example, a simple library agent is shown in figure 5 (adopted from Shoham 1990 and Thomas 1993). It has the abilities to shelve books (line 2) and Xerox documents (line 3). Initially (at time 1), it knows the customers Alice (line 6) and Bob (line 7). Finally, it has one commitment rule to do whatever requested from customers. Whenever the agent receives a request to perform some action at a given time (event pattern in line 10) and the sending agent is known to be a customer (mental condition in line 11) the agent will adopt the commitment to execute the action for the sender as requested (line 12).

```
01: CAPABILITIES :=
02:   ((shelve ?x) (book ?x))
03:   ((xerox ?x) (document ?x))
04:
05: INITIAL BELIEFS :=
06:   (1 (customer alice))
07:   (1 (customer bob))
08:
09: COMMITMENT RULES :=
10:   (COMMIT   (?sender REQUEST (DO ?time ?action))
11:             (B (now (customer ?sender)))
12:             (?sender (DO ?time ?action)))
```

Figure 5: Example of a library agent

The architecture of an AOP agent is illustrated in figure 6. Most important part of an agent program is the set of commitment rules (bottom right). Agent-0 provides only a few basic actions like sending messages to other agents. Domain specific actions (e.g. a move operation for a robot) can be defined using so called capabilities (bottom left). Capabilities specify a guard (i.e. a mental condition), which must be true for executing the associated action. The actions itself are

specified by mechanisms outside the agent language such as a procedure in a traditional programming language. Both commitment rules and capabilities are specific at design time. At runtime, the agent continuously updates it's initially given beliefs (top left) and creates/executes/drops commitments (top right) according to the commitment rules and the current situation. At the heart of the agent, there is a clocked interpreter that is responsible for incrementing the current time and managing the other components.

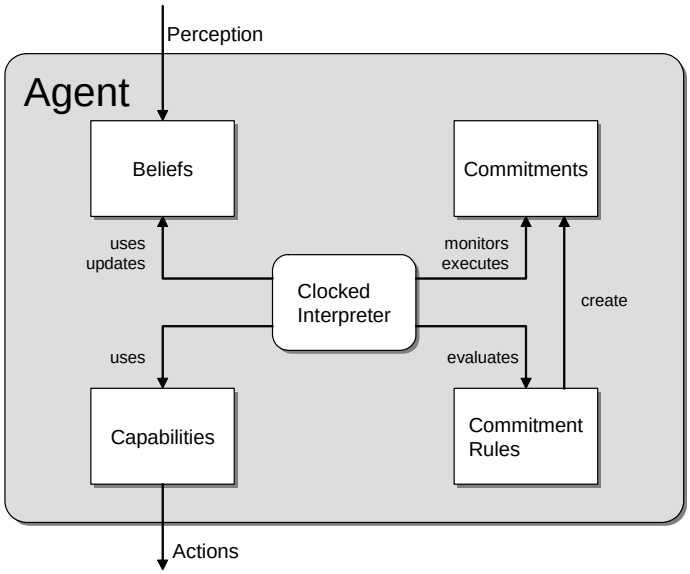

Figure 6: AOP agent architecture

Deliberation cycle

The deliberation cycle in AOP basically consists of three steps:

1. Updating beliefs
2. Updating commitments
3. Executing due commitments

In the first step, the interpreter takes care of updating the agent's beliefs when receiving INFORM messages from other agents. While the original Agent-0 interpreter just adds any information into beliefs, newer languages provide advanced mechanisms like considering authenticity and authority of agents (AgentBuilder) or leave the interpretation of message contents up to the agent developer (AgentFactory). In the second step, the interpreter checks all commitment rules and creates new commitments when required. Moreover, existing commitments are dropped, when the agent no longer believes to be able to perform the desired action or when it is "unrequested" to perform the action (i.e. the original requestor revokes the request). In the third step, the interpreter checks, whether some of the current commitments are due at the current time point and executes the associated actions.

Agent properties

Reactive behavior. Reactive behavior can be expressed quite naturally in AOP. Although rules do not directly lead to actions but only to future-directed commitments instead, the keywords *self* and *now* allow to specify commitments to actions, which should be executed instantaneously. Therefore a direct ECA-style of reactive agent programming can be done in AOP quite easily.

Proactive behavior. The basic AOP model has only very limited support for pro-active behavior, in the sense that the agent will not adopt a commitment to an action when it already has a commitment to refrain from that action. Newer AOP languages support commitments to complex objectives in the sense of procedural goals that an agent pursues by executing predefined sequences of actions (called plans), but these goals usually lack a declarative aspect (i.e. a condition that can be used to check, if the goal is achieved).

Communication. Communication is an important aspect of AOP agents. AOP languages provide primitives for communicative actions as well as message patterns that can be used in the event part of a commitment rules. Moreover, the interpreter usually provides mechanisms that automatically deal with incorporating information received via messages into the beliefs of an agent.

Mentalistic notions. The semantics of the mentalistic notions in AOP (e.g. beliefs or commitments) follows an intuitive interpretation, which is therefore quite easy to understand. Newer languages extend the initial basic model by introducing further notions such as goals or plans. Although a wide range of complex behavioral patterns can be developed with most AOP languages, these approaches remain on a quite technical level and do not integrate the mentalistic notions in an abstract, self-contained model of practical reasoning or problem solving.

Rule-based mechanisms

Rule types. The core concept of AOP are the commitment rules, which are a specialized form of ECA-rules. In the basic model, these rules are restricted in the sense that only the creation of commitments is allowed in the action part. Newer AOP languages partially relax this restriction and allow also sequences of actions, where actions may include sending messages, updating beliefs or creating new commitments.

State representation. Conceptually, the state of an AOP agent is represented using a first order logical language, extended with modal operators for the mental attitudes. Such a representation, in principle, allows reasoning over arbitrary complex logical statements. Nevertheless, because such a representation is usually computationally intractable, substantial simplifications are applied in the implemented interpreters, e.g. allowing only atomic statements for facts.

Rule evaluation mechanism / Conflict resolution. Most AOP implementations follow a naive approach, i.e. in each execution pass of the deliberation cycle, all rules are tested and the activated rules are executed in the order they are specified in the agent program. Due to the moderate number of rules in most agent programs, the performance of such an approach has been found to be sufficient for many practical settings.

Truth maintenance. No AOP system to date employs or provides truth maintenance functionality.

Rule incorporation

In AOP, rules are the primary programming construct for representing the behavior of agents. Unlike pure rule-based programs, AOP languages introduce a set of agent-oriented concepts and constructs with a predefined semantics. In this respect, the mentalistic notions of the architectural model help structuring the rule based program into intuitive agent-oriented concepts. I.e. the working memory is structured into flat factual knowledge (beliefs) as well as knowledge about the agent's skills (capabilities) and intended actions (commitments). Additionally, capabilities are used as an access point to functionality that is implemented in imperative programming languages. Another difference to pure rule-based programs is that the form of rules is restricted. This means that the mentalistic notions that may be used in the event, condition and action parts are fixed in a way to support the intended semantics of the agent model (e.g. in actions, only commitments may be created). Finally, the interpreter supports the developer by automatically performing additional operations related to the semantics of the constructs (e.g. updating beliefs and executing or dropping commitments automatically).

Conclusion

This section has introduced approaches, which strive to provide high-level abstractions as first-class programming constructs. To implement the resulting languages, interpreters are developed, which provide additional functionality besides the simple execution of rules. These approaches can be seen as a restriction and specialization of rule-based systems for certain application domains, where the abstractions and constructs provided by an approach will be beneficial.

Below, advantages and disadvantages of the C1 category are presented:
+ The main motivation of the approach is to provide high-level conceptual abstractions, which are made available to the programmer directly as first-class programming language constructs with an intuitive semantics. The resulting languages are therefore quite easy to learn and the built-in semantics allows quickly developing complex applications with only a few statements.
+ Due to still using rules as the main programming construct, approaches from this category are especially well suited for encoding reactive behavior. Different reactions to varying situations can be intuitively described and easily maintained by adding or removing behavioral rules.
- The rule-based style of behavior representation has led to architectures, which have difficulties representing long term activities. Although this is not an inherent problem of the category per se, all currently available approaches lack high-level concepts for such long term activities.
- The restrictions imposed by the architecture prevent the developer from using arbitrary rules. Therefore, the expressive power of the language is reduced in favor of better supporting a certain application domain.

Generalization

As a generalization, we consider approaches that introduce higher-level concepts (not necessarily mentalistic notions) and provide an extended or controlled rule base for interpreting programs using these contructs. One emerging research/application field, where this kind of approach is quite common is the area of Complex Event Processing (CEP, see e.g. Luckham 2002, Fiege et al. 2006). Thereby, a complex event is a composition of a specific pattern of many simpler events. The purpose of CEP is to filter and extract meaningful domain-level events from low-level occurrences (e.g. a missed deadline might be common while three missed deadlines in a row indicate a serious problem). Complex events are defined by event patterns, which are described in en event algebra with operators to relate events top each other (e.g. *between*, *before*, *after*, …). CEP languages extend traditional ECA-rules by providing the operators of the event algebra as first-class programming constructs. For implementing CEP languages, the rule-based execution needs to be extended, because the system has to keep a repository of past events for all incomplete matches. Moreover, the system has to make sure that past events are automatically removed from the memory, when they are no longer needed.

CONTROLLED RULE ENGINES AND RULES (B1)

This category contains rule engine based architectures, which have changed the typical way of iteratively executing activations via an agenda. Nonetheless, the offered programming concepts have not been changed, i.e. the agent behavior specification is mainly done by programming traditional rules.

As Figure 1 shows, two typical representatives for this category are RC++ (Wright and Marshall 2003) and SOAR (Lehman et al. 1996). The motivations behind those approaches are quite different. RC++ is an extension to the C++ programming language, which incorporates rules directly into the base language. It therefore realizes a conservative extension approach, which aims at a tight integration with the underlying procedural core language. The RC++ language is a general purpose language, but has been designed with a clear application focus in mind. It should facilitate the programming of game AI for the game console. In contrast, SOAR has been developed to be a general problem solver for arbitrary complex problems. Hence, the SOAR architecture has primarily been used for knowledge-rich intelligent agent applications. Examples include intelligent control, natural language understanding, human behavior experiments and simulation (Wray and Jones 2006). As SOAR has advanced the ordinary rule interpretation in many directions, it will be explained in more details in the following.

SOAR

The official project start of SOAR (originally for State, Operator And Result) can be dated back to 1983. Initially, the work on SOAR was influenced strongly by Alan Newell and his former students John Laird and Paul Rosenbloom. In the course of time an international research community emerged that jointly works on extensions and enhancements for SOAR.

One main motivation for the conception of SOAR was Newell's (1990) idea of a Unified Theories of Cognition (UTC). Such an UTC should encompass all relevant areas of human abilities into one major psychological theory, which is able to explain e.g. problem solving, learning, perception, language understanding, emotion and dreaming aspects. In order to advance research in this direction and to overcome the multitude of existing psychological micro-theories, each applicable for one very specific phenomenon, he proposed SOAR as a promising UTC candidate. The SOAR architecture has been devised to be able to cope with all tasks that could be principally posed to an intelligent agent, i.e. it should be able to handle simple routine jobs as well as complex problem solving tasks using different solution strategies. Furthermore, interaction with the environment and learning from past experiences should be facilitated by the architecture. Despite these challenging requirements it has been a primary design principle to keep the architecture complexity as small as possible (*parsimony principle*). Therefore, the number of architecture mechanisms has been reduced to a minimum, i.e. the architecture comprises only one representation for long-term and short-term memory as well as one mechanism for goal construction and learning.
Being a multi-disciplinary effort, different viewpoints exist regarding SOAR as a theory of general intelligence or human cognition, and also as an agent architecture or a programming language. Discussing SOAR as an agent architecture is useful, because SOAR demonstrates, how algorithms from artificial intelligence can be integrated into an architecture that allows building software systems exhibiting general intelligent behavior (Wray and Jones 2006). This viewpoint will therefore allow comparing SOAR to the other agent approaches described in this chapter.

Architecture components

The SOAR architecture (Lehman et al. 1996, Laird et al. 2006) is based on the hypothesis that behavior determination should be done by manipulating knowledge only (*physical symbol system hypothesis*). SOAR proposes state space searching as core mechanism for problem solving. Primary concepts are therefore problem spaces and operators, which are applied to transfer one state to the follow-up state. Declarative goal achievement can be realized by specifying a target goal state and apply an operator search until the desired state has been reached.

In Figure 7 the SOAR architecture, its main components and their interplay are depicted. The interpreter performs the problem space transformations by operating on its memories. Generally, two different kinds of memories are distinguished. The working memory contains information about the current situation of the agent including sensor data, results of inferencing as well as active operators. SOAR's working memory is hierarchically structured and contains substates for each subgoal, which are connected via state objects. The long-term memory of a SOAR agent contains its problem solving capabilities in form of production rules. In SOAR three different kinds of rules are supported. *Operator proposal rules* are used for finding applicable operators. Given that an operator was selected, *operator application rules* help to realize its behavior. The third kind of rules is *state elaboration rules*, which are used for deriving new knowledge from the existing knowledge in the working memory. In the following a more detailed review of the interpreter's deliberation cycle will be given.

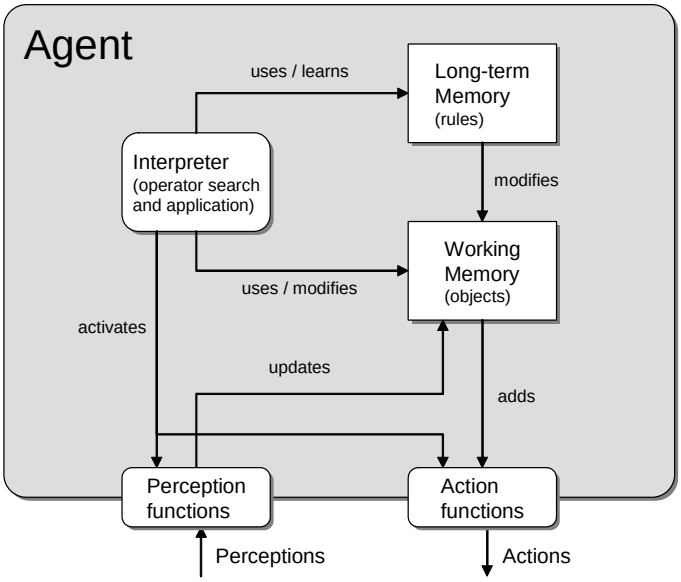

Figure 7: SOAR agent architecture

Deliberation cycle

SOAR's deliberation is performed in a typical perceive-decide-act cycle. It consists of the following five phases, which are continuously executed:

Input phase: In the input phase external data from the environment is transferred to the SOAR working memory. The external data can be in arbitrary format and specific input mapping functions need to be provided, which transform the external representation to the internal triple representation. This design separates the perception from the cognitive processes and allows a concurrent execution of both.

Proposal phase: In the proposal phase two different activities are performed. On the one hand all activated state elaboration rules are fired in parallel. Given that these rules lead to new activations, these will also be applied. The process stops when no activations are available any more and *quiescence* has been reached. In addition operator proposal rules are fired. These rules have the one and only purpose to suggest applicable operators.

Decision phase: The aim of this phase is to select one of the proposed operators for execution. For this purpose a built-in evaluation mechanism is applied, which has the task to weigh operator preferences and select the best one. If the decision yields a single winning operator, this operator will be marked for application. Otherwise, it is assumed that the knowledge for operator selection is insufficient and additional knowledge needs to be obtained. This situation is called an *impasse* and can basically have three different reasons:

- No operator can be selected, because no operator has been proposed.
- No operator can be selected, because multiple equivalent operators have been proposed.
- An operator can be selected but the agent is missing information for operator execution.

In case of an impasse the interpreter automatically tries to resolve it by using the built-in subgoaling mechanism (*automatic subgoaling hypothesis*). For the subgoal a new problem space

will be created, which is a substate of the original state. As the creation of subgoals can be applied in each problem space this may lead to arbitrary deep goal hierarchies. When the subgoal processing generates information suitable for resolving the tie, the results can be merged into the higher-level state leading to a removal of the substate. In order to avoid the possibly expensive search for resolving the same tie more than once, SOAR employs a learning mechanism called *chunking*. This mechanism enables a SOAR agent to summarize and generalize a problem solving strategy used in the substate to a new rule. The rule will be added to the long-term memory and therefore can be utilized in future problem solving activities.

Application phase: In this phase the execution of the selected operator will be performed. As the operator is itself part of the working memory it can be used in the condition part of operator application rules. Similar to the proposal phase, all activated application rules are fired in parallel until quiescence has been reached.

Output phase: The output phase serves for propagating information about actions to the environment. Similar to the input phase user-defined functions are used to transform the internal knowledge representation to an external one.

Agent properties

Reactive behavior. In SOAR reactive behavior can be encoded via operator proposal and application rules. New environmental information is consumed via user-specific input functions, which add the new information to the input-links of the state. Operator rules can be specified with condition parts that are sensitive to those changes. They produce operators and provoke the invocation of application rules. These application rules can trigger external actions by adding data to the output-links of the state, which will be interpreted by the also user-specified action functions once in every deliberation cycle.

Proactive behavior. Proactive behavior can be achieved by goal directed search and planning strategies. For this purpose the target state to achieve can be observed by a dedicated monitoring rule. In addition, the goal can be represented e.g. via state attributes that guide the operator proposal process. Whenever a goal is to be achieved, dedicated search rules and possibly heuristic rules for controlling the search directions can be employed. In this respect, automatic subgoaling in impasse situations and the distinction between proposal, selection and application rules facilitates the implementation of proactive behavior.

Communication. In SOAR, interaction and communication facilities have to be provided on the user level by exploiting the in- and output functions. On top of these, simple environment-based agent interaction, e.g. in the TanksSoar Game, as well as sophisticated social models, such as the STEAM teamwork model based on the Joint Intention theory, can be implemented.

Mentalistic notions. Mentalistic notions are not directly used in SOAR. Designing SOAR solutions requires thinking in terms of problem spaces and operators, whereas the programming of SOAR agents is done with (different kinds of) production rules only. Wray and Jones (2006) state that this minimalist design of the programming concepts leads to an "'assembly language' for intelligent systems".

Rule-based mechanisms

Rule types. SOAR is based on forward-chaining and production rules are used only. In contrast to normal rule-based systems, a further distinction of generic rule types has been introduced. In this respect it is distinguished between operator proposal and application rules as well as state elaboration rules. These different kinds of rules help in realizing complex search-based behavior and establish a meaningful separation of concerns between what can be done (operator proposal) and how it can be accomplished (operator application).

State representation. The basic representation of facts is the same as in the OPS5 expert system (Brownston et al. 1985) SOAR is based on. The working memory consists of facts, which are represented by object-attribute-value (OAV) triples. A value of a triple can either be a constant or an identifier of another object. SOAR uses a weak type system, which does not require types to be defined before usage, i.e. one can add new attributes to an object as needed. On the contrary this also means that the system cannot apply type checks and typing errors cannot be automatically detected. Despite these similarities to OPS5, SOAR also exhibits important distinctions. One important aspect is the hierarchical organization of substates according to the subgoaling mechanism. A substate represents a new problem space, which can contain arbitrary new information and is additionally linked to the superstate.

Rule evaluation mechanism. The rule evaluation is based on the RETE algorithm (Forgy 1982). In the context of the SOAR project also alternatives (e.g. TREAT, Miranker 1989) have been evaluated, but the RETE algorithm proved to be the most effective one. In the course of the development several improvements to the basic RETE algorithm have been found and applied (e.g. Doorenbos 1995).

Conflict resolution. Due to the specific execution mode of SOAR, conflict resolution is handled completely different from standard rule-based systems. In the proposal and application phases the rules are fired in parallel until quiescence is reached. This renders conflict resolution within these phases unnecessary. On the contrary, within the operator selection phase an elaborated conflict resolution mechanism is applied for choosing among the proposed operators, i.e. conflict resolution is not applied with respect to activated rules but to operators. The resolution strategy is predefined and is based on operator preferences, which are assigned to the operators within the proposal phase. The resolution has then the task to determine the best operator. If this is impossible due to an impasse situation, subgoaling will be applied for resolving the tie.

Truth maintenance. In SOAR between permanent and temporary working memory elements is distinguished. State changes that are effect of operator execution in the operator application phase are considered as permanent and are said to be o-supported (operator-supported). However, all state changes that occur within the proposal phase are volatile and rely on the production that produced them to continue to match. SOAR supports this by logical facts that exist only as long as the productions condition is valid. If it gets invalid the fact is retracted automatically.

Rule incorporation

At the programming language level SOAR incorporates rules as primary programming concept. In contrast to traditional rule-based approaches, rules are used in SOAR for realizing higher-level architecture concepts. SOAR does not strive for an integration of the SOAR language with common mainstream programming languages, i.e. integration with external systems is possible only by using the input and output functions. This also means that SOAR actions, i.e. the right hand side of a rule, can only affect the working memory of the agent.
At the interpreter level, SOAR builds on an enhanced rule engine, which also makes up its only core component. The interpreter deliberation cycle is largely different from the well-known recognize-select-act execution cycle employed in standard rule systems. The reason for this significant change is mainly caused by the introduction of operator-based problem space search. Hence, the interpreter uses rules for the realization of problem space exploration via rule application.

Conclusion

In this category proven rule-based programming concepts have been made usable on top of custom agent interpreters. These agent interpreters extend rule-kernels in different directions and especially try to adapt the deliberation cycle according to the sense-think-act loop of situated agents. This means that the extended interpreters take into account that an agent regularly senses its environment and exert actions on it.

In the following a short overview about the pros and cons of the B1 category is given:
+ Rules are a well-known programming concept and exploiting only rules leads to a flat learning curve for people having already a background in this area.
+ Reactive agent behavior can be encoded easily in terms of rules and therefore the rule metaphor fits for specifying simple behavior patterns.
- The representation of proactive behavior is rather unintuitive and difficult using rule descriptions only. One important reason for this is that proactive behavior needs to describe what is to be achieved and in addition how it can be accomplished. If no explicit concept for the representation of the agent's goals are available, they have to be encoded indirectly, which renders the descriptions hard to understand.
- Procedural knowledge is difficult to capture using rule descriptions. Especially, if a plan consists of multiple logical steps (performing an action, sending a message, waiting for the response, performing the next action), this cannot be easily captured in one rule, but needs to be scattered among several rules.

Generalization

The approach of using rules for programming in a system environment that extends the basic rule engine behavior has not been considered to a great extent outside the agent community. Despite this fact, at least one other application area exists, in which a similar approach has been pursued. It is widely recognized that the object-oriented programming paradigm fits well for implementing many application domains. Nevertheless, other approaches such as logic

programming have emerged and can partly complement the weaknesses of purely procedural representations. In (Lodha et al. 2005) an architecture for the combination of rule and object-oriented programming has been proposed. Despite these efforts, the mainstream software development does not focus on extending rule interpreters, but instead puts effort in a better integration of rule engines as one part of a business infrastructure. In the advent of service-oriented approaches e.g. rule services and repositories have been proposed, which have the task to centrally bundle rule functionalities and make them available enterprise-wide.

RULE ENGINES AND MENTALISTIC NOTIONS (C2)

In this category approaches are subsumed that introduce mentalistic notions as programming constructs and rely on a rule engine as execution machinery for the agent interpreter. In contrast to category C1 the agent architecture is built on top of a rule kernel using standard rule programming. This means that the rule engine directly becomes an agent interpreter and no changes at the execution level need to be applied. Nonetheless, from the agent programmer's point of view the difference between both categories is blurred, because in both cases high-level mentalistic programming concepts are offered for application development.

Figure 1 highlights that currently Jadex is the only system which falls into this category. Jadex offers belief-desire-intention (BDI) abstractions for programming agents. It has been designed as general purpose agent architecture, which aims at a wide applicability within various kinds of application domains. Some of the most interesting applications have tackled agile business process management (Burmeister et al. 2006) and dynamic appointment scheduling in hospitals (Paulussen et al. 2006). In the following the details of the underlying Jadex architecture will be presented.

Jadex

Foundation of the Jadex architecture is the BDI model, which has been originally proposed by Bratman (1987) as a framework for explaining rational behavior of human agents. BDI assumes a two-staged practical reasoning process which consists of a goal deliberation and a means-end reasoning phase (Wooldridge, 2000). Goal deliberation aims at deciding what to do by selecting a coherent set of goals which should be pursued and the downstream means-end reasoning phase has the purpose to find means for the realization of a specific goal. The BDI model has been picked-up by Rao and Georgeff (1995) who conceived the PRS (Procedural Reasoning System) agent architecture on basis of the BDI model. This architecture focuses mainly on the means-end reasoning process and was subsequently extended by Pokahr et al. (2005a) supporting also the goal deliberation phase. The resulting architecture is based on a rule-kernel but exhibits only the BDI concepts as programming constructs to the developer.

Architecture components

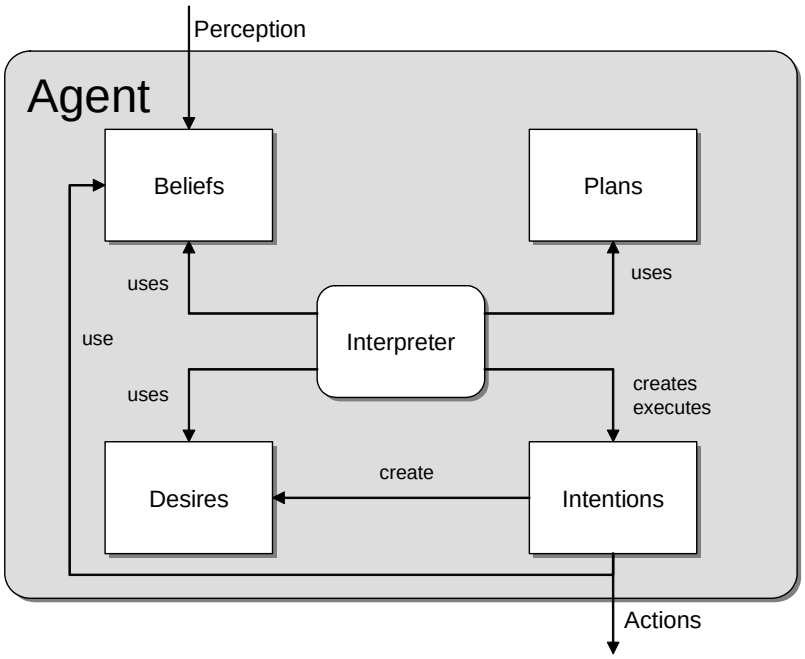

Figure 8: Abstract PRS agent architecture (cf. Georgeff and Lansky 1987)

In figure 8 a high-level view of the Jadex architecture is depicted. It basically shows that the agent interpreter operates on the three attitudes: beliefs, desires and intentions. These basic notions represent, what the agent knows about the world (beliefs), which things its tries to achieve (desires), and the already committed courses of actions it is currently using (intentions).

In order to decide which goals to pursue the interpreter evaluates its current set of desires and selects a coherent subset, which does not contain any conflicting goals. The Jadex interpreter uses the “easy deliberation strategy” (Pokahr et al. 2005b) for performing the selection process, which requires that meta information for goals has been defined.

The means-end reasoning is done for each of the active goals separately. Foundation of this process is that a library of predefined plans is available. A plan hereby represents a procedural piece of knowledge, which encodes how a certain objective can be handled. In contrast to a single action, a plan can be arbitrary complex including sophisticated control structures and may consist of many subsequent steps. In addition to the plan code, also meta information about the plans is stored in the plan library. This allows the means-end reasoning process to search the library for relevant plans, i.e. plan that have declared to be usable for achieving the current goal. From this plan set only those will be kept, whose context condition holds, i.e. which are applicable with respect to the situation the agent is in. If still more than one plan is available, further reasoning for choosing among them can be applied (meta-level reasoning), or a single one can be picked according to a simple heuristics (e.g. random or first one). This plan will then be instantiated and will be added to the active intentions of the agent. During plan execution the plan can access the agent’s beliefs and also can issue subgoals for further task delegation. If a

plan cannot achieve its associated goal, the goal is not automatically considered as failed. Instead the means-end reasoning mechanism will try to accomplish the goal using other applicable plans until no plan is left or the goal could finally be attained.

Deliberation cycle

In contrast to classical PRS agents (Rao und Georgeff 1995) and also many other agent architectures, Jadex does not employ a specific deliberation cycle. Instead, Jadex has a built-in rule program, which controls the interpreters operation. This rule program encodes the complete BDI-logic as explained above and realizes goal-deliberation as well as means-end reasoning.

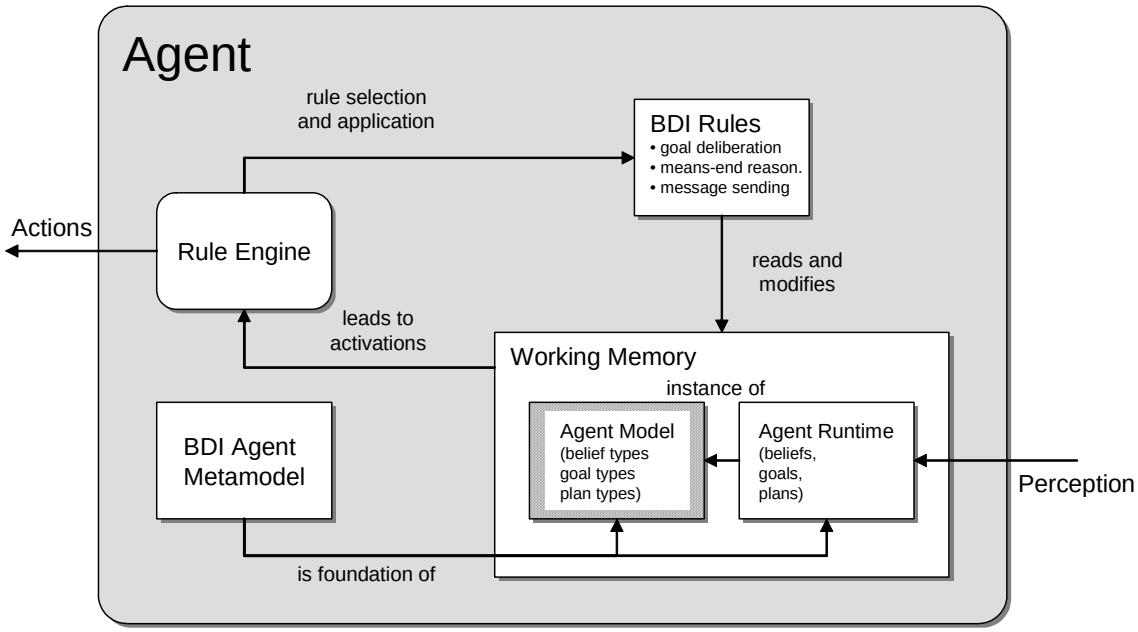

Figure 9: Detailed Jadex agent architecture

In Figure 9 the detailed Jadex agent architecture is presented. It can be inferred that the only active component is the rule engine, which operates on a predefined set of BDI rules. The execution cycle is therefore exactly the same as in a normal rule engine and consists of the recognize-select-act loop that is repeated until no new activations can be found. In contrast to normal rule programs Jadex agents are equipped with a fixed set of BDI rules. These rules are responsible for realizing all agent as well as BDI-specific behavior, i.e. they model e.g. message receiving as well as goal and plan handling. They rely on a predefined agent memory structure, which is encoded in the BDI agent metamodel. This model specifies exactly which kinds of attributes an agent type and an agent instance may have, i.e. the agent type metamodel e.g. constitutes that the agent has specific slots for storing beliefs, plans and goals. At runtime the working memory therefore contains two information sources. The agent model represents the agent class, which is defined by the developer and mainly consists of the definition of the structural elements (beliefs, goals and plan meta-information) as well as the procedural plan code. Similar to object-oriented systems an instance of this class is held in the memory, which e.g. contains runtime information about the currently active goals and plans.

Agent properties

Reactive behavior. A Jadex agent can receive new information via incoming messages as well as through direct senor update via its beliefs. Immediate reactions are possible to both kinds of events. On the one hand plans can be triggered by incoming message events and on the other hand belief changes can be tracked for defining data-driven plans. Within agent plans arbitrary actions can be performed. This includes changes to the agent state, initiating environmental actions as well as sending and waiting for messages.

Proactive behavior. Jadex has been conceived with specific support for proactive behavior. In Jadex, goals are a first-class concept and are explicitly represented within the agent. In this respect it is distinguished between different kinds of goals, which express different motivations towards their accomplishment. Currently, the platform supports perform, achieve, query and maintain goals. The simplest goal kind is the perform goal, which directly leads to action execution and does not contain a declarative part. In contrast achieve goals are associated with a declarative description of a desired world state. Therefore an achieve goal is finished only once this condition is satisfied or it has been determined that it cannot be attained in the future. Query goals can be used to retrieve information. These goals only lead to plan execution, if the requested information is not immediately available from the existing agent knowledge. The most complex goal types are maintain goals, which have the purpose to continuously monitor a specified world state. Whenever the goal detects that the world state is not valid any longer it will get active and lead to plan execution for the re-establishment of the invalid state. Jadex handles all these different goal kinds within the same execution machinery, which is only possible, because a generic goal lifecycle has been defined for all kinds of goals. This goal lifecycle is used for the goal deliberation as well as for the means-end reasoning. It basically partitions the adopted agent goals into active and inactive sets. The goal deliberation can then shift goals between these states as needed and the means-end reasoning processes the goals of the active set as needed. Details about the goal representation and execution can be found in (Braubach et al. 2005).

Communication. The Jadex architecture includes message-based communication facilities. Even though the architecture does not prescribe a specific message format, the FIPA standard (http://www.fipa.org/) has been adopted for representing messages. As in Jadex the agent architecture is decoupled from the platform architecture, messages can be sent using any transport layer the platform offers. The standard implementation uses simple TCP/IP based communication channels.

Mentalistic notions. Mentalistic notions play a crucial role in Jadex. The programming model is built on the intentional stance and considers mentalistic notions as an adequate vehicle for constructing agents. As all important concepts are represented explicitly, the behavior of agents can be introspected at runtime. This introspection reveals, which goals and plans are currently pursued and also which belief values are stored. This allows programmers to easily grasp the agent's state and understand immediately the motivations for the externally perceived agent actions. In addition, further explanation mechanisms have been developed (Lam and Barber

2005) and also made available in Jadex, which visualize mentalistic event traces and interconnections.

Rule-based mechanisms

Rule types. The Jadex agent interpreter uses a forward-chaining rule engine and employs production rules for rule specification. Within Jadex two different kinds of rules can be distinguished. The system relies on a set of fixed BDI rules for the realization of the BDI base functionality. On the other hand, conditions are part of many higher-level concepts, such as an achievement condition of a goal. This means that a programmer can define the condition part of a goal, which is internally handled as rule. The action part of such an implicit rule is determined by its context and cannot be altered by the developer.

State representation. The state representation of Jadex has two facets. Internally, the working memory is organized in an object-attribute-value style similar to other rule engines. The state is strongly typed and relies on a type system for verifying that objects are correctly accessed. This also means that types have to be defined in beforehand of their usage. Additionally, the state provides automatic garbage collection facilities that simplify the memory management, as unreachable state objects are cleaned up automatically. For the agent programmer an object-oriented perspective is offered as access point to the working memory. Hence, externally the state is perceived as a structure of Java objects and can be accessed and modified by using normal method calls.

Rule evaluation mechanism. In Jadex, the RETE algorithm is used for rule evaluation (Forgy 1982). The original mechanisms has been adapted to the strong type system and is similar to the functionality of JESS (Friedman-Hill 2003).

Conflict resolution. In general, the importance of rule ordering is kept low in Jadex, i.e. the BDI rules are formulated in a way that tries to avoid ambiguities and the overall functionality of the agent should not depend on the rule order. Conflict resolution has to be applied in Jadex when multiple activations of BDI rules are present in the agenda. In this case a simple FIFO strategy is used for fetching the next activation to execute.

Truth maintenance. Truth maintenance is currently not used in Jadex.

Rule incorporation

Jadex offers a hybrid programming language approach and separates structural from procedural specifications, i.e. an agent is programmed using an XML dialect for defining the agent type and ordinary Java for the plan bodies. Within the structural part conditions are currently specified in a CLIPS-like language (Giarratano and Riley 2005). Probably this will change in the future and a more Java-like syntax will be adopted for rules, too (possibly similar to Drools, http://www.jboss.org/drools/). As the plans are written in Java, an integration with external libraries and systems is possible using the standard Java mechanisms. Hence, in Jadex agent programming is done using intuitive mentalistic notions and rules appear only as conditions within predefined higher-level constructs such as goals.

The Jadex agent interpreter is a pure rule engine, which operates on a fixed set of BDI rules. These rules encode the agent and BDI knowledge especially for realizing the goal deliberation and means-end reasoning mechanisms.

Conclusion

This category has highlighted that rule engines can be used as backbone for realizing higher-level architectures with more abstract concepts than rules itself. The extensions performed in this category hide the internal rule execution mechanism to a great extent and instead present other conceptual notions for the programming of agents. The idea of this category is therefore twofold: using established rule engines for the efficient agent execution and provide intuitive mentalistic notions to the agent developer.

A short list of the main advantages and disadvantages summarizes the main properties of Category C2:
+ Mentalistic notions for programming agents realize the intentional stance and therefore exhibit advantages with respect to the specification and understanding of its behavior. Moreover, it can help in debugging or improving the agent.
+ Using rule engines for agent implementation has the big advantage that algorithms such as RETE, TREAT or LEAPS exist, which ensure an efficient and scalable execution.
- Rules may not be applicable for implementing all kind of higher-level system behavior. This means that an agent interpreter should introduce further extension (cf. category C1) if a pure rule implementation is not adequate for the problem at hand.
- The usage of higher-level architectures for programming does not pay off for all applications. For specific problems it will be simpler and more convenient to use a rule-based solution, e.g. if an agent is only required to exhibit reactive abilities.

Generalization

The proposed approach of using rule engines as execution kernel for higher-level has been used outside the agent research area as well. One reason for this is that rule engines have a long track of research and are proven reliable technology today. In the area of workflow management systems, questions of how to describe and implement a process have to be tackled. For the design of processes notations such as EPCs (event process chains) and BPMN (business process modeling notation) have emerged. The execution of workflows is in most cases performed within specialized workflow engines, which can handle workflow specifications. As EPCs do not posses an executable semantics per se, in Knolmayer et al. (2000) an approach for a transformation of EPC descriptions into rules has been proposed. For each of the different EPC connector types rules patterns have been conceived so that an automatic translation can be achieved. The integration of rule and workflow concepts has also been recently discussed in the context of the Drools rule engine and the JBoss workflow management system. The idea here is also to use the Drools engine as core of a workflow system.

FUTURE TRENDS

Rule bases have been used for building intelligent agents at least since the eighties of the last century. This chapter has shown that rules are nowadays an important part of many prevalent agent architectures. The continued integration of agent concepts and rule-based technology opens up many interesting areas for future developments. Some opportunities for future research are illustrated in the following.

From the multi-agent perspective distributed systems are one important application area. Despite this fact, in many commercial projects not multi-agent but service-oriented architectures are employed. This has advantages with respect to the proven and standards-driven technology base but also leads to problems, when the passive service character is insufficient and e.g. dynamic and proactive behavior is needed. In order to cope with this dilemma the event-driven architectures (EDA) have been proposed as an extension for SOA. Core concept of EDA is that the environment is observed via a network of agents, which can detect business-relevant situations using complex event processing (CEP). For building EDA-enabled infrastructures a combination of all three aforementioned technology strands is necessary.

Another upcoming trend is the enrichment of the Internet with semantics and its transformation to the SemanticWeb (Antoniou and van Harmelen 2004). This shall enable not only humans but also machines to understand web content and use it for fulfilling tasks. In a typical application case a human user could just delegate a task to a dedicated agent and let the agent perform this task in the background. The agent then possibly would gather information from the web and interact with other agents. For exploiting the semantics and deducing information from the given ontology knowledge, often a rule-based machinery is used. Hence, the bringing together of rule-based knowledge representation and reasoning with agent technology could be considered as enabling research area for the semantic web.

CONCLUSION

In this chapter agent architectures have been described as one application area for rule bases. It has been shown in what different ways agent architectures make use of rule concepts and in what directions they extend existing rule-based approaches.

To identify the differences and commonalities of the approaches, a categorization has been introduced. This categorization is made up of two dimensions: The first dimension considers the execution and distinguishes between pure rule engines and approaches that introduce additional execution logic in an extended or controlled rule engine. The second dimension takes a programmers viewpoint and differentiates approaches that introduce higher level constructs such as beliefs or goals from those that do not.

Each of the resulting four categories has been investigated and a prominent representative of each category has been presented in detail. Besides a general description of the features of a representative, the focus has been put on the incorporation of rule mechanisms into the overall

architecture. Discussions of the advantages and disadvantages of the approaches reveal that all approaches legitimately coexist, as all have their strength and weaknesses with respect to certain application requirements.

KEY TERMS AND THEIR DEFINITIONS

(Software) Agent. Although a great variety of definitions of the term ‘software agent’ exists (Franklin and Graesser 1997), a widely accepted definition is given by Jennings and Wooldridge (1998, p. 4): “an agent is a computer system situated in some environment, and that is capable of *autonomous action* in this environment in order to meet its design objectives”.

Agent Architecture. Following the definition of ‘software architecture’ from Bass et al. (2005) as “the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them”, we regard an ‘agent architecture’ as the control structures that facilitate the specification and execution of agent behavior.

BDI Model. The Belief-Desire-Intention (BDI) model of human practical reasoning, developed by the philosopher Michael Bratman (1987), is a model for assessing the rationality of human actions. Unlike earlier models, which only consider desires and beliefs, the BDI model introduces future-directed intentions, which are composed to plans, as an important and irreducible concept. In the agent research community, the philosophical model has been slightly adapted to specify the behavior of software agents in terms of beliefs, goals and plans.

Deliberation Cycle. The term ‘deliberation cycle’ refers to a conceptual “main loop” of an agent interpreter to illustrate the basic mode of operation (e.g. “sense-reason-act”).

Intentional Stance. The term ‘intentional stance’ was coined by philosopher Daniel C. Dennett (1971) and refers to a viewpoint in which we use (human) mental properties for explaining the behavior also of animals or inanimate things. McCarthy (1979) has argued that the intentional stance is also useful for developing software systems.

Mentalistic Notions / Mental Attitudes. The terms mentalistic notions resp. mental attitudes refer to human properties like beliefs and goals when used for describing software agents.

Multi-agent System. (Wooldridge 2001, p. 3) defines: “A multi-agent system is on that consists of a number of agents, which interact with one another […].”

Physical Symbol System Hypothesis. The Physical Symbol System Hypothesis has been formulated by Newell and Simon (1976) and states that: "A physical symbol system has the necessary and sufficient means of general intelligent action."

UTC. (Newell 1990) introduced the term Unified Theories of Cognition (UTC) as an objective for the psychology research field. A Unified Theories of Cognition should overcome the existing

variety of psychological theories (over 3000 in 1990) and offer a unified explanation framework for human cognition.

FIGURE CAPTIONS

Figure 1: Agent Architecture Classification
Figure 2: Rule Responder agent architecture
Figure 3: Rule Responder Middleware
Figure 4: AOP language family
Figure 5: Example of a library agent
Figure 6: AOP agent architecture
Figure 7: SOAR agent architecture
Figure 8: Abstract PRS agent architecture (cf. Georgeff and Lansky 1987)
Figure 9: Detailed Jadex agent architecture

REFERENCES

Antoniou, G, van Harmelen, F. (2004). *A Semantic Web Primer*. MIT Press.

Bass, L., Clements, P., Kazman, R. (2005). *Software architecture in practice*. Addison-Wesley.

Bellifemine, F., Bergenti, F., Caire, G., Poggi, A. (2005). JADE - A Java Agent Development Framework. In R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (Eds.), *Multi-Agent Programming: Languages, Platforms and Applications* (pp. 125–147), Springer.

Bordini, R., Hübner, J. F., Vieira, R. (2005). Jason and the Golden Fleece of Agent-Oriented Programming. In R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (Eds.), *Multi-Agent Programming: Languages, Platforms and Applications* (pp. 3–37), Springer.

Bratman, M. (1987). *Intention, Plans, and Practical Reason*. Harvard University Press.

Bratman, M., Israel, D., Pollack, M. (1988). Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4(4), 349–355.

Braubach, L., Pokahr, A., Lamersdorf, W., Moldt, D. (2005). Goal Representation for BDI Agent Systems. In R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (Eds.), *Proceedings of the 2nd International Workshop on Programming Multiagent Systems, Languages and Tools (PROMAS 2004)*, 3rd International Joint Conference on Autonomous Agents & Multi-Agent Systems (AAMAS'04) (pp. 46-67), New York, USA, Springer.

Braubach, L., Pokahr, A., Lamersdorf, W. (2006). Tools and Standards. In S. Kirn, O. Herzog, P. Lockemann, O. Spaniol (Eds.), *Multiagent Engineering - Theory and Applications in Enterprises*, (pp. 503-530). Springer.

Burmeister, B., Steiert, H.-P., Bauer; T., Baumgärtel H. (2006). Agile Processes Through Goal- and Context-Oriented Business Process Modeling. *Proceedings of Business Process Management Workshops*, Vol. 4103, (pp. 217-228), Springer.

Brownston, L ., Farrell, R., Kant, E., Martin, N. (1985). *Programming Expert Systems in OPS5*, Addison-Wesley.

Brooks, R. A. (1989). How To Build Complete Creatures Rather Than Isolated Cognitive Simulators. In K. VanLehn (Ed.), *Architectures for Intelligence*, pp. 225–239.

Byrne, C., Edwards, P. (1996). Refinement in Agent Groups. In G. Weiß (Ed.): *Proceedings of the IJCAI'95 Workshop on Adaption and Learning in Multi-Agent Systems* (pp. 22–39), Springer.

Cardoso, H. L. (2007). *Integrating JADE and Jess*. Retrieved from http://jade.tilab.com/doc/tutorials/jade-jess/jade_jess.html

Dastani, M., Hobo, D., Meyer, J.-J. (2007). Practical Extensions in Agent Programming Languages. *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07)* (pp. 923-925), ACM Press.

Davies, W., Edwards, P. (1994). Agent-K: An Integration of AOP and KQML. *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*, ACM Press.

Dennett, D. (1971). Intentional Systems. *Journal of Philosophy*, 68, 87–106.

Dietrich, J., Kozlenkov, A., Schroeder, M., Wagner, G. (2003). Rule-based agents for the semantic web. *Electronic Commerce Research Applications*, 2(4), 323-338.

Doorenbos, R. (1995). *Production matching for large learning systems*. PhD Thesis, Carnegie Mellon University, UMI Order No. GAX95-22942.

Finin, T., Weber, J., Wiederhold, G., Genesereth, M., McKay, D., Fritzson, R. et al. (1993) Specification of the KQML Agent-Communication Language – plus example agent policies and architectures. (EIT TR 92-04).

Fiege, L., Mühl, G., Pietzuch, P. (2006). *Distributed Event-Based Systems*. Springer, 2006.

Forgy, C. (1982) Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artificial Intelligence* 19(1), 17–37.

Franklin, S., Graesser, A. (1997). Is it an Agent, or Just a Program? A Taxonomy for Autonomous Agents. In J. Müller, M. Wooldridge, N. Jennings (Eds.) *Proceedings of the 3rd Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages (ATAL 1996)* (pp. 21–35), Springer.

Friedman-Hill, E. (2003). *Jess in Action, Java Rule-based Systems*. Manning Publishers, 2003.

Georgeff, M., Lansky, A. (1987). Reactive Reasoning and Planning: An Experiment With a Mobile Robot. *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI 1987)* (pp. 677–682).

Giarratano, J. C., Riley, G. D. (2005). *Expert Systems. Principles and Programming*. Thomson Corse Technology.

Hindriks, K., Boer, F. de, Hoek, W. van d., Meyer, J.-J. (1999). Agent Programming in 3APL. Autonomous Agents and Multi-Agent Systems 2(4), 357–401.

Jennings, N. R., Wooldridge, M. J. (1998). *Agent Technology - Foundations, Applications and Markets*. Springer.

Knolmayer, G., Endl, R., Pfahrer, M. (2000). Modeling Processes and Workflows by Business Rules. In W. M. Aalst, J. Desel und A. Oberweis (Eds.), Business *Process Management, Models, Techniques, and Empirical Studies* (pp. 16-29). Springer.

Krempels, K.H., Panchenko, A. (2003). An Approach for Automated Surgery Scheduling. In Edmund K Burke and Hana Rudov (Eds.), *Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling (PATAT '06).*

Laird, J., Congdon, C.B., Coulter, K. (2006). The Soar User's Manual. Version 8.6.3. University of Michigan. http://sitemaker.umich.edu/soar

Lam, D. N., Barber, K. S. (2005). Comprehending agent software. In: F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, M. Wooldridge (Eds.): *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)* (pp. 586-593), ACM Press.

Lehman, J. F., Laird, J., Rosenbloom, P. (1996) A gentle introduction to Soar, an architecture for human cognition. In S. Sternberg, D. Scarborough (Eds.), *Invitation to Cognitive Science*, 4, 212–249, MIT Press.

Lodha, B., Dinesha, K. V., Kumar, P. (2005). Need for Incorporating a Rule-Based Component in Conventional Object Oriented Systems. *In Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*.

Luck, M., McBurney, P., Shehory, O., Willmott, S. (2005). Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing). AgentLink.

Luckham, D. (2002). *The Power of Events*. Addison Wesley.

McCarthy, J. (1979). Ascribing mental qualities to machines. In M. Ringle (Ed.): *Philosophical Perspectives in Artificial Intelligence* (pp. 161–195). Humanities Press.

Miranker, D. (1989). TREAT: A new and efficient match algorithm for AI production systems. *Proceedings of National Conference on Artificial Intelligence* (pp. 42-47), Pittman/Morgan Kaufman.

Newell, A. (1990). *Unified Theories of Cognition*. Harvard University Press.

Newell, A., Simon, H. A. (1976). Computer Science as Empirical Enquiry. *Communications of the ACM,* 19, 113-126.

Nwana, H. (1995) Software Agents: An Overview. *Knowledge Engineering Review,* 11(2), 205–244.

Paulussen, T.O., Zöller, A., Heinzl, A., Braubach, L., Pokahr, A, Lamersdorf, W. (2006). Agent-Based Patient Scheduling in Hospitals. In S. Kirn, O. Herzog, P. Lockemann, O. Spaniol (Eds.), *Multiagent Engineering - Theory and Applications in Enterprises* (pp. 255-275). Springer.

Paschke, A., Bichler, M. (2005). SLA Representation, Management and Enforcement. *Proceedings IEEE International Conference on e-Technology, e-Commerce, e-Service (EEE'05)*.

Paschke, A., Kiss, C., Al-Hunaty, S. (2006). NPL: Negotiation Pattern Language - A Design Pattern Language for Decentralized (Agent) Coordination and Negotiation Protocols, In R. Banda (Ed.). *E-Negotiation - An Introduction*. ICFAI University Press.

Paschke, A. (2007). Rule-Based Service Level Agreements - Knowledge Representation for Automated e-Contract, SLA and Policy Management, IDEA Publishing.

Paschke, A., Boley, H., Kozlenkov, A., Craig, B. (2007). Rule Responder: RuleML-based agents for distributed collaboration on the pragmatic web. *Proceedings of the 2nd international Conference on Pragmatic Web* (pp. 17-28). ACM Press.

Paschke, A., Kozlenkov, A., Boley, H. (2007a) A Homogenous Reaction Rules Language for Complex Event Processing. *Processings of International Workshop on Event Drive Architecture for Complex Event Process (EDA-PS 2007)*.

Pokahr, A., Braubach, L., Lamersdorf, W. (2005a) A BDI Architecture for Goal Deliberation. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. Singh, M. Wooldridge (Eds.), *Proceedings of 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)* (pp. 1295–1296), ACM.

Pokahr, A., Braubach, L., Lamersdorf, W. (2005b). A Goal Deliberation Strategy for BDI Agent Systems. In T. Eymann, F. Klügl, F, W. Lamersdorf, M. Klusch, M. Huhns (Eds.): *Proceedings of the Third German Conference on Multi-Agent System TEchnologieS (MATES-2005)* (pp. 82-94), Springer.

Rao, A., Georgeff, M. (1995). BDI Agents: From Theory to Practice. In V. Lesser, L. Gasser (Eds.), *Proceedings of the First International Conference on Multiagent Systems* (pp. 312-319), MIT Press.

Ross, R., Collier, R., O'Hare, G. (2005). AF-APL - Bridging Principles and Practice in Agent Oriented Languages. In R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni, (Eds.), *Proceedings of Second International Workshop on Programming Multi-Agent Systems (ProMAS 2004)* (pp. 66-88). Springer.

Schroeder, M., Wagner, G. (1998) Vivid agents: theory, architecture, and applications. *Applied Artificial Intelligence* 14, 645-675.

Shoham, Y. (1993) Agent-oriented programming. *Artificial Intelligence* 60(1), 51–92.

Thomas, S. R. (1995). The PLACA Agent Programming Language. In M. Wooldridge, N. Jennings (Eds.), *Proceedings of the 1st International Workshop Intelligent Agents I, Agent Theories, Architectures and Languages (ATAL 1994)* (pp. 355–370) Springer.

Winikoff, M. (2005). JACK Intelligent Agents: An Industrial Strength Platform. In R. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (Eds.), *Multi-Agent Programming: Languages, Platforms and Applications* (pp. 175-193). Springer.

Wooldridge, M., Jennings, N. (1995) Agent Theories, Architectures, and Languages: A Survey. In M. Wooldridge, N. Jennings (Eds.), *Proceedings of International Workshop on Intelligent Agents I, Agent Theories, Architectures, and Languages (ATAL'94)* (pp. 1-39), Springer.

Wooldridge, M. (2000). *Reasoning about Rational Agents*. MIT Press.

Wooldridge, M. (2001). *An Introduction to MultiAgent Systems*. John Wiley & Sons.

Wray, R., Jones, R. (2006) Considering Soar as an agent architecture. In R. Sun (Ed.), *Cognition and Multi-Agent Interaction - From Cognitive Modeling to Social Simulation*, Cambridge University Press.

Wright, I., Marshall, J. (2003). The Execution Kernel of RC++: RETE*, A Faster RETE With Treat as a Special Case. *International Journal of Intelligent Games and Simulation*, 2(1), 36–48.