Knowledge-based Middleware as an Architecture 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

We present an architecture that provides a robust, scalable and flexible software framework for planning and scheduling systems through the use of standardized industrial-strength middleware and multi-agent technology. It utilizes knowledge-based components that dynamically perform and verify the system's configuration.

The system is based on a proper formal account of hybrid planning, the integration of HTN and POCL planning, which 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 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. 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.

(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. 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 can be specified declaratively and search can be performed on the basis of flaws and modifications without taking their actual computation into account. This 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 system design, in which arbitrary system configurations, i.e. planning and scheduling functionalities, can be seamlessly integrated. A corresponding prototype served as an experimental environment for the evaluation of flexible strategies and demonstrated the expansibility of the system: the integration of scheduling (Schattenberg & Biundo 2002) and probabilistic reasoning (Biundo, Holzer, & Schattenberg 2005).

In addition to an effective planning and scheduling methodology, real-world application scenarios like crisis management support, assistance in telemedicine, and personal assistance in ubiquitous computing environments call for software support to address the following issues: 1.) a declarative, automated system configuration and verification – for flexible and safe application-specific system tailoring; 2.) scalability, including transparency with respect to system distribution, access mechanisms, concurrency, etc. – for providing computational power on demand; 3.) standards compliance – for interfacing with other services and software environments.

This paper describes a novel planning and scheduling system architecture that addresses these issues and does so on the solid base of a formal framework. The resulting system dynamically configures its components and even reasons about the consistency of that configuration. The planning components are transparently deployed and distributed while retaining a simple programming model for the component developer.

Multiagent techniques have been proven to provide an appropriate architectural basis. The O-Plan system (Tate, Drabble, & Kirby 1994) uses a blackboard mechanism on which planning related knowledge sources perfom changes of the plan data structure. It 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 application tailored tools, whereas the planner itself remains to be a monolithic sub-system.

The Multi-agent Planning Architecture MPA (Wilkins & Myers 1998) relies on a very generic agent-based approach for a more general kind of problem solving. Designated coordinators decompose the planning problem into sub-problems, which are solved by subordinated groups of agents that may again decompose and delegate the problem.

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 sets of PL1 literals 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 inital state, an initial task network, i.e. an abstract plan, optionally a goal state specification, 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 only primitive operators, the ordering and variable constraints are consistent, and the causal links support all operator preconditions, including the goal state description, without being threatened. The violation of solution criteria is made explicit by *flaws* – data structures which literally “point” to deficiencies in the plan and allow for the problems' classification: A flaw $\mathtt{f}$ 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: $(\mathtt{Threat}, \{\langle te_i, \phi, te_j\rangle, te_k\})$. 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 $f_x^{det} : \mathcal{P} \rightarrow 2^{\mathcal{F}_x}$. 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:

$f_{CausalThreat}^{det}(P) \ni (\mathtt{Threat}, \{\langle te_i, \phi, te_j\rangle, te_k\})$ 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 $\sigma(\phi) \in \sigma(\mathrm{del}(te_k))$ for positive and $\sigma(|\phi|) \in \sigma(\mathrm{add}(te_k))$ for negative literals ϕ.

The refinement steps for obtaining a solution out of a problem specification are explicit representations of changes to the plan structure. A plan modification $\mathtt{m}$ 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. Adding an ordering constraint between two plan steps te_i and te_j is represented by $(\mathtt{AddOrdCons}, \{\oplus(te_i \prec te_j)\})$. Hybrid planning also uses the insertion of new action schema instances, variable (in-) equations, and causal links, and the expansion of complex tasks according to appropriate methods.

Modification generation is encapsulated by modules $f_y^{mod} : \mathtt{P} \times 2^{\mathcal{F}_x} \rightarrow 2^{\mathcal{M}_y}$ which compute all plan refinements that solve flaws. E.g., promotion and demotion as an answer to a causal threat is defined as:

$$f_{\mathtt{AddOrdCons}}^{mod}(P, \{\mathtt{f}, \ldots\}) \supseteq \{\, (\mathtt{AddOrdCons}, \{\oplus(te_k \prec te_i)\}), (\mathtt{AddOrdCons}, \{\oplus(te_j \prec te_k)\})\}$$

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

Obviously, particular classes of modifications address particular classes of flaws. This relationship is explicitly represented by a *modification triggering function* α which relates flaw classes with suitable modification classes. 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 inequality constraints (Biundo & Schattenberg 2001):

$$\alpha(\mathcal{F}_{\mathtt{Threat}}) = \mathcal{M}_{\mathtt{ExpandTask}} \cup \mathcal{M}_{\mathtt{AddOrdCons}} \cup \mathcal{M}_{\mathtt{AddVarCons}}$$

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 $\bigcup_{1 \leq i \leq n} f_{y_i}^{mod}(P, f_x^{det}(P)) = \emptyset$ then P cannot be refined into a solution.

A generic algorithm can then be defined which uses these modules (see Algorithm 1): In a first loop, the results of all detection module implementations are collected. A second loop propagates the resulting flaws according to the triggering function α to the respective modification modules, any un-answered flaw indicates a failure. A strategy module selects (a backtracking point) finally the most promising modification, which is then applied to the plan. The algorithm is called recursively with that modified plan.

Algorithm 1 : The generic planning algorithm

```
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)
```

The framework allows for a formal definition of various planning strategies, ranging from existing strategies of the literature to novel *flexible* strategies (Schattenberg, Weigl, & Biundo 2005). The latter exploit the explicit flaw and modification information to perform an opportunistic search. E.g., a *least commitment* strategy selects modifications from the smallest answer set of the detected flaws, i.e., any *class* of modification may be chosen in each cycle.

Architecture Overview

The basic architecture of our system is a multiagent blackboard system. The agent societies correspond to the presented module structure, with the agent metaphor providing maximal flexibility for the implementation. *Inspectors* are implementations of flaw detection modules, *Constructors* that of plan modification generating modules. *Assistants* perform inferences that are required by other agents. They propagate implications of temporal information transparently into the ordering constraints, canonize constraints, etc. *Strategies* implement planning strategy modules; they synchronize the execution of the other agents, select modifications, initiate backtracking, etc. The planning process breaks down in 4 phases (cf. Fig. 1):

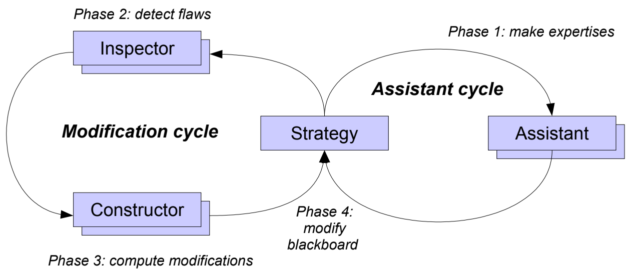

Figure 1: The reference planning process model

Phase 1: Assistants repeatedly derive additional information and post it on the blackboard until no member of that community adds information anymore.

Phase 2: Inspectors analyse the current plan on the blackboard and post the detected flaws to the strategy and to the constructors that are assigned to them via α.

Phase 3: Constructors compute all modifications solving the received flaws and send them to the strategy.

Phase 4: The strategy compares the received results and selects one of the modifications for execution.

Phase transitions are performed by the strategy when all participating agents have finished their parallel 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, i.e., if the current plan consitutes a solution to the given planning problem, or the planning process has reached a dead end and the system has to backtrack.

An Ontology-based Middleware

A naive implementation of the reference process model would run 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 sources, the planning process requires a lot of computational power, etc. Scalability and distribution play therefore key roles in the proposed architecture, while maintaining the (simple but effective) reference process.

The main aspect in middleware systems like application servers is to keep distribution issues *transparent* to the programmer by providing an abstract interface. This includes location and scalability transparency; the latter means that it is completely transparent to the programmer how a middleware system scales in response to a growing load. In order to benefit from application server technology, our system builds upon an open-source implementation (Stark 2003).

Regarding agent life-cycle management and communication mechanisms, BlueJADE (Cowan & Griss 2002) integrates a FIPA-compliant[1], standardized agent platform with the application server and provides all agent specifics. It puts the agent system life cycle under full control of the application server, therefore all its distribution capabilities apply to the agent societies. The reference process (Fig. 1) omits the actual means for *calling* agents; in a distributed implementation, these remote calls are typically message-based. Our messages are encoded in the agent communication language *FIPA-ACL* that is based on speech-act theory. Every message describes an action that is intended to be carried out with that message simultanously, a *performative*, e.g. a request “compute detections”.

Agents can be spread and migrated transparently over several BlueJade agent containers running on different nodes in a network, even on mobile devices such as Java capable cellular phones and PDAs. This distribution management is “on top” of the middleware facilities: agent migration typically anticipates pro-actively computation or communication requirements in a relative abstract manner, while middleware migration reacts on load changes based on low-level operating system specific metrics. It makes sense to provide both mechanisms in parallel, e.g. to migrate agents, which are known to require much computational resources onto dedicated compute servers.

The last component is the knowledge representation and reasoning facilities which are used throughout the system. Not only planning related concepts, i.e. flaw and modification classes, are explicitly represented, but also the system configuration and the plan generation process itself. This enables us to effectively configure parts of the system without interfering with its implementation; given that it is represented in a formalism that is expressive enough to capture all modelling aspects on one side and that allows efficient reasoning on the other side. 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; it combines the key features of description logics with internet standards such as XML or RDF. By using DAML as the content language for the BlueJADE agent communication (Schalk *et al.* 2002) and also as the language for describing system configurations and communication means, we achieve a homogenous representation of knowledge in the system. To this end we developed a respective content language ontology that provides the necessaray concepts like

[1]Foundation for Intelligent Physical Agents

flaws, modifications, system actions (computation and backtracking procedures), respective performatives, etc., and that describes the appropriate message structures. All knowledge is stored and reasoned about by the description logic system RACER (Haarslev & Möller 2001).

DAML plays its second key role in the automated configuration of the agent container via a system ontology which captures, among others, knowledge about the implemented agent classes, which flaws or modification classes they generate, and the triggering function. The configuration process is composed of two sub-processes. First, the agents that are part of the planning process must be instantiated. The RACER reasoner is used to derive the leaf concepts of the abstract agents classes *Inspector*, etc., and to determine the implementation assignments, basically references to the corresponding software artefacts. After their creation, agents insert their descriptions into the ABox of RACER, which keeps track of the deployed agent instances. Second, the communication links for implementing the triggering function α have to be established. 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. If the model includes an α-relationship between the latter, modelled by a respective property, the agents' communication channels are linked. Based upon the subsumption capabilities that come with description logics, it is even possible to exploit sub-class relationships. An example for a hierarchy on modification classes are ordering relation manipulations with sub-classes *promotion* and *demotion*. Or 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: Uninformed configuration mechanisms can only be based on type checking by, e.g., Java class loaders. The presented architecture can check the specified system model on startup for possible inconsistencies, e.g., constructors with missing links to suitable inspectors, undefined implementations for flaws and modifications, etc.

Conclusions and Future Work

We have presented a novel agent-based architecture for planning systems. It relies on a formal account of hybrid planning, which allows to decouple flaw detection, modification computation, and search control. Planning capabilities like HTN and POCL can easily be combined by orchestrating respective agents and an appropriate strategy. The implemented system can be employed as a platform to implement and evaluate various planning methods and strategies. Extended functionality, e.g. the integration of scheduling and probabilistic reasoning, can easily be achieved without requiring changes to the deployed agent implementations – in particular when using flexible strategies – and without jeopardizing system consistency through interfering activity.

The exploitation of a semantics for system components and configurations, by making use of description logics representation and inference techniques, significantly extends the capabilities of the system. High level configuration verification can be performed, and the system becomes more flexible and configurable as previously implicitly encoded knowledge is made explicit. With the use of application server technology and standardized communication protocols, we have laid the foundation for fielding our system in real-world application scenarios; we plan to deploy it as a central component in projects for assistance in telemedicine applications as well as for personal assistance in ubiquitous computing environments.

Future versions will extend the ontology representation, including knowledge about communication (message structures, etc.) and interaction (protocols and the planning process). Making the planning process model explicit and interpretable allows for tailoring the planning system to applications without changing any code. Furthermore, the process model itself can be verified by transforming it into a petri net representation (Narayanan & McIlraith 2002).

References

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief – A preliminary report on combining state abstraction and HTN planning. In *Proceedings of ECP-01*.

Biundo, S.; Holzer, R.; and Schattenberg, B. 2005. Project planning under temporal uncertainty. In *Planning, Scheduling, and Constraint Satisfaction: From Theory to Practice*, volume 117. IOS Press. 189–198.

Cowan, D., and Griss, M. 2002. Making software agent technology available to enterprise applications. Technical Report HPL-2002-211, Software Technology Laboratory, HP Laboratories.

Haarslev, V., and Möller, R. 2001. Description of the racer system and its applications. In *Working Notes of DL-2001*.

Horrocks, I.; Harmelen, F.; and Patel-Schneider, P. 2001. DAML+OIL Specification (March 2001).

Narayanan, S., and McIlraith, S. A. 2002. Simulation, verification and automated composition of web services. In *Proceedings of WWW '02*, 77–88. ACM Press.

Schalk, M.; Liebig, T.; Illmann, T.; and Kargl, F. 2002. Combining FIPA ACL with DAML+OIL - a case study. In *Proc. of the 2nd Int. 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 *Proceedings of AIPS-02*, 263–272. AAAI.

Schattenberg, B.; Weigl, A.; and Biundo, S. 2005. Hybrid planning using flexible strategies. In *Proc. of the 28th German Conf. on AI*, volume 3698 of *LNAI*, 258–272. Springer.

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

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

Tate, A. 2000. Intelligible AI planning. In *Proceedings of ES-2000*, 3–16. Springer.

Wilkins, D., and Myers, K. 1998. A multiagent planning architecture. In *Proceedings of AIPS-98*, 154–163. AAAI.