Semantic Web Technology as a Basis for Planning and Scheduling Systems

Bernd Schattenberg and **Steffen Balzer** and **Susanne Biundo**
Dept. of Artificial Intelligence
University of Ulm, D-89069 Ulm, Germany
{firstname}.{lastname}@uni-ulm.de

Abstract

This paper presents an architecture for planning and scheduling systems that addresses key requirements of real-world applications in a unique manner. The system provides a robust, scalable and flexible framework for planning and scheduling software through the use of industrial-strength middleware and multi-agent technology. The architectural concepts extend knowledge-based components that dynamically perform and verify the system's configuration. The use of standardized components and communication protocols allows a seamless integration with third-party libraries and existing application environments.

The system is based on a proper formal account of hybrid planning, the integration of HTN and POCL planning. The theoretical framework allows to decouple flaw detection, modification computation, and search control. In adopting this methodology, planning and scheduling capabilities can be easily combined by orchestrating respective elementary modules and strategies. The conceptual platform can be used to implement and evaluate various configurations of planning methods and strategies, without jeopardizing system consistency through interfering module activity.

Introduction

Hybrid planning – the combination of hierarchical task network (HTN) planning with partial order causal link (POCL) techniques – turned out to be most appropriate for complex real-world planning applications (Estlin, Chien, & Wang 1997), like crisis management support (Biundo & Schattenberg 2001; Castillo, Fdez-Olivares, & González 2001). Here, the solution of planning problems often requires the integration of planning from first principles with the utilization of predefined plans to perform certain complex tasks.

Previous work (Schattenberg, Weigl, & Biundo 2005) introduced a formal framework for hybrid planning, in which the plan generation process is functionally decomposed into well-defined flaw detecting and plan modification generating functions. As an important feature of this approach, an explicit triggering function defines, which modifications are suitable candidates for solving which flaws. This allows to completely separate the computation of flaws from the computation of possible plan modifications, and in turn both computations can be separated from search related issues. The system architecture relies on this separation and exploits it in two ways: module invocation and interplay are specified through the triggering function while the explicit reasoning about search can be performed on the basis of flaws and modifications without taking their actual computation into account. This explicit representation of the planning strategy allows for the formal definition of a variety of strategies, and even led to the development of novel so-called *flexible* strategies.

The functional decomposition induces a modular and flexible system design, in which arbitrary system configurations –viz. planning and scheduling functionalities– can be integrated seamlessly. A prototype of this architecture served as an experimental environment for the evaluation of flexible strategies as well as a conceptual proof for the expandability of the system with respect to new techniques: namely, the integration of scheduling (Schattenberg & Biundo 2002; 2006) and probabilistic reasoning (Biundo, Holzer, & Schattenberg 2004; 2005).

While (Schattenberg, Weigl, & Biundo 2005) presented the theoretical framework for a straight-forward system design for hybrid plan generation, a number of functional and non-functional requirements are obviously not met by such an architectural nucleus when it comes closer to real-world application scenarios like crisis management support, assistance in telemedicine, personal assistance in ubiquitous computing environments, etc. Like any other mission critical software in these contexts, planning and scheduling systems should feature characteristics which call for highly sophisticated software support:

1. declarative, automated system configuration and verification – for fast, flexible, and safe system deployment and maintenance, and for an easy application-specific configuration tailoring
2. scalability, including transparency with respect to system distribution, access mechanisms, concurrency, etc. – for providing computational power on demand without additionally burdening system developers
3. standards compliance – for integrating third-party systems and libraries, and for interfacing with other services and software environments

Each of these characteristics represents a challenge in its own for any software environment, and this is in particular the case for planning and scheduling applications. This paper describes a novel planning and scheduling system architecture which essentially addresses all of the above challenges, and shows how the formal framework of (Schattenberg, Weigl, & Biundo 2005) has been incorporated. It shows not only how modern software technology –in particular middleware and knowledge-based systems– can be successfully applied to a prototypical academic planning software, but also illustrates how (in principle) any planning and scheduling system can benefit from it. The resulting system performs a dynamical configuration of its components and even reasoning about the consistency of that configuration is possible. The planning components are transparently deployed, distributed (including an optimized concurrency), and load-balanced while retaining a relatively simple programming model for the component developer. Standardized protocols and components finally provide easy access to other software products and services.

The rest of this document is organized as follows: The next section presents the formal framework of hybrid planning on which our approach is based. Then a reference planning process model is defined, as an overview for the architecture. This is followed by a description of the architecture components, how the middleware is used and how a refined planning process model is realized. After that, there is a section devoted to the use of knowledge representation mechanisms in the system. The paper concludes with an overview over related work and some final remarks.

Formal Framework

Our planning system relies on a formal specification of hybrid planning (Schattenberg, Weigl, & Biundo 2005): The approach features a STRIPS-like representation of action schemata with PL1 literal lists for preconditions and effects and state transformation semantics based on respective atom sets. It discriminates primitive operators and abstract actions (also called complex tasks), the latter representing abstractions of partial plans. The plan data structure, in HTN planning referred to as *task network*, consists of complex or primitive task schema instances, ordering constraints and variable (in-)equations, and causal links for representing the causal structure of the plan. For each complex task schema, at least one *method* provides a task network for implementing the abstract action.

Planning problems are given by an initial task network, i.e. an abstract plan, a set of primitive and complex task schemata, and a set of methods specifying possible implementations of the complex tasks. A partial plan is a solution to a given problem, if it contains primitive operators only, the ordering and variable constraints are consistent, and the causal links support all operator preconditions without being threatened.

Flaws

The violation of solution criteria is made explicit by so-called *flaws* – data structures which literally "point" to deficiencies in the plan and allow for the problems' classification: A flaw $\texttt{f}$ is a pair $(flaw, E)$ with $flaw$ indicating the flaw class and E being a set of plan components the flaw refers to. The set of flaws is denoted by $\mathcal{F}$ with subsets $\mathcal{F}_{flaw}$ for given labels $flaw$.

E.g., the flaw representing a threat between a plan step te_k and a causal link $\langle te_i, \phi, te_j \rangle$, is defined as: $(\texttt{Threat}, \{\langle te_i, \phi, te_j\rangle, te_k\})$. In the context of hybrid planning, flaw classes also cover the presence of abstract actions in the plan, ordering and variable constraint inconsistencies, unsupported preconditions of actions, etc.

The generation of flaws is encapsulated by detection modules, i.e. functions that take as an argument a plan and return a set of flaws. Without loss of generality we may assume, that there is exactly one such function for each flaw class. The function for the detection of causal threats, e.g., is defined as follows:

$f^{det}_{CausalThreat}(P) \ni (\texttt{Threat}, \{\langle te_i, \phi, te_j\rangle, te_k\})$ iff: $te_k \not\prec^* te_i$ or $te_j \not\prec^* te_k$ in the transitive closure $\prec^*$ of P's ordering relation and the variable (in-) equations of P allow for a substitution σ such that $\sigma(\phi) \in \sigma(\text{del}(te_k))$ for positive literals ϕ and $\sigma(|\phi|) \in \sigma(\text{add}(te_k))$ for negative literals ϕ.

Modifications

The refinement steps for obtaining a solution out of a problem specification (which means to get rid of any flaws) are explicit representations of changes to the plan structure. A plan modification $\texttt{m}$ is a pair $(mod, E^{\oplus} \cup E_{\ominus})$ with mod denoting the modification class. $E^{\oplus}$ and $E_{\ominus}$ are elementary additions and deletions of plan components, respectively. The set of all plan modifications is denoted by $\mathcal{M}$ and grouped into subsets $\mathcal{M}_{mod}$ for given classes mod.

The following structure represents adding an ordering constraint between to plan steps te_i and te_j: $(\texttt{AddOrdConstr}, \{\oplus(te_i \prec te_j)\})$. Further examples of hybrid planning modifications are the insertion of new action schema instances, variable (in-) equations, and causal links, and of course the expansion of complex tasks according to appropriate methods.

As with the flaws, the generation of plan modifications is encapsulated by modification modules. These functions take a plan and a set of flaws as arguments and compute all possible plan refinements that solve flaws. E.g., promotion and demotion as an answer to a causal threat is defined as:

$$f^{mod}_{\texttt{AddOrdConstr}}(P, \{\texttt{f}, \ldots\}) \supseteq \{\, (\texttt{AddOrdConstr}, \{\oplus(te_k \prec te_i)\}), \\ (\texttt{AddOrdConstr}, \{\oplus(te_j \prec te_k)\})\}$$

for $\texttt{f} = (\texttt{Threat}, \{\langle te_i, \phi, te_j\rangle, te_k\})$.

Refinement-based Planning

It is obvious that some classes of modifications address particular classes of flaws while others do not. This relationship is explicitly represented by the so-called *modification triggering function* α which relates flaw classes with suitable modification classes (cf. (Schattenberg, Weigl, & Biundo 2005)). As an example, causal threat flaws can in principle be solved by expanding abstract actions which are involved in the threat, by promotion or demotion, or by separating variables through in-equality constraints (cf. (Biundo & Schattenberg 2001)):

$$\alpha(\mathcal{F}_{\texttt{Threat}}) = \mathcal{M}_{\texttt{ExpandTask}} \cup \mathcal{M}_{\texttt{AddOrdConstr}} \cup \\ \mathcal{M}_{\texttt{AddVarConstr}}$$

Please note, that the triggering function states nothing about the relationship of the actual flaw and modification instances.

Apart from serving as an instruction, which modification generators to consign with which flaw, the definition of the triggering function gives us a general criterion for discarding un-refineable plans: For any detection and modification modules associated by a trigger function α, f_x^{det} and $f_{y_1}^{mod}, \ldots, f_{y_n}^{mod}$ with $\mathcal{M}_{y_1} \cup \ldots \cup \mathcal{M}_{y_n} = \alpha(\mathcal{F}_x)$: if $\bigcup_{1 \le i \le n} f_{y_i}^{mod}(P, f_x^{det}(P)) = \emptyset$ then P cannot be refined into a solution.

A generic algorithm can then be defined which uses these modules (see Alg. 1): In a first phase, the results of all detection module implementations are collected. In a second phase, the resulting flaws are propagated according to the triggering function[1] α to the respective modification module implementations. If any flaw remains un-answered, a failure is indicated. A strategy module selects in a third phase the most promising modification, which is then applied to the plan. The algorithm is then called recursively with that modified plan. The strategy also serves as a backtracking point of the procedure.

Algorithm 1 A generic planning algorithm, based on explicit flaw and modification computation

```
plan($P$, T, M):
$F \leftarrow \emptyset$
for all $f_x^{det}$ do
    $F \leftarrow F \cup f_x^{det}(P)$
if $F = \emptyset$ then
    return $P$
$M \leftarrow \emptyset$
for all $F_x = F \cap \mathcal{F}_x$ with $F_x \neq \emptyset$ do
    answered $\leftarrow$ false
    for all $f_y^{mod}$ with $\mathcal{M}_y \subseteq \alpha(\mathcal{F}_x)$ do
        $M' \leftarrow f_y^{mod}(P, F_x)$
        if $M' \neq \emptyset$ then
            $M \leftarrow M \cup M'$
            answered $\leftarrow$ true
    if answered $=$ false then
        return fail
return plan(apply($P$, $f_z^{strat}(P, F, M)$), T, M)
```

(Schattenberg, Weigl, & Biundo 2005) demonstrated how planning strategies are formally defined in that framework and illustrated its potential. Several adaptations of strategies taken from the literature were presented, as well as a set of novel *flexible* planning strategies. The latter exploit the explicit flaw and modification information, which allows for selection schemata that are not defined along flaw or modification type preferences, but perform an opportunistic way of plan generation. In a first series of experiments, a set of flexible and fixed, classical strategies competed on a former planning competition benchmark for HTN systems, the UMTranslog domain as it has been shipped with the UMCP system. It turned out, that flexible strategies are not only competitive to their fixed ancestors, but also showed high optimization potential and –due to their opportunistic modus operandi– can easily be combined.

[1]This makes the algorithm completely independent from the actually deployed module implementations.

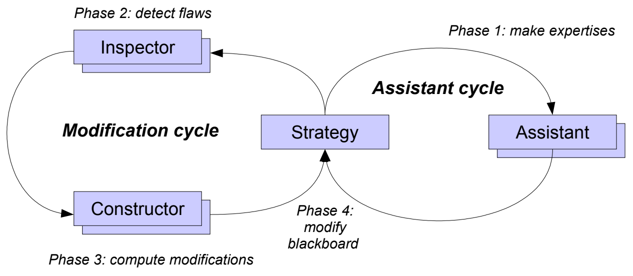

Figure 1: The reference planning process model for PANDA

Architecture Overview

In following the proposed design of the last sections, the basic architecture of PANDA (*Planning and Acting in a Network Decomposition Architecture*) is that of a multiagent-based blackboard system. The agent societies map directly on the presented module structure, with the agent metaphor providing maximal flexibility for the implementation.

Inspectors are implementations of flaw detection modules. There is one such agent per flaw class.

Constructors are agent incarnations of the plan modification generating modules. We may assume, that each modification class is represented by one such agent.

Assistants provide shared inference and services which are required by other agents. Assistant agents propagate implications of temporal action information transparently into the ordering constraints, simplify variable constraints, etc.

Coordinators implement the planning strategy module by synchronizing the execution of the other agents and performing the modification selection. Currently only one coordinator is allowed in the system – the so called *strategy*.

Figure 1 shows the reference planning model for PANDA, which defines the agent interaction. A planning *cycle* corresponds to an iteration of a monolithic algorithm (cf. Alg. 1). It consists of two sub-cycles that are divided into 4 phases (see Fig. 1), in which the agents execute concurrently.

Phase 1: Assistants repeatedly derive additional information and post it on the blackboard. This phase ends when no member of the assistant community added information anymore.

Phase 2: Inspectors analyze the current plan residing on the blackboard and post the results, i.e. the detected and prioritized flaws, to the strategy and to the constructors assigned to them.

Phase 3: Constructors compute all possible modifications for the received flaws and send them along with a priorization to the strategy.

Phase 4: The strategy compares all results received from the inspectors and constructors and selects one of the modifications to be executed on the current plan. A planning cycle is hereby completed and the system continues with phase 1 to execute the next planning cycle.

Phase transitions are performed only by the strategy when all participating agents have finished execution. Thus, the phase transitions can be viewed as synchronization points within the planning process. The strategy modifies the plan until no more flaws are detected or an inspector published a flaw for which no resolving modification is issued. In the first case, the current plan constitutes a solution to the given planning problem, in the latter case the planning process has reached a dead end and the system has to backtrack in order to execute a different modification on the strategy's stack. The blackboard is implemented as a stack that stores the current plan, all derived information, and performed modifications. This structure enables the strategy to backtrack the system to a certain point.

The following sections will show, how the reference planning process model has been implemented, using middleware and knowledge-based technology. The chosen multiagent-system is based on on an industrial-strength middleware and uses an explicit knowledge representation in the implementation of the necessary protocols. A refined version of the reference model will then allow us to exploit agent concurrency more efficiently.

A Knowledge-based Middleware

Core Components

An obvious implementation for a planning system following the reference process model would still run in a sole Java virtual machine, viz. on a single computational resource. This stands in contrast to the requirements that complex and dynamic application domains demand. For crisis management support, e.g., information must be gathered from distributed and even mobile sources, the planning process requires a lot of computational power, etc. So scalability and distribution play key roles in the proposed system architecture, while maintaining the (simple but effective) reference process.

The main aspect in middleware systems like application servers is to hide the mechanisms that enable the distributed handling of objects from the programmer. Thus, it is possible to develop distributed applications much more efficiently. In other words, such middleware systems make distribution issues *transparent* to the programmer. Examples for transparency in middleware systems are location transparency, scalability transparency, access transparency, concurrency transparency etc. (Emmerich 2000). Scalability transparency for example means that it is completely transparent to the programmer how a middleware system scales in response to a growing load. In summary, middleware systems take care of the complexity of handling distributed objects and provide an abstract and easy to use API to the programmer.

In order to benefit from application server technology, the PANDA system builds upon the open-source implementation JBoss (Stark 2003). The most important components that a *Java 2 Enterprise Edition – J2EE* (Sun 1999) based application server delivers w.r.t. this work are the following:

- *Enterprise Java Beans – EJBs* are the objects that are managed by an application server. All transparency aspects apply to them. They are the building blocks of a distributed J2EE application (Sun 2003).
- The *Java Naming and Directory Interface – JNDI* is the directory service that enables location and access transparency. It provides a mapping between Java names and remote interfaces of Java objects. The access to all EJBs and other services of the application server is provided through this interface.
- The *Java Messaging Service – JMS* enables asynchronous and location transparent communication between Java components (especially EJBs) beyond virtual machine boundaries. So this service will be of interest when it comes to communication between the different components of PANDA.

In addition to the features described above, application server implementations also cover aspects like security, database access, transaction management etc. They all belong to the J2EE specification. However, a full discussion of their benefits for the PANDA system is beyond the scope of this paper.

Although the application server technology provides powerful mechanisms, we still need more support for realizing the multiagent system functionalities of the reference model. Instead of investigating proprietary agent life-cycle management and communication mechanisms, we decided to take advantage of the work of the Foundation for Intelligent Physical Agents (FIPA), which has been developing standards for that area since 1996. The second core component which is chosen to implement all agent specific features, i.e. to take care of the agent computing capabilities of the system, is therefore BlueJADE (Cowan & Griss 2002). BlueJADE integrates the well-known multiagent framework JADE (Bellifemine *et al.* 2005) with JBoss. This integration puts the agent system life cycle under full control of the application server, that means all distribution capabilities of the application server apply to the agent societies. Access to the JADE agent API is provided by BlueJADE's ServiceMBean interface. It has been selected as the agent computing platform for PANDA because of the following key features:

It is a FIPA-compliant agent platform and provides a library of ready-to-use agent interaction protocols. This enables the PANDA system to interact with other multi-agent-systems and their services.

It is a distributed system, i.e. its agents can be spread transparently over several agent containers running on different nodes in a network, including the migration of running agents between containers. These features can be exploited for distributed information gathering and (automated) load balancing. It has to be noted, that this kind of distribution management is "on top" of the middleware facilities: agent migration typically anticipates pro-actively the computation or communication load in a relative abstract manner, while middleware migration reacts on such load changes based on very low-level operating system specific sensors. It makes sense to provide both mechanisms in parallel, e.g. to migrate scheduling inspector agents, which are known to require much computational resources onto dedicated compute servers.

The LEAP extension (Caire 2005) adds support for ubiquitous computing. Agents are able to run even on mobile devices such as Java capable cellular phones, PDAs, etc., which are all coveted user-interfaces in many application domains.

BlueJADE supports application defined content languages and ontologies. So a DAML-based content language can be easily integrated (this will be discussed later).

The system comes with a set of sophisticated graphical debugging tools. This speeds up the development process significantly.

The knowledge representation and reasoning facilities which are used throughout the system constitute the third core component. During its development, the PANDA framework required an increasing amount of knowledge that represents planning related concepts (flaw and modification classes, etc.), the system configuration (which inspectors, constructors, and strategies to deploy), and the plan generation process itself (the reference process, including the backtracking procedure, etc.). Most of this knowledge is typically represented implicitly through algorithms and data structures. To make it explicit and modifiable without touching the system's implementation, it must be extracted and represented in a common knowledge base which uses a representation formalism that is expressive enough to capture all modeling aspects on one side and that allows efficient reasoning on the other side. As a result of this, the system can be configured generically and that configuration can be verified on a higher semantic level.

There is a large number of knowledge representation systems available on the market which promise to meet the requirements. But since special regard is spent on standards compliance, the *DARPA Agent Markup Language – DAML* (Horrocks, Harmelen, & Patel-Schneider 2001) has been chosen as the grounding representation formalism for this task. It combines the key features of description logics (Baader *et al.* 2003) with Internet standards such as XML or RDF (Manola & Miller 2003) and – even more important – powerful reasoners and other freely available tools exist to integrate the language into applications. In our case, the knowledge encoded in DAML must be made available to the Java programming language. Therefore, a Java object model is necessary that provides mappings in both directions – from Java to DAML and vice versa. The JENA API (McBride 2000) from Hewlett Packard delivers an in-memory object model of a DAML document along with a rich API to query and manipulate it. By using DAML as the content language for the BlueJADE agent communication and also as the language for describing system configurations and communication means, we achieve a homogeneous representation in the system.

Last, but not least, it is of course necessary to integrate a suitable description logic system to store the knowledge and to reason about it. The RACER system (Haarslev & Möller 2001) has the essential capabilities that are required: a DAML codec, an efficient reasoning component, and a knowledge store based on a client-server architecture.

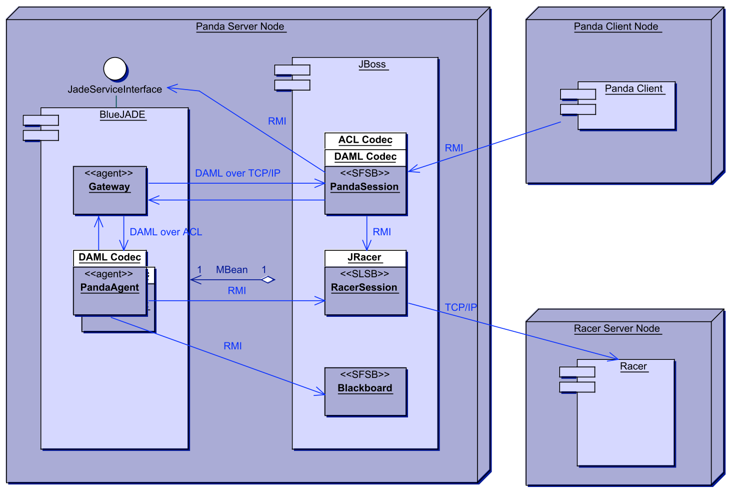

Figure 2: Static system structure of PANDA

The System Structure

Figure 2 shows PANDA's static system structure as a UML deployment diagram. The association arrows indicate which components communicate with each other. Their labels denote the transport protocols being used. BlueJADE is aggregated by JBoss as a (Service-) MBean. It's functionality is exposed via the *JadeServiceInterface*.

The PANDA *Client* component represents the client application that controls the PANDA system. Currently, an RMI-based communication is used. The PANDA client obtains an interface to the PANDA system by querying JBoss's directory service JNDI. But also web-based approaches using SOAP or HTTP are supported. In this way, JBoss provides technologies like Web Services and Java Servlets.

Regarding the integration with the JBoss infrastructure, the PANDA prototype defines three specializations of EJBs for non-agent system components: the interface to the RACER system, to the blackboard, and to the agent society (from outside the system).

Access to the Racer server is provided by the *RacerSessionBean*. The main reason for integrating the Racer system via an EJB proxy is that all components that depend on the Racer system – i.e. EJBs and Agents – are able to access it transparently. They do not need to know its IP address or socket number. Furthermore, the RacerSessionBean can be viewed as generic integration approach for all kinds of reasoner architectures. The communication between Racer and the RacerSessionBean is realized with the *JRacer* client API which translates Java method calls into Lisp function calls. It should be emphasized that each component that obtains a reference to the RacerSessionBean gets its own instance – as usual for SessionBeans. Therefore, queuing of requests is delegated to the Racer server. In a similar fashion, the *BlackboardSessionBean* represents a proxy to the blackboard component.

The *PandaSessionBean* represents the facade by which the PANDA client configures and controls the planning process. It uses the RacerSessionBean to derive the agents and

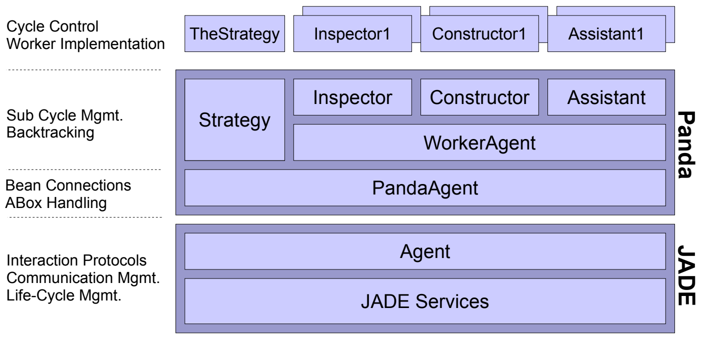

Figure 3: The logical layer structure of the agent framework

their implementations that must be instantiated and creates them using the *JadeServiceInterface*. Communication with the agent framework is done via the *JadeBridge* class of the BlueJADE package, which creates and accesses agent messages (see below) in an object-oriented manner.

Basically, two main *classes of agents* exist in the BlueJADE agent container. The first is the class of standard agents that come with the JADE and BlueJADE software packages. They provide the FIPA infrastructure, several debugging tools and JADE specific communication services. The *Gateway* agent's role is to mediate messages between JADE agents and components outside the JADE agent container. It's counterpart in the EJB container is the JadeBridge. The Gateway sends and receives stringified messages in the *Agent Communication Language* (ACL) via a TCP/IP socket connection.

Custom agents in the PANDA system, i.e. all agent types from the reference model, are all derived from the `PandaAgent` class which encapsulates low level data conversion and communication mechanisms. The PANDA agents form the second class in the JADE agent container. The `PandaAgent` class on its part is derived transitively from the JADE agent base class `Agent` which provides the integration into the JADE agent container (cf. Figure 3).

The reference model (Fig. 1) omits the actual means for *calling* agents, in a distributed implementation, these remote calls are typically message based. From the agent container's point of view, agents in the JADE agent container and the *PandaSessionBean* communicate by using messages encoded in the agent communication language *FIPA-ACL* (FIP 2002b) (in short ACL). ACL is a language based on speech-act theory, i.e. every message describes an action that is intended to be carried out with that message simultaneously (e.g. the request “compute detections”). Such intentions are called *performatives*. ACL defines formal semantics for performatives (FIP 2002a) that induce basic interaction protocols upon which more complex protocols like contract nets and auctions are built.

Besides parameters that are necessary for communication like performative name, participant information, etc., an ACL message includes parameters that describe the content that is intended for the receiving participant like the *content language* the content is encoded in, the domain the content refers to, etc. In order to be qualified for the use as a content language in an ACL message, a language must meet certain requirements that are induced by the semantics of the performatives. For example a *request* requires always at least an action to be delivered with the content. Otherwise the agent that receives the request does not know what it is requested to do. Furthermore, when an agent *informs* another agent about the result of an action, the content must contain the propositions that represent the result. Thus, a content language must at least provide representations of actions and propositions, so the agents are able to interact in a meaningful way. The content language that is used by the PANDA agents is described below.

The Planning Process

Figure 4 gives an overview of the refined model of the planning process that was taken as the basis for implementation. The white colored states specify the life-cycle management of a planning session (initializing the process, starting planning, suspending it, etc.). Each state transition is labeled with the triggering ACL message and its originator: `sender:performative` followed by `action` or `proposition`. Senders can also be described by their class, e.g. `Worker` denotes all worker agents. The same applies to propositions and actions, e.g. `Compute` denotes the action `Compute` and all sub-actions like `Inspect`, `Construct`, etc.

The planning process starts in an artificial `undefined` state in which all agents are deployed and send agreements for their initialization process. After that, the strategy informs all agents, that the system is initialized, and this is where the reference model started with phase 1: The assistants are requested to perform their inference on which they have to agree. After their processing (leaving the `assisting` state), the inspectors are requested to search for flaws (phase 2), and so on.

Please note, that not all states have been modeled in the state machine. Most states are abstract in order to reduce complexity of the state automaton while maintaining a degree of granularity that allows the user to monitor the planning process. E.g., the state `backtracking` summarizes all possible sub-states that describe the interaction between each particular worker agent and the strategy.

The refined planning process model has two major improvements over the reference model: First, it extends agent concurrency. The reference planning process model in Fig. 1, defines the agent classes to execute one after another, synchronized by phase transitions. In that model, concurrency is only allowed within a particular phase. But especially between phase 2 and 3 such a synchronization is too strict, because a constructor must wait until the last inspector has finished, even if a constructor has already received all flaws it requires for computation. Constructors should therefore be able to decide on their own when to start execution. The refined process model reflects this by a combined `inspecting&constructing` state. The constructors' behavior has therefore been changed from reactive to proactive – resulting in a stronger notion of agency.

Second, an enhanced backtracking procedure allows for the implementation of optimized and more sophisticated

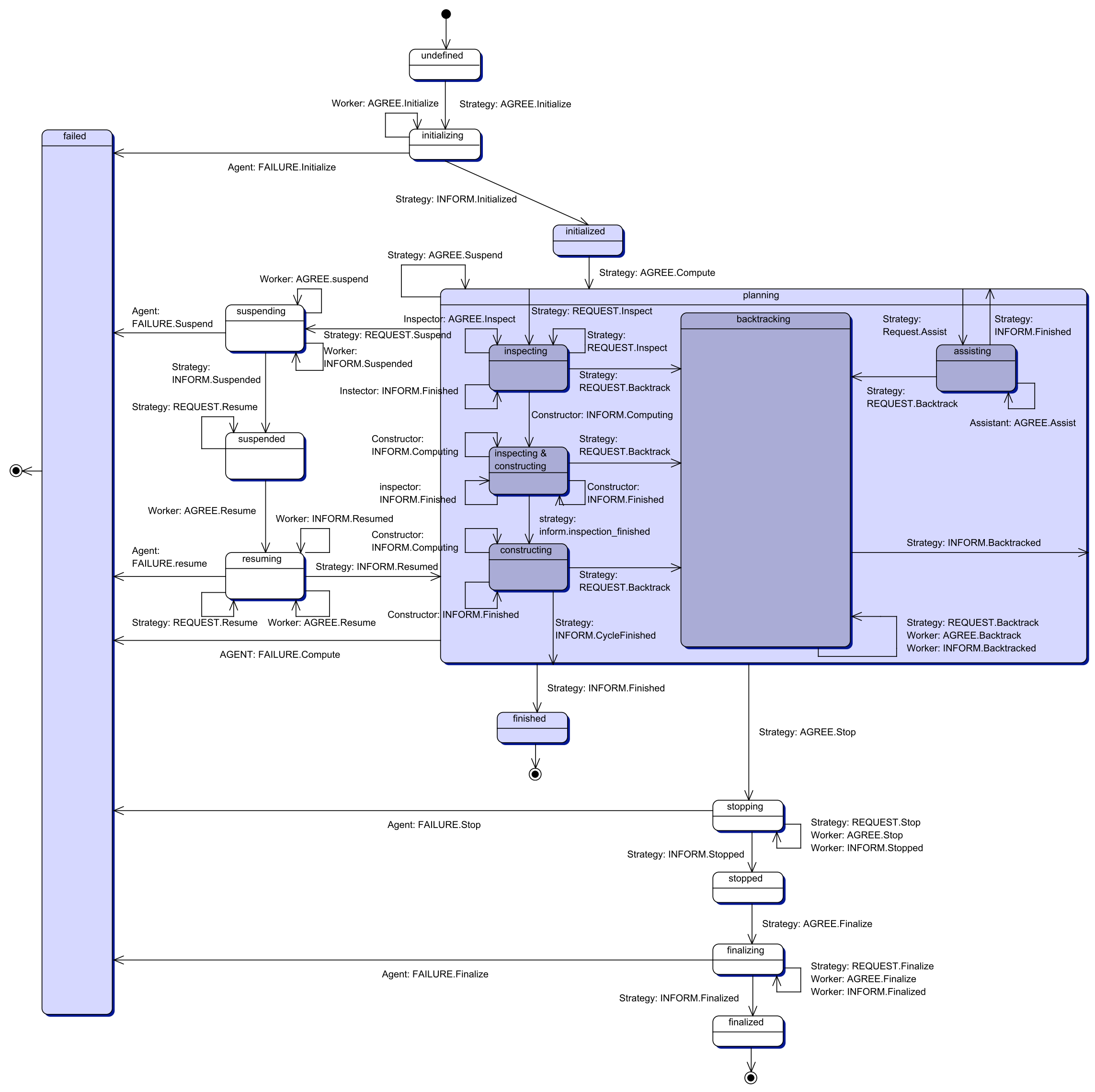

Figure 4: The refined PANDA planning process model

reasoning techniques, including worker agents, i.e. assistants, inspectors, and constructors, to be equipped with a state history or caches, etc. To keep backtracking consistent, the worker agents now participate in the backtracking process: they have to synchronize via agreement statements and then notify the strategy when they are finished (cf. state transitions from `backtracking`).

Thus, the agent behavior is extended by a backtracking mechanism with three core capabilities: First, a synchronized restart of the system must be guaranteed, i.e. a restart can only take place, if all worker agents have finished backtracking. Second, the different granularity of the state histories of agents working in different sub-cycles is considered. Assistants can be executed multiple times in a planning cycle, whereas inspectors and constructors will be executed only once. Therefore, assistants must be backtracked independently from Inspectors and Constructors. Third, in order to backtrack the system immediately, the strategy must be able to interrupt the worker agents' execution, the agents therefore perform their computations in a non-blocking way.

In summary, the enhanced mechanisms for concurrency and backtracking allow the system to benefit from early fail decisions in terms of an increased performance: Non-repairable inconsistencies are typically very quickly detected and processed by constructors.

Ontology-based Components

Like it has been mentioned before, DAML is used as representation formalism to describe and share knowledge in the PANDA system, ranging from flaw communication to system state transitioning. It is one of the emerging standards in the Semantic Web community for representing and communic-

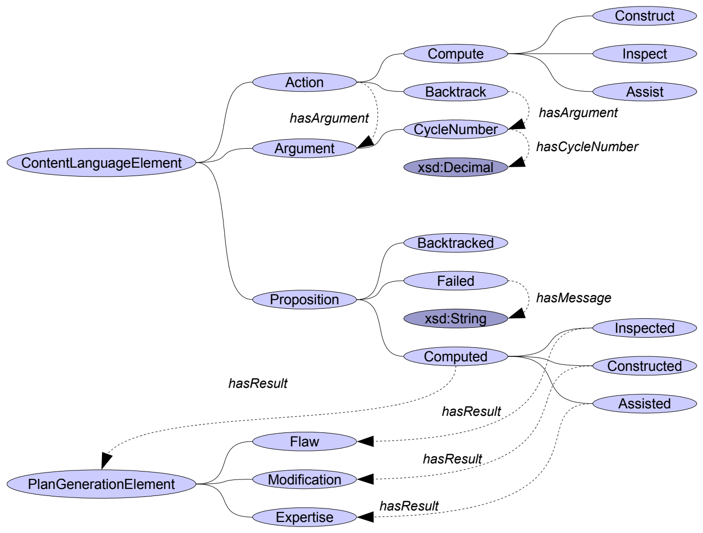

Figure 5: The content language ontology

ating knowledge (Horrocks, Harmelen, & Patel-Schneider 2001). It ensures interoperability with third-party systems like RACER and forms the basis for communicating knowledge among agents. Most important, it enables knowledge to be represented in a uniform, explicit, and declarative manner, so the system becomes more robust, flexible, and maintainable.

To be able to use DAML as a content language in ACL messages (recall the speech act structure), at least actions and propositions must be able to be represented within DAML (Schalk *et al.* 2002). This is sufficient for the needs of PANDA. Figure 5 shows the ontology that provides the concepts to enable DAML-based communication. Property cardinalities have been omitted for clarity.

Actions can have arguments, e.g., the action sub-concept `Backtrack` must come with a `CycleNumber` whose value is represented as the XML-schema type `decimal`. So an action could be compared with a method signature without argument order. In PANDA, every argument of an action is modeled in the ontology in order to give it a formal semantics. Therefore, in contrast to (Schalk *et al.* 2002), the argument order does not have to be considered. Propositions are currently only used to represent `ActionResults`. The sub-concept `Computed` carries the `PlanGenerationElements` that are the results of the worker agents' computations, e.g. a `Constructed` proposition has an `Modification` element as a result. The content of an ACL message is represented by instances of the PANDA system ontology embedded in a DAML-document. JENA takes care of encoding and decoding the DAML content of ACL messages. For any content that has to be sent, its JENA object model is constructed using the described ontology. After that, the model is serialized and inserted into the appropriate ACL message. The decoding of DAML content works exactly the opposite way. The object model of the DAML content is constructed by parsing its serialized representation and can then be queried with the JENA API.

DAML plays its second key role in the automated configuration of the agent container (Figure 6 shows the underlying ontology). The configuration process is composed of two sub-processes. First, the agents that are part of the planning process must be instantiated. The PandaSessionBean uses RACER to derive the leaf concepts of `PandaAgent` and to determine the implementation assignments `ImplementationElements` of the WorkerAgents. In the example of Figure 6, the assigned implementation for the `Inspector1` agent is an instance of Java class `panda.jade.agent.Inspector1Impl`. After being created, the PANDA agents insert their descriptions into the ABox of RACER, so RACER keeps track of the deployed agent instances.

Second, the communication links, viz. the implementation of the triggering function α, must be established between the instantiated agents. RACER is used by each PANDA agent on startup to derive its communication links to other agents, i.e. from which agents it will receive and to which agents it has to send messages. This is done by using the ontology to derive the dependencies between agents from defined dependencies between the flaws and modifications: The system ontology specifies which agent instance implements which type of `Inspector`, and it does the same for the constructor agents. RACER derives from that, which flaw and modification types will be generated by the agent instances, and if the model includes an α-relationship between them (`solved-by`), the agent instances' communication channels are linked. Based upon the subsumption capabilities that come with DAML and description logics, it is even possible to exploit sub-class relationships between `PlanGenerationElements` (illustrated by the bold printed concept connections in Fig. 6). An example for a modification class hierarchy are ordering relation manipulations with sub-classes *promotion* and *demotion*. Regarding flaws, the system ontology distinguishes *primitive* open preconditions and those involving *decomposition axioms* (Biundo & Schattenberg 2001).

A knowledge based configuration offers even more benefits: Imagine a less informed configuration mechanism, say, reading a respective file in XML format, that holds the descriptions on the agents to be loaded and the message links to be established between them as a representation of the triggering function α. Semantic verification can then only be based on type checking by, e.g., Java class loaders. In the presented architecture, the system model can be checked on startup for possible inconsistencies in a verification step of the planning process in state `initializing` before plan generation starts (cf. Fig. 4). An example for such an inconsistency is a constructor that is missing a link to a flaw, warnings can be issued for flaws and modifications without implementations of their generating agents, etc.

Related Work

There are two major agent-based planning architectures on the market. In the O-Plan system (Tate, Drabble, & Kirby 1994), a blackboard is examined by (in our terminology combined inspector and constructor) modules that write

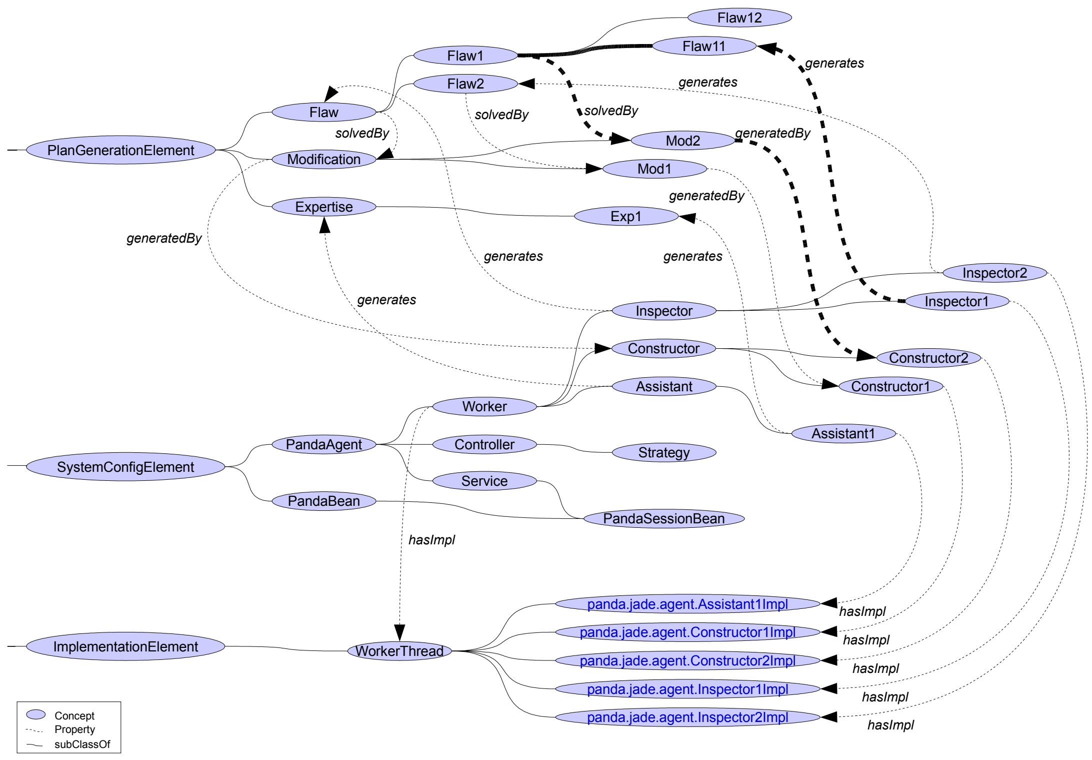

Figure 6: The system configuration ontology

their individually highest ranked flaw on the agenda of a search controller. This controller selects one agenda entry and triggers the respective module to perform its highest prioritized modification. O-Plan has been extended by a workflow-oriented infrastructure, called the I-X system integration architecture (Tate 2000). A plug-in mechanism serves as an interface to various (application tailored) tools. The planner itself is a monolithic system structure.

The Multi-agent Planning Architecture MPA (Wilkins & Myers 1998) relies upon a very generic agent-based approach. It executes an agent society in which designated coordinators decompose the planning problem into sub-problems, which are solved by subordinated groups of agents that may again decompose the problem again. Single agents return their solutions to their associated managers, which synthesizes the overall solution of its sub-agents. Communication of queries and results is based on the highly abstract KQML formalism. To our knowledge, no (standardized) middleware functionality has been incorporated.

The SIADEX architecture (de la Asunción *et al.* 2005) uses XML-RPCs for building a distributed planning environment, that is accessible via standardized HTTP and Java protocols. The architecture decouples a (monolithic) planning server, knowledge base management, and execution monitoring.

In order to address connectivity issues, some planning systems offer their functionality as web service. Examples are the CGI-based O-Plan interface (Tate & Dalton 2003) and the approach in (Tsoumakas *et al.* 2005), where a planner uses SOAP for communication and WDSL for presentation of the service. Although this view helps in enhancing the accessibility of planning software, the system (development) itself is not directly supported.

A representative for an application framework for building planning applications is Aspen (Fukunaga *et al.* 1997). It provides planning-specific data infrastructure, supportive inference mechanisms, and algorithmic templates, in order to facilitate rapid planning application development "out-of-the-box". It does not support the development of (standardized) concurrent planning functionality.

None of the above systems or architectures features a flexible, knowledge-based configuration of the plan generation process.

Conclusions and Future Work

We have presented a novel architecture for planning systems. It relies on a formal account of hybrid planning, which allows to decouple flaw detection, modification computation, and search control (Schattenberg, Weigl, & Biundo 2005). Planning capabilities, like HTN and POCL, can easily be combined by orchestrating respective elementary modules via an appropriate strategy module. The implemented system can be employed as a platform to implement and evaluate various planning methods and strategies. It can be easily extended to additional functionality, like integrated scheduling (Schattenberg & Biundo 2002; 2006) and probabilistic reasoning (Biundo, Holzer, & Schattenberg 2004; 2005), without implying changes to the deployed modules

– in particular flexible strategy modules – and without jeopardizing system consistency through interfering activity.

This work has investigated three main areas of interest of the PANDA planning system. The incorporation of higher semantics by making use of knowledge representation and inference techniques extends the capabilities of the system significantly in both functional and non-functional manner. Verification can be performed on a much higher level, and the system becomes more flexible and configurable the more hard-coded knowledge is extracted and described declaratively. With the use of application server technology and standardized communication protocols, PANDA has laid the foundation for a distributed system that is able to handle real-world application scenarios in an adequate manner. Still much work has to be done to evolve PANDA to a full-fledged planning web service, but application server technology seems to provide the appropriate architectural basis.

We plan to deploy this system as a central component in projects for assistance in telemedicine applications as well as for personal assistance in ubiquitous computing environments.

Future versions will not only keep the agents but also the planning state and agent messages in the system ontology and ABox in order to extend the verification capabilities of the system, including multiple ABoxes to enable multi-session planning. This will make the PANDA system even more robust w.r.t. corrupted agent behavior by reasoning in real-time over the dependencies between system states, possible actions and sent messages that trigger state transitions. To achieve this, knowledge about communication (i.e. message performatives, sender, receivers etc.) and interaction (i.e. FIPA-protocols and the planning process model) will be incorporated into the description logics representation of the system.

By extracting the hard-coded planning process model, describing it in a declarative manner, and executing it on a generic engine that uses this description as process template, changes to the planning process would not involve a change of code anymore. The process model itself an then be verified by transforming it into a petri net representation (Narayanan & McIlraith 2002).

References

Baader, F.; Calvanese, D.; McGuinness, D.; and Nardi, D. 2003. *The Description Logic Handbook.* Cambridge.

Bellifemine, F.; Caire, G.; Trucco, T.; and Rimassa, G. 2005. JADE programmer's guide. `http://jade.tilab.com/doc/programmersguide.pdf`.

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief – A preliminary report on combining state abstraction and HTN planning. In Cesta, A., and Borrajo, D., eds., *Proceedings of the 6th European Conference on Planning (ECP-01)*.

Biundo, S.; Holzer, R.; and Schattenberg, B. 2004. Dealing with continuous resources in AI planning. In *Proceedings of the 4th International Workshop on Planning and Scheduling for Space (IWPSS'04)*, number 228 in WPP, 213–218. ESA-ESOC, Darmstadt, Germany: European Space Agency Publications Division.

Biundo, S.; Holzer, R.; and Schattenberg, B. 2005. Project planning under temporal uncertainty. In Castillo, L.; Borrajo, D.; Salido, M. A.; and Oddi, A., eds., *Planning, Scheduling, and Constraint Satisfaction: From Theory to Practice*, volume 117 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 189–198.

Caire, G. 2005. LEAP users guide. `http://jade.tilab.com/doc/LEAPUserGuide.pdf`.

Castillo, L.; Fdez-Olivares, J.; and González, A. 2001. On the adequacy of hierarchical planning characteristics for real-world problem solving. In Cesta, A., and Borrajo, D., eds., *Proceedings of the 6th European Conference on Planning (ECP-01)*.

Cowan, D., and Griss, M. 2002. Making software agent technology available to enterprise applications. Technical Report HPL-2002-211, Software Technology Laboratory, HP Laboratories, Palo Alto. `http://www.hpl.hp.com/techreports/2002/HPL-2002-211.pdf`.

de la Asunción, M.; Castillo, L.; Fdez.-Olivares, J.; García-Pérez, O.; González, A.; and Palao, F. 2005. Knowledge and plan execution management in planning fire fighting operations. In Castillo, L.; Borrajo, D.; Salido, M. A.; and Oddi, A., eds., *Planning, Scheduling, and Constraint Satisfaction: From Theory to Practice*, volume 117 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 159–168.

Emmerich, W. 2000. *Engineering Distributed Objects*. Wiley. ISBN: 0-471-98657-7.

Estlin, T. A.; Chien, S. A.; and Wang, X. 1997. An argument for a hybrid HTN/operator-based approach to planning. In Steel, S., and Alami, R., eds., *Proceedings of the 4th European Conference on Planning (ECP-97)*, volume 1348 of *LNAI*, 182–194. Springer.

FIPA - Foundation for Intelligent Physical Agents. 2002a. *FIPA-ACL Communicative Act Library Specification*. `http://www.fipa.org/specs/fipa00037/SC00037J.pdf`.

FIPA - Foundation for Intelligent Physical Agents. 2002b. *FIPA-ACL Message Structure Specification*. `http://www.fipa.org/specs/fipa00061/SC00061G.pdf`.

Fukunaga, A.; Rabideau, G.; Chien, S.; and Yan, D. 1997. Towards an application framework for automated planning and scheduling. In *Proceedings of the 1997 International Symp. on AI, Robotics & Automation for Space*.

Haarslev, V., and Möller, R. 2001. Description of the racer system and its applications. In Goble, C.; McGuinness, D. L.; Möller, R.; and Patel-Schneider, P. F., eds., *Working Notes of the 2001 International Description Logics Workshop (DL-2001)*, volume 49 of *CEUR Workshop Proceedings*. ISSN 1613-0073.

Horrocks, I.; Harmelen, F.; and Patel-Schneider, P. 2001. DAML+OIL Specification (March 2001). `http://www.daml.org/2001/03/daml+oil-index.html`.

Manola, F., and Miller, E. 2003. RDF primer. `http://www.daml.org/2001/03/daml+oil-index.html`.

McBride, B. 2000. Making software agent technology available to enterprise applications. `http://www-uk.hpl.hp.com/people/bwm/papers/20001221-paper/`.

Narayanan, S., and McIlraith, S. A. 2002. Simulation, verification and automated composition of web services. In *WWW '02: Proceedings of the 11th International Conference on World Wide Web*, 77–88. ACM Press. ISBN 1-58113-449-5.

Schalk, M.; Liebig, T.; Illmann, T.; and Kargl, F. 2002. Combining FIPA ACL with DAML+OIL - a case study. In Cranefield, S.; Finin, T.; and Willmott, S., eds., *Proceedings of the Second International Workshop on Ontologies in Agent Systems*.

Schattenberg, B., and Biundo, S. 2002. On the identification and use of hierarchical resources in planning and scheduling. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS-02)*, 263–272. AAAI.

Schattenberg, B., and Biundo, S. 2006. A unifying framework for hybrid planning and scheduling. In Freksa, C., and Kohlhase, M., eds., *KI 2006: Advances in Artificial Intelligence, Proceedings of the 29th German Conference on Artificial Intelligence*, LNAI. Springer. to appear.

Schattenberg, B.; Weigl, A.; and Biundo, S. 2005. Hybrid planning using flexible strategies. In Furbach, U., ed., *KI 2005: Advances in Artificial Intelligence, Proceedings of the 28th German Conference on Artificial Intelligence*, volume 3698 of *LNAI*, 258–272. Springer.

Stark, S. 2003. *JBoss Administration and Development*. JBoss Group, LLC, second edition. JBoss Version 3.0.5.

Sun Microsystems. 1999. *Simplified Guide to the Java 2 Platform, Enterprise Edition*. `http://java.sun.com/j2ee/white/j2ee_guide.pdf`.

Sun Microsystems. 2003. *Enterprise JavaBeans 2.1 Documentation*. `http://java.sun.com/products/ejb/docs.html`.

Tate, A., and Dalton, J. 2003. O-plan: a common lisp planning web servide. In *Proceedings of the International Lisp Conference*, 12–25.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-Plan2: An architecture for command, planning and control. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. Morgan Kaufmann. 213–240.

Tate, A. 2000. Intelligible ai planning. In *Research and Development in Intelligent Systems XVII, Proceedings of the ES2000, The 20th British Computer Society Special Group on Expert Systems International Conference on Knowledge Based Systems and Applied Artificial Intelligence*, 3–16. Springer.

Tsoumakas, G.; Meditskos, G.; Vrakas, D.; Bassiliades, N.; and Vlahavas, I. 2005. Web services for adaptive planning. In Castillo, L.; Borrajo, D.; Salido, M. A.; and Oddi, A., eds., *Planning, Scheduling, and Constraint Satisfaction: From Theory to Practice*, volume 117 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 159–168.

Wilkins, D., and Myers, K. 1998. A multiagent planning architecture. In Simmons, R.; Veloso, M.; and Smith, S., eds., *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, 154–163. AAAI.