Time for AI and Society

PROCEEDINGS OF THE AISB'00 SYMPOSIUM ON AI PLANNING AND INTELLIGENT AGENTS

17th-20th April, 2000
University of Birmingham

AISB'00 Convention

17th-20th April 2000

University of Birmingham
England

Proceedings of the AISB'00 Symposium on

AI Planning and Intelligent Agents

Published by

The Society for the Study of Artificial Intelligence and the Simulation of Behaviour

United Kingdom

http://www.cogs.susx.ac.uk/aisb/

ISBN 1 902956 14 6

Printed at the University of Birmingham, Edgbaston, Birmingham B15 2TT, England.

Contents

The AISB'00 Convention

The millennial nature of current year, and the fact that it is also the University of Birmingham's centennial year, made it timely to have the focus of this year's Convention be the question of interactions between AI and society. These interactions include not just the benefits or drawbacks of AI for society at large, but also the less obvious but increasingly examined ways in which consideration of society can contribute to AI. The latter type of contribution is most obviously on the topic of societies of intelligent artificial (and human) agents. But another aspect is the increasing feeling in many quarters that what has traditionally been regarded as cognition of a single agent is in reality partly a social phenomenon or product.

The seven symposia that largely constitute the Convention represent various ways in which society and AI can contribute to or otherwise affect each other. The topics of the symposia are as follows: Starting from Society: The Application of Social Analogies to Computational Systems; AI Planning and Intelligent Agents; Artificial Intelligence in Bioinformatics; How to Design a Functioning Mind; Creative and Cultural Aspects of AI and Cognitive Science; Artificial Intelligence and Legal Reasoning; and Artificial Intelligence, Ethics and (Quasi-)Human Rights. The Proceedings of each symposium is a separate document, published by AISB. Lists of presenters, together with abstracts, can be found at the convention website, at http://www.cs.bham.ac.uk/~mgl/aisb/.

The symposia are complemented by four plenary invited talks from internationally eminent AI researchers: Alan Bundy ("what is a proof?"- on the sociological aspects of the notion of proof); Geoffrey Hinton ("how to train a community of stochastic generative models"); Marvin Minsky ("an architecture for a society of mind"); and Aaron Sloman ("from intelligent organisms to intelligent social systems: how evolution of meta-management supports social/ cultural advances"). The abstracts for these talks can be found at the convention website.

We would like to thank all who have helped us in the organization, development and conduct of the convention, and especially: various officials at the University of Birmingham, for their efficient help with general conference organization; the Birmingham Convention and Visitor Bureau for their ready help with accommodation arrangements, including their provision of special hotel rates for all University of Birmingham events in the current year; Sammy Snow in the School of Computer Science at the university for her secretarial and event-arranging skills; technical staff in the School for help with various arrangements; several research students for their volunteered assistance; the Centre for Educational Technology and Distance Learning at the university for hosting visits by convention delegates; the symposium authors for contributing papers; the Committee of the AISB for their suggestions and guidance; Geraint Wiggins for advice based on and material relating to AISB'99; the invited speakers for the donation of their time and effort; the symposium chairs and programme committees for their hard work and inspirational ideas; the Institue for Electrical Engineers for their sponsorship; and the Engineering and Physical Sciences Research Council for a valuable grant.

John Barnden & Mark Lee

AI PLANNING AND INTELLIGENT AGENTS

Introduction

AIMS AND THEMES OF THE SYMPOSIUM

Automated Planning has been a central research area in Artificial Intelligence for over thirty years. AI Planning is increasingly being exploited in high-profile projects involving space and military applications, and the growing interest in this area has led to the setting up in 1998 of the EU-funded network of excellence in AI Planning, PLANET. Intelligent Agents have more recently become a topic of great interest to various branches of both AI and computer science. Intelligent agents can be characterised by their capability for autonomous action, which they can modify as appropriate in order to meet their goals. They can perceive and respond to their environments when problem-solving, interacting with other agents if necessary. Intelligent Agents need to reason and plan, often in real time, whilst a plan produced by an AI planning engine must eventually be executed by some agent. A good illustration of this is the recent RAX (Remote Agent Experiment) project on NASA's Deep Space 1 spacecraft which allowed the primary command of a spacecraft to be given to a Remote Agent which incorporated a planning component.

We feel an exchange of ideas and views between these 2 areas is timely and will be both stimulating and useful. We hope this Symposium will provide a forum for the discussion of both current issues in AI Planning and Scheduling and of the connections between them and agent-based reasoning. During the Symposium we would also like to address some of the more fundamental questions regarding the interplay between these two areas. For example, although there are various definitions of Intelligent Agents, it is not clear what the significance of the planning function within an Intelligent Agent is. Do all Intelligent Agents have to plan? What *sort* of planning must an Intelligent Agent do? Is reactive planning sufficient or does generative planning also have a role to play? We need to clarify the relationship between generative and reactive planning, and the function of Intelligent Agents. In particular, the contributors to our Symposium will be looking at such issues as:

- how agents deal with dynamic environments, including uncertainty and resources
- how we can plan with incomplete information
- how we can apply the agent paradigm to scheduling problems
- how agents can communicate with each other

We hope that in the process of considering these issues, we can come to some conclusions regarding the fundamental questions posed above. Additionally, we would like the Symposium to work towards the identification of a set of common research problems in Agent Technology and AI Planning and Scheduling as well as areas ripe for cross-fertilisation.

SYMPOSIUM PROGRAMME COMMITTEE

- Diane Kitchin, University of Huddersfield (SYMPOSIUM CHAIR)
- Ruth Aylett, University of Salford
- Lee McCluskey, University of Huddersfield
- Julie Porteous, University of Durham
- Sam Steel, University of Essex

Planning as Abductive Updating

José Júlio Alferes*†; João Alexandre Leite†
Luís Moniz Pereira†; Paulo Quaresma*†
*Universidade de Évora, R. Romão Ramalho, 59, 7000 Évora, Portugal
†Centro de Inteligência Artificial (CENTRIA), Departamento de Informática
Universidade Nova de Lisboa, 2825-114 Caparica, Portugal
jja@dmat.uevora.pt; jleite@di.fct.unl.pt
lmp@di.fct.unl.pt; pq@dmat.uevora.pt

Abstract

In this paper we show how planning can be achieved by means of abduction, a form of non-monotonic reasoning, in the *LUPS* language. *LUPS* employs the recently introduced notion of *Dynamic Logic Programming*, whereby the knowledge representation rules, namely those representing actions, can dynamically change, crucial when agents are to be situated in evolving environments. By integrating into a single framework several recent developments in the logic programming and non-monotonic reasoning field of research, this work contributes to a better modeling and understanding of rational agents. At the same time, it enjoys the advantages of a declarative and implementable specification, shortening the usual gap between theory and practice often found in logical based approaches to agents. The system integrating *Dynamic Logic Programming*, *LUPS* and *Abduction*, in order to achieve this form of planning, has been implemented.

1 Introduction and Motivation

In the last few years *agent-based computing* has been one of the most debated concepts. Being a paradigm that virtually invaded every sub-field of computer science, it found in imperative languages the most common adopted vehicle to its implementation, mainly for reasons of efficiency. However, since efficiency is not always a real issue, but clear specification and correctness is, *Logic Programming* and *Non-monotonic Reasoning* have been brought (back) to the spot-light. Add to this significant recent improvements in the efficiency of *Logic Programming* implementations (Niemelä and Simons, 1997; XSB System, 1999). Besides allowing for a unified declarative and procedural semantics, eliminating the traditional high gap between theory and practice, the use of several and quite powerful results in the field of non-monotonic extensions to *Logic Programming (LP)* can represent an important added value to the design of rational agents. For a better understanding a thorough exposition of how *Logic Programming* can contribute to agent-based computing, the reader is referred to Sadri and Toni (1999) and Bozzano et al. (1999). Embedding agent rationality in the LP paradigm affords us with a number of tools and formalisms captured in that paradigm, such as belief revision, inductive learning, argumentation, preferences, etc. (Kowalski and Sadri, 1996; Rochefort et al., 1999)

Traditionally, the work on logic programming was mainly focused on representing static knowledge, i.e. knowledge that does not evolve with time. Some work had been done on updating knowledge bases but limited to factual updates. The problem of updating the knowledge rules, as opposed to updating the models generated by them remained an open issue. Recently, in Leite and Pereira (1997), the authors argued that the *principle of inertia* could be successfully applied to the rules of a knowledge base, instead of to the literals in its models, thereby yielding the desired result. This lead to the introduction of the paradigm of *Dynamic Logic Programming* (Alferes et al., 1998, 2000).

Dynamic Logic Programming, supported by the notion of *Logic Program Updates*, is simple and quite fundamental. Suppose that we are given a set of theories (encoded as generalized logic programs) representing different states of the world. Different states may represent different time periods or different sets of priorities. Consequently, the individual theories contain mutually contradictory as well as overlapping information. The rôle of *Dynamic Logic Programming* is to use the mutual relationships existing between different states to precisely determine the declarative as well as the procedural semantics of the combined theory composed of all individual theories.

Although solving the problem of dynamically evolving logic programs, *Dynamic Logic Programming* does not by itself provide a proper language for its specification. To achieve this goal, and in particular to allow logic programs to describe transitions of knowledge states in addition to the knowledge states themselves, Alferes et al. (1999) introduced the language *LUPS*. *LUPS* al-

lows the association, with each state, of a set of transition rules, providing an interleaving sequence of states and transition rules in an integrated declarative framework. It is worth pointing out that, the most notable difference between *LUPS* and Action Languages (Gelfond and Lifschitz, 1998) is that the latter deal only with updates of propositional knowledge states while *LUPS* updates knowledge states that consist of knowledge rules i.e. the outcome of a *LUPS* update is not a simple set of propositional literals but rather a set of rules. *LUPS* also makes it easier to specify so-called *"static laws"*, to deal with indirect effects of actions and to represent, and reason about, simultaneous actions.

It is perhaps useful to remark at this point that in *imperative programming* the programmer specifies only transitions between different knowledge states while leaving the actual (resulting) knowledge states implicit and thus highly imprecise and difficult to reason about. On the other hand, dynamic knowledge updates, as described above, enabled us to give a precise and fully declarative description of actual knowledge states but did not offer any mechanism for specifying state transitions. With the high-level language of dynamic updates, we are able to make *both* the knowledge states and their transitions fully declarative and precise.

Rational agents must be able to reason about the knowledge they possess and, among other things, plan to achieve their goals. In general, planning consists of the deliberative process by which a set of (partially or totally) ordered actions is generated to achieve one or more goals. Most approaches to agents based on computational logic achieve planning but none of them allowing for dynamically changing rules since they are limited to static intensional knowledge. For a survey and roadmap of computational logic based agents the reader is referred to Sadri and Toni (1999) and Bozzano et al. (1999).

Examples where this dynamic behaviour of rules is essential, where new rules come into play while, at the same time, some rules cease to be valid, can be found in *Legal Reasoning*, namely in what concerns the application of law over time. In countries with legal systems where laws are often changed, jurisprudence makes heavy use of the articles governing the application of law in time. The representation of and reasoning about such articles is not trivial, being most important the part dealing with transient situations, for example when an event occurs after some new law has been approved but hasn't taken effect yet. Different outcomes can be obtained depending on the date of the trial. In such situations an agent acting as a lawyer for example, would have to plan its course of action in quite complex situations due to the changing rules. In Sect. 5 we present an example of such a situation.

Other examples can be found in computer games (often not taken very seriously by the computer science community but extremely challenging and lucrative (Jennings et al., 1998)) where an agent has to deal with partially different rules from level to level, and cross level planning is needed. In general, this framework is quite useful, in all its power, in situations where agents with learning capabilities are moved into slightly different domains (or the domain descriptions change) and it is useful to use the knowledge from the previous domain.

In this paper we show how planning can be achieved by means of abduction, a form of non-monotonic reasoning, in the *LUPS* language, where actions to be performed can be envisaged as abduced update rules. By integrating into a single framework several present developments in the logic programming and non-monotonic reasoning field of research, this work contributes to a better modeling and understanding of rational agents while, at the same time, it enjoys the advantages of a declarative and implementable specification, shortening the usual gap between theory and practice often found in logical based approaches to agents (Sadri and Toni, 1999).

The system integrating *Dynamic Logic Programming*, *LUPS* and *Abduction*, to achieve this form of planning, has been implemented and tested on top of the XSB System (1999). This overall system allows for several forms of reasoning having many applications currently being explored.

The paper is structured thus: After an introductory section to briefly recap *Dynamic Logic Programming* and *LUPS*, the planning problem in *LUPS* is formalized and its solutions characterized. This is followed by the presentation of an implemented solution based on abduction, proven correct according to the formal characterization. Finally we illustrate with an example and conclude.

2 Dynamic Logic Programming and LUPS

In the LUPS framework (Alferes et al., 1999), knowledge evolves from one state to another as a result of sets of (simultaneous) update commands. In order to represent *negative* information in logic programs and in their updates, the framework resorts to more general logic programs, those allowing default negation *not A* not just in the premises of their rules but in their heads as well. In Alferes et al. (1998), such programs are dubbed *generalized logic programs*:

Definition 1 (Generalized Logic Program) *A* generalized logic program *P in the language $\mathcal{L}$ is a (possibly infinite) set of propositional rules of the form:*

$$L \leftarrow L_1, \ldots, L_n \qquad (1)$$

where L and L_i are literals. A literal is either an atom A or its default negation not A. Literals of the form not A are called default literals. If none of the literals appearing in heads of rules of P are default literals, then the logic program P is normal. ◇

The semantics of generalized logic programs is defined as a generalization of the stable models semantics (Gelfond and Lifschitz, 1988).

Definition 2 (Default assumptions) *Let M be a model of P. Then:*

$$Default(P, M) = \{not\ A \mid \nexists r \in P : head(r) = A, M \models body(r)\} \quad \diamond$$

Definition 3 (Stable Models of Generalized Programs) *A model M is a stable model of the generalized program P iff:*

$$M = least(P \cup Default(P, M)) \quad \diamond$$

As proven in Alferes et al. (1998), the class of stable models of generalized logic programs extends the class of stable models of normal programs Gelfond and Lifschitz (1988) in the sense that, for the special case of normal programs both semantics coincide.

In LUPS, knowledge states, each represented by a generalized logic program, evolve due to sets of update commands. By definition, and without loss of generality Alferes et al. (1999), the initial knowledge state KS_0 is empty and, in it, all predicates are assumed false by default. Given a current knowledge state KS_i, its successor state is produced as a result of the occurrence of a non-empty set U of simultaneous update commands. Thus, any knowledge state is solely determined by the sequence of sets of updates commands performed from the initial state onwards. Accordingly, each non-initial state can be denoted by:

$$KS_n = U_1 \otimes \cdots \otimes U_n \qquad n > 0$$

where each U_i is a set of update commands.

Update commands specify assertions or retractions to the current knowledge state (i.e. the one resulting from the last update performed). In LUPS a simple assertion is represented as the command:

$$assert\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (2)$$

Its meaning is that if $L_{k+1}, \ldots, L_m$ is true in the current state, then the rule $L \leftarrow L_1, \ldots, L_k$ is added to its successor state, and persists by inertia, until possibly retracted or overridden by some future update command.

In order to represent rules and facts that do not persist by inertia, i.e. that are one-state events, LUPS includes the modified form of assertion:

$$assert\ event\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (3)$$

The retraction of rules is performed with the update command:

$$retract\ [event]\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (4)$$

Its meaning is that, subject to precondition $L_{k+1}, \ldots, L_m$ (verified at the current state) rule $L \leftarrow L_1, \ldots, L_k$ is either retracted from its successor state onwards, or just temporarily retracted in the successor state (if governed by *event*).

Normally assertions represent newly incoming information. Although its effects remain by inertia (until countervened or retracted), the assert command itself does not persist. However, some update commands may desirably persist in the successive consecutive updates. This is especially the case of laws which, subject to some preconditions, are always valid, or of rules describing the effects of an action. In the former case, the update command must be added to all sets of updates, to guarantee that the rule remains indeed valid. In the latter case, the specification of the effects must be added to all sets of updates, to guarantee that, when the action takes place, its effects are enforced.

To specify such persistent update commands, LUPS introduces:

$$always\ [event]\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (5)$$

$$cancel\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (6)$$

The first states that, from the current state onwards, in addition to any newly arriving set of commands, whenever the preconditions are verified, the persistent rule is added too. The second command cancels this persistent update.

Definition 4 (LUPS language) *An update program in LUPS is a finite sequence of updates $U_1 \otimes \cdots \otimes U_n$ where each U_i is a non-empty set of (simultaneous) commands of the forms (2)-(6).* $\diamond$

Any knowledge state KS_q ($0 \leq q \leq n$) resulting from an update program $U_1 \otimes \cdots \otimes U_n$ can be queried via "$holds(L_1, \ldots, L_m)\ at\ q$?". The query is true iff the conjunction of its literals holds at KS_q.

The semantics of LUPS (Alferes et al., 1999) is defined by incrementally translating update programs into sequences of generalized logic programs. The meaning of such sequences of programs is determined by the semantics defined in Alferes et al. (1998). Given a sequence of generalized programs $P_1 \oplus \cdots \oplus P_n$ the semantics has to ensure that the newly added rules (in the later programs of the sequence) are in force, and that previous rules are still valid (by inertia) as far as possible, i.e. they are kept for as long as they do not conflict with newly added ones. Accordingly, given a model M of the last program P_n, start by removing all the rules from previous programs whose head is the complement of some later rule with true body in M (i.e. by removing all rules which conflict with more recent ones). All other persist through by inertia. Then,

as for the stable models of a single generalized program, add facts *not* A for all atoms A which have no rule at all with true body in M, and compute the least model. If M is a fixpoint of this construction, M is a stable model of the sequence up to P_n.

Definition 5 (Dynamic Logic Program) *Let S be an ordered set with a smallest element s_0 and with the property that every $s \in S$ other than s_0 has an immediate predecessor $s-1$ and that $s_0 = s - n$ for some finite n. Then $\bigoplus\{P_i : i \in S\}$ is a Dynamic Logic Program, where each of the P_is is a generalized logic program.* ◇

Definition 6 (Rejected rules) *Let $\bigoplus\{P_i : i \in S\}$ be a Dynamic Logic Program, let $s \in S$, and let M be a model of P_s. Then*

$$Reject(s,M) = \{r \in P_i \mid \exists r' \in P_j,\ M \models body(r'),\ head(r) = not\ head(r'),\ i < j \leq s\}$$ ◇

To allow for querying a dynamic program at any state s, the definition of stable model is parameterized by the state:

Definition 7 (Stable Models of a DLP at state s) *Let $\bigoplus\{P_i : i \in S\}$ be a Dynamic Logic Program, let $s \in S$, and let $\mathcal{P} = \bigcup_{i<s} P_i$. A model M of P_s is a stable model of $\bigoplus\{P_i : i \in S\}$ at state s iff:*

$$M = least([\mathcal{P} - Reject(s,M)] \cup Default(\mathcal{P},M))$$

If some literal or conjunction of literals ϕ holds in all stable models at state s of the Dynamic Program, we write $\bigoplus_s\{P_i : i \in S\} \models_{sm} \phi$. ◇

The translation of a LUPS program into a dynamic program is made by induction, starting from the empty program P_0, and for each update U_i, given the already built dynamic program $P_0 \oplus \cdots \oplus P_{i-1}$, determining the resulting program $P_0 \oplus \cdots \oplus P_{i-1} \oplus P_i$. To cope with persistent update commands we will further consider, associated with every dynamic program in the inductive construction, a set containing all currently active persistent commands, i.e. all those that were not cancelled until that point in the construction, from the time they were introduced. To be able to retract rules, we need to uniquely identify each such rule. This is achieved by augmenting the language of the resulting dynamic program with a new propositional variable "$rule(L \leftarrow L_1,\ldots,L_n)$" for every rule $L \leftarrow L_1,\ldots,L_n$ appearing in the original LUPS program.

Definition 8 (Translation into dynamic programs) *Let $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ be an update program. The corresponding dynamic program $\Upsilon(\mathcal{U}) = \mathcal{P} = P_0 \oplus \cdots \oplus P_n$ is obtained by the following inductive construction, using at each step i an auxiliary set of persistent commands PC_i:*

***Base step**: $P_0 = \{\}$ with $PC_0 = \{\}$.*

***Inductive step**: Let $\mathcal{P}_i = P_0 \oplus \cdots \oplus P_i$ with set of persistent commands PC_i be the translation of $\mathcal{U}_i = U_1 \otimes \cdots \otimes U_i$. The translation of $\mathcal{U}_{i+1} = U_1 \otimes \cdots \otimes U_{i+1}$ is $\mathcal{P}_{i+1} = P_0 \oplus \cdots \oplus P_{i+1}$ with set of persistent commands PC_{i+1}, where PC_{i+1} is:*

$$PC_i \cup \{assert\ [event]\ (R)\ when\ (C) : always\ [event]\ (R)\ when\ (C) \in U_{i+1}\}$$
$$- \{assert\ [event]\ (R)\ when\ (C) : cancel\ (R)\ when\ (D) \in U_{i+1}, \bigoplus\nolimits_i \mathcal{P}_i \models_{sm} D\}$$
$$- \{assert\ [event]\ (R)\ when\ (C) : retract\ (R)\ when\ (D) \in U_{i+1}, \bigoplus\nolimits_i \mathcal{P}_i \models_{sm} D\}$$

$NU_{i+1} = U_{i+1} \cup PC_{i+1}$, *and* P_{i+1} *is:*

$$\left\{ \begin{array}{c} R,\ rule(R) : assert\ [event]\ (R)\ when\ (C) \in \\ NU_{i+1}, \bigoplus_i \mathcal{P}_i \models_{sm} C \end{array} \right\}$$
$$\cup \left\{ \begin{array}{c} not\ rule(R) : retract\ [event]\ (R)\ when\ (C) \in \\ NU_{i+1}, \bigoplus_i \mathcal{P}_i \models_{sm} C \end{array} \right\}$$
$$\cup \left\{ \begin{array}{c} not\ rule(R) : assert\ event\ (R)\ when\ (C) \in \\ NU_i, \bigoplus_{i-1} \mathcal{P}_{i-1} \models_{sm} C \end{array} \right\}$$
$$\cup \left\{ \begin{array}{c} rule(R) : retract\ event\ (R)\ when\ (C) \in \\ NU_i, \bigoplus_{i-1} \mathcal{P}_{i-1} \models_{sm} C, rule(R) \end{array} \right\}$$

where R denotes a generalized logic program rule, and C and D a conjunction of literals. In the inductive step, if $i = 0$ the last two lines are omitted. In that case NU_i does not exist. ◇

Definition 9 (LUPS semantics) *Let $\mathcal{U}$ be an update program. A query*

$$holds(L_1,\ldots,L_n)\ at\ q$$

is true in $\mathcal{U}$ iff $\bigoplus_q \Upsilon(\mathcal{U}) \models_{sm} L_1,\ldots,L_n$. ◇

3 LUPS and Plans

LUPS, by allowing to declaratively specify both knowledge states and their transitions, can be used as a powerful representation language in planning scenarios. Its variety of update commands can serve to model from the simplest condition-effect action to parallel actions and their indirect effects.

In this section we formalize the planning problem in LUPS, and characterize its solutions. Throughout we consider $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ to be an update program in the language $\mathcal{L}$. We begin by considering a set of actions from $\mathcal{L}$, whose specification is defined by update commands.

Definition 10 (Action) *Let $\mathcal{L}_\alpha = \{\alpha_1,\ldots,\alpha_m\}$ be a set of atoms from $\mathcal{L}$ where each $\alpha \in \mathcal{L}_\alpha$ represents an action. We call the elements of $\mathcal{L}_\alpha$* actions. *Typically for every* action *there will be one (or more) update commands of the forms (2)-(6), where the action α appears in the 'when' clause.* ◇

Fo example, in

$$always\ [event]\ (L \leftarrow L_1, \ldots, L_k) \\ when\ (L_{k+1}, \ldots, L_m, \alpha)$$

we have, intuitively, that α is an action whose preconditions are $L_{k+1}, \ldots, L_m$ and whose effect is an update that, according to its type, can model different kinds of actions, all in one unified framework. Examples of kinds of actions are:

- actions of the form "α causes F if $F_1, \ldots, F_k$", where $F, F_1, \ldots, F_k$ are fluents (such as in the language $\mathcal{A}$ of Gelfond and Lifschitz (1998)) translates into the update command "$always\ F\ when\ (F_1, \ldots, F_k, \alpha)$".
- actions whose epistemic effect is a rule update of the form "α updates with $L \leftarrow L_1, \ldots, L_k$ if $L_{k+1}, \ldots, L_m$" translates into the update command "$always\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m, \alpha)$".
- actions that, when performed in parallel, have different outcomes, of the form "α_a or α_b cause L_1 if $L_{k+1}, \ldots, L_m$" and "α_a and α_b in parallel cause L_2 if $L_{k+1}, \ldots, L_m$" translates into the three update commands:

  $$always\ L_1\ when\ (L_{k+1}, \ldots, L_m, \alpha_a, not\ \alpha_b)$$
  $$always\ L_1\ when\ (L_{k+1}, \ldots, L_m, not\ \alpha_a, \alpha_b)$$
  $$always\ L_2\ when\ (L_{k+1}, \ldots, L_m, \alpha_a, \alpha_b)$$

- actions with non-deterministic effects of the form "α causes (L_1 or L_2) if $L_{k+1}, \ldots, L_m$" translates into the update commands (where β_a, β_b are new auxiliary predicates and I is a unique identifier of an occurrence of α):

  $$always\ (L_1 \leftarrow \beta_a(I)) \\ when\ (L_{k+1}, \ldots, L_m, \alpha(I))$$
  $$always\ (L_2 \leftarrow \beta_b(I)) \\ when\ (L_{k+1}, \ldots, L_m, \alpha(I))$$
  $$always\ (\beta_a(I) \leftarrow not\ \beta_b(I)) \\ when\ (L_{k+1}, \ldots, L_m, \alpha(I))$$
  $$always\ (\beta_b(I) \leftarrow not\ \beta_a(I)) \\ when\ (L_{k+1}, \ldots, L_m, \alpha(I))$$

  In this representation of the non-deterministic effects of an action α, we create two auxiliary actions (β_a, β_b) with deterministic effects, and make the effect of α be the non-deterministic choice between actions β_a or β_b.

Next, we formalise *action updates* and *plans*:

Definition 11 (Action Update) *An* action update, U^α, *is a set of update commands of the form:*

$$U^\alpha = \{assert\ event\ \alpha : \alpha \in \mathcal{L}_\alpha\} \quad \diamond$$

Intuitively, each command of the form "$assert\ event\ \alpha$" will represent the performing of action α. Note that performing an action is something that does not persist by inertia. Thus, according to the description of LUPS in Sect. 2 the assertion must be of an event. By asserting $event\ \alpha$, the effect of the action will be enforced if the preconditions are met. Each *action update* represents a set of simultaneous actions. For simplicity, we represent the update commands in an action update just by their corresponding $\alpha's$. Instead of $U^\alpha = \{assert\ event\ \alpha_1; assert\ event\ \alpha_2\}$ we write $U^\alpha = \{\alpha_1; \alpha_2\}$.

Definition 12 (Plan) *A* plan *is a finite sequence of action updates.* $\diamond$

In order to relate the goals to be achieved with the plans, we need to know the effect of executing the plan. This is given by the following function:

Definition 13 (Result) *The* result *of applying a plan $\mathcal{U}^\alpha = U_1^\alpha, \ldots, U_m^\alpha$ to an update program $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ is given by the update program:*

$$Result(\mathcal{U}^\alpha, \mathcal{U}) = U_1 \otimes \cdots \otimes U_n \otimes U_1^\alpha \otimes \cdots \otimes U_m^\alpha \quad \diamond$$

Finally, the planning problem is about finding a plan such that a goal, given in the form of a LUPS query, is achieved as the result of applying the plan to the initial state update program.

Definition 14 (Planning Solution) *Given an update program $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ and a query (goal) "$holds(L_1, \ldots, L_k)$ at q?", the plan $\mathcal{U}^\alpha = U_{n+1}^\alpha, \ldots, U_q^\alpha$ is a* planning solution *if the query is true in the result of applying $\mathcal{U}^\alpha$ to $\mathcal{U}$, i.e. such that*

$$\bigoplus_q \Upsilon(Result(\mathcal{U}^\alpha, \mathcal{U})) \models_{sm} L_1, \ldots, L_n. \quad \diamond$$

This definition suggests that planning solutions can be seen as abductive update solutions in the LUPS framework. This will be explored in the next section.

4 On the implementation of LUPS and planning

In Alferes et al. (1999), a translation is presented of update programs and queries into single normal logic programs which are written in a meta-language. The translation is purely syntactic, and has been proven correct there: a query holds in an update program iff its translation belongs to all stable models of the update program translation. The latter directly supports a mechanism for implementing update programs: after a pre-processor performs the translations, query answering is reduced to that over normal logic programs by means of a meta-interpreter.

The pre-processor and a meta-interpreter for answering queries have been implemented[1]

The translation uses a meta-language generated by the language of the update programs. For each objective atom A in the language of the update program, and each special propositional symbol $rule_{L \leftarrow Body}$ or $cancel_{L \leftarrow Body}$ (where these symbols are added to the language for each rule $L \leftarrow Body$ in the update program), the meta-language includes the following symbols: $A(s,t)$, $A^u(s,t)$, $\overline{A(s,t)}$, and $\overline{A^u(s,t)}$, where s and t range over the indexes of the update program. Intuitively, these new symbols mean, respectively: A is true at state s considering available all states up to t; A is true due to the update program at state s, considering all states up to t; A is false at state s considering all states up to t; *not* A is true due to the update program at state s, considering all states up to t.

Intuitively, the first indexical argument added to atoms stands from the update state at which the atom was been introduced via a rule. So, according to the transformation in Alferes et al. (1999), in non-persistent asserts, the first argument of atoms in the head of rules is instantiated with the index of the update state where the rule was asserted. In persistent asserts, the argument ranges over the indexes where the rule should be asserted (i.e. all those greater than the state where the corresponding *always* command is). The second indexical argument stands for the query state. Accordingly, when translating (non-event) asserts, the second argument of atoms in the head of rules ranges over all states greater than the one where the rule was asserted. For event asserts, the second argument is instantiated with the index of the update state where the event was asserted. This is so in order to guarantee that the event is only true when queried about in that state.

Inertia rules are added to allow for access to rules asserted in states before the state the query is posed. Such rules say that one way to prove L at state s with query state t, is by proving L at state $s-1$ with the same query state (unless its complement is proven at state s, thus *blocking* the inertia of L).

Literals in the body of asserted rules are translated such that both arguments are instantiated with the query state. This guarantees that body literals are always evaluated with respect to the query's state. Literals in the *when* clause have both arguments instantiated with the state immediately prior to that in which the rule was asserted. This guarantees that those literals are always evaluated considering that prior state as their query state. The complete formalization of the translation $Tr(\mathcal{U})$ can be found in Alferes et al. (1999).

Planning solutions, as defined in the previous section, are clearly similar to abductive solutions in the LUPS framework: when finding a plan, one is looking for sets of "assert event´´commands, standing for actions performed, that when added to the sequence derive the top query and are consistent (in the sense that a stable model exists). In this abductive setting, the abducibles are commands of the form *assert event* (α) where $\alpha \in \mathcal{L}_\alpha$. Given the above described translation of a sequence of update commands into a single normal logic program with inertia rules, this abduction problem in the LUPS setting, can easily be transformed into an abduction problem in normal logic programs. Simply translate the sequence of update commands into a single program, and define as abducibles those translation predicates arising from translating commands *assert event* (α) for every α, where α is an action. It is easily seen that finding abductive solutions for queries over the translated abductive logic program is tantamount to finding planning solutions in the original update program.

Theorem 1 (Planning as abduction) *Let $Tr(\mathcal{U})$ be the logic program obtained from the translation of an update program $\mathcal{U} = U_1 \otimes \ldots U_n$, and let $Ab = \{\alpha(i,i) : n < i \leq m+n, \alpha \in \mathcal{L}_\alpha\}$ be the set of abducibles. Given a subset A of Ab, let $\mathcal{U}^\alpha(A) = U_1^\alpha \otimes \ldots U_m^\alpha$ be such that assert event $(\alpha) \in U_i^\alpha$ iff $\alpha(i,i) \in A$.*

Then A is an abductive solution for $Tr(holds\ L_1, \ldots L_k\ at\ q)$ in the logic program $Tr(\mathcal{U})$ iff $\mathcal{U}^\alpha(A)$ is a planning solution for "holds $L_1, \ldots L_k$ at q" in $\mathcal{U}$. ◇

According to this theorem, to implement a plan generator in LUPS, all that needs doing is to implement the translation from LUPS programs to logic programs, and then to use an interpreter for abduction on top of the translated program. This is the basis of our implementation which, aside from the aforementioned preprocessor for the translation, employs an interpreter for the abduction procedure ABDUAL (Alferes et al., 1999). Note that multiple abductions at one state, ie parallel actions, can be generated.

5 Illustrative Example

In this section we present an example illustrating the ability of this framework to deal with dynamically changing rules. One domain where such dynamic behaviour of rules is essential is *Legal Reasoning*. In this domain new rules come into play while, at the same time, other rules cease to be valid. In countries with legal systems where laws are often changed, jurisprudence makes heavy use of the articles governing the application of law over time. The representation of, and reasoning about, such articles is non trivial, the one most important being the part dealing with transient situations. For example, when an event occurs after some new law has been approved, but hasn't yet taken effect. Different outcomes could be obtained

[1]The system, running under XSB-Prolog, a system with tabling, is available from:
`http://centria.di.fct.unl.pt/~jja/updates/`
Rather than stable models semantics, the well-founded semantics is used instead.

depending on the date of the trial. In such situations an agent, acting as a lawyer, would have to plan its course of action in complex situations due to the changing rules.

Consider a fictional situation where someone is conscripted if he is draftable and healthy. Moreover a person is draftable when he reaches a specific age. In this situation, if someone is conscripted and not incorporated (for example because he hides), he is cited for a crime and, if tried, goes to jail. Of course one cannot be tried while in hiding. Consider that a person is electable for office if electable previously and not in hiding. Moreover, the person ceases to be electable if ever been to jail. After some time, a new law is approved that renders one not conscripted if a conscientious objector. However, this law will only take effect after 20 days. How could John, electable, healthy, conscientious objector, that became of age 10 days after this new law has been enacted, avoid being incorporated, and remain electable for office in the future? This is an illustrative scenario easily expressible in LUPS. It translates into the update commands[2]:

$\mathbf{U}_1$:
$always\ draftable(X)\ when\ of\_age(X)$
$assert\ (conscripted(X) \leftarrow draftable(X),$
$\quad healthy(X))$
$always\ hiding(X)\ when\ hide(X)$
$always\ not\ hiding(X)\ when\ unhide(X)$
$assert\ (jail(X) \leftarrow trialed(X), cited(X))$
$always\ incorporated(X)$
$\quad when\ (conscripted(X), not\ hiding(X))$
$always\ cited(X)\ when\ (conscripted(X),$
$\quad not\ incorporated(X))$
$assert\ (trialed(X) \leftarrow cited(X), not\ hiding(X))$
$always\ electable(X)$
$\quad when\ (electable(X), not\ hiding(X))$
$always\ not\ electable(X)\ when\ jail(X)$
$assert\ objector(John)$
$assert\ healthy(John)$

$\mathbf{U}_{10}$:
$always\ (not\ conscripted(X) \leftarrow objector(X))$
$\quad when\ current(30)$

$\mathbf{U}_{20}$:
$assert\ of\_age(John)$

where $\mathcal{L}_\alpha = \{hide(X), unhide(X)\}$ is the set of possible actions. If we have the goal of never being incorporated and being electable after 30 , the desirable solution would be to perform the action $hide(John)$ at 20, and perform the action $unhide(John)$ after 30. Note that if John does not hide at 20, he will be cited for a crime, be trialed, sent to jail and thus would never be electable for office again. The goal is:

$$holds\ electable(John)\ at\ 31$$

would return an abductive solution that would correspond to the plan $\mathcal{U}^\alpha = U^\alpha_{20}, U^\alpha_{30}$, where:

$$U^\alpha_{20} = \{assert\ event(hide(John))\}$$
$$U^\alpha_{30} = \{assert\ event(unhide(John))\}$$

representing the desired solution.

[2]Where $current(S)$ is a built in predicate such that it is true iff the current time is S, which could be implemented in LUPS itself.

6 Conclusions and Future Work

The ability to deal with dynamic situations is one of the major features of our proposal, as it allows one to handle a new class of planning problems. In fact, extant planning frameworks do not easily encompass the description of worlds with dynamically changing rules. For instance, neither PDDL (McDermott et al., 1998) nor OCL (Liu and McCluskey, 2000) are capable of describing dynamic situations. Dynamic Logic Programming permits as well the representing of actions in the STRIPS- or ADL-style, utilized in these planners, with pre-conditions, effects, and conditional and logical operators. Additionally, it caters for simultaneous actions and, due to its expressiveness, it can model complex effects of actions. By using an explicit representation of the world, and of the actions available at each state, its history and attending change can itself be queried and reconstructed.

Above all, embedding planning into a logic programming framework with a precise declarative semantics, makes it amenable to integration with other, already developed, monotonic and non-monotonic knowledge representation and reasoning functionalities. Among these:

- Extensive declarative knowledge representation, comprising default and explicit negation
- Semantics (and implementation) for non-stratified knowledge
- Observance and updating of integrity constraints
- Knowledge rules updating, besides that of action rules updating
- Abductive reasoning, over and above the abduction of actions
- Inductive learning of knowledge and action rules
- Belief revision and contradiction removal
- Argumentation for collaboration and competition
- Preference semantics, combinable with updates
- Meta- and object-language combination through meta-interpreters, facilitating language extensions and execution control
- Model-based diagnosis of artifacts, via observations and actions on them
- Declarative debugging of logic programs representing knowledge bases

- Explanation generation
- Distribution with communication
- Agent architectures
- Test and tried implemented logic programming systems, tabled execution

Our ongoing research has promoted and achieved some of these integrative *desiderata*, and currently pursues a number of them. Such integrateable facilities pave the way for the building of complex rational agents employing sophisticated planning amongst themselves.

Acknowledgements

All authors were partially supported by PRAXIS XXI project MENTAL. J. A. Leite was partially supported by PRAXIS XXI scholarship no. BD/13514/97.

References

J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinski and T. C. Przymusinski. *Dynamic Logic Programming*. In A. Cohn, L. Schubert and S. Shapiro (eds.), Procs. of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, pages 98-109. Morgan Kaufmann, June 1998.

J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinski and T. C. Przymusinski. *Dynamic Updates of Non-Monotonic Knowledge Bases*. To appear in The Journal of Logic Programming, 2000.

J. J. Alferes, L. M. Pereira, H. Przymusinska and T. C. Przymusinski, *LUPS - a language for updating logic programs*. In M. Gelfond, N. Leone and G. Pfeifer (eds.), Procs. of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99), El Paso, Texas USA, pages 162-176, Springer-Verlag, LNAI 1730, 1999.

J. J. Alferes, L. M. Pereira, H. Przymusinska, T. C. Przymusinski and P. Quaresma, *Preliminary exploration on actions as updates*.In M. C. Meo and M. Vialres-Ferro (eds.), Procs. of the 1999 Joint Conference on Declarative Programming (AGP'98), L'Aquila, Italy, pages 259-271, September 1999.

J. J. Alferes, L. M. Pereira and T. Swift, *Well-founded Abduction via Tabled Dual Programs*. In Procs. of the 16th International Conference on Logic Programming, Las Cruces, New Mexico, Nov. 29 - Dec. 4, 1999.

M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi and F. Zini, *Logic Programming and Multi-Agent System: A Synergic Combination for Applications and Semantics*. In K. Apt, V. Marek, M. Truszczynski and D. S. Warren (eds.), The Logic Programming Paradigm - A 25-Year Perspective, pages 5-32, Springer 1999.

M. Gelfond and V. Lifschitz. *The stable model semantics for logic programming*. In R. Kowalski and K. A. Bowen. editors. 5th International Logic Programming Conference, pages 1070-1080. MIT Press, 1988.

M. Gelfond and V. Lifschitz. *Classsical negation in logic programs and disjunctive databases*. New Generation Computing, 9:365-385, 1991.

M. Gelfond and V. Lifschitz. *Action languages*. Linkoping Electronic Articles in Computer and information Science, 3(16), 1998.

N. Jennings, K. Sycara and M. Wooldridge. *A Roadmap of Agent Research and Development*. In Autonomous Agents and Multi-Agent Systems, 1, 275-306, Kluwer, 1998.

R. Kowalski and F. Sadri. *Towards a unified agent architecture that combines rationality with reactivity*. In D. Pedreschi and C. Zaniolo (eds), Logic in Databases, Intl. Workshop LID'96, pages 137-149, Springer-Verlag, LNAI 1154, 1996.

J. A. Leite and L. M. Pereira. *Generalizing updates: from models to programs*. In J.Dix, L.M. Pereira and T.C.Przymusinski (eds), Selected extended papers from the LPKR'97: ILPS'97 workshop on Logic Programming and Knowledge Representation, pages 224-246, Springer-Verlag, LNAI 1471, 1998.

D. Liu and L. McCluskey. *The Object Centered Language Manual-OCL$_2$*. University of Huddersfield. 2000.

D. McDermott et al. *PDDL - The Planning Domain Definition Language*. Yale University, 1998.

I. Niemelä and P. Simons. *Smodels - an implementation of the stable model and well-founded semantics for normal logic programs*. In Procs. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), pages 420-429, Springer, July 1997.

S. Rochefort, F. Sadri and F. Toni, editors, *Proceedings of the International Workshop on Multi-Agent Systems in Logic Programming*, Las Cruces, New Mexico, USA,1999. Available from `http://www.cs.sfu.ca/conf/MAS99`.

F. Sadri and F. Toni. *Computational Logic and Multiagent Systems: a Roadmap*, 1999. Available from `http://www.compulog.org`.

The XSB Group. *The XSB logic programming system, version 2.0*, 1999. Available from `http://www.cs.sunysb.edu/~sbprolog`.

HTN Knowledge and Action Planning with Incomplete Information

Ralph Becket
Microsoft Research Ltd,UK
rbeck@microsoft.com

Abstract

Classical planners typically construct linearisable plans where the execution of any step is independent of the results of any preceding step. This is only possible if the initial conditions as expressed to the planner are sufficient to support the construction of such a plan. Knowledge and action planning tackles the problem that arises when this assumption doesn't hold true. For example, if I want to call someone on the telephone, but I don't know their number, I first have to look it up in the telephone directory. What number I actually dial depends upon the outcome of the information gathering step. In the past, KAAP has been tackled in two ways. The first is to combine highly expressive modal epistemic logics with logics of action and the second is to interleave classical planning with execution of information gathering plans. The former, it seems, is simply too expressive a formalism for tractable automatic reasoning methods. The latter is philosophically unsatisfying and has major efficiency problems. This paper introduces the simple notion of explicitly reasoning about the connection between agent state at run-time and the state of the world. This idea is philosophically straightforward and fits nicely in classical HTN planning frameworks without compromising efficiency. Moreover, it supports the construction of plans featuring non-trivial control constructs such as conditional execution and various forms of iteration. The technique described here is powerful enough to solve a broad, interesting class of KAAP problems.

1 Introduction

Conventional planners solve problems by deriving plans composed of (partially) ordered sequences of primitive actions. When any linearisation of a plan is applied in a world consistent with the initial conditions described to the planner, then the world should end up in a state in which the goal obtains. Two characteristics of such plans are that (a) plans contain no control constructs other than ordering constraints on steps – they do not contain loops or branches – and (b) the primitive action steps have constant arguments – the outcome of an earlier step cannot affect the execution of a later step. Such plans are derivable only if the planner has access to sufficient information at plan-time. Knowledge and action planning, or KAAP, is the less well known branch of planning research concerned with plan generation when this assumption is *not* true. In particular, KAAP problems are typified by the requirement to perform some 'finding out' steps in the course of obtaining a solution.

To set the stage consider the two following problems:

1. I want to call Fred on the phone, but I don't know his number. The obvious solution is to consult the telephone directory, find Fred's phone number, and then dial that.
2. I want to decide whether a solution is an acid or an alkali. This time I get a piece of litmus paper, dip it in the solution, *observe* the colour change, *deduce* the nature of the solution, and then act appropriately.

In the first problem, the number I end up dialling depends upon the number I find listed for Fred in the telephone directory – the dialling action is necessarily dependent upon the outcome of the 'finding out' step. In the second problem, merely observing the colour change in the litmus paper doesn't answer my question; I also have to perform a deduction action. Moreover, deciding the truth of some proposition like this is fairly useless unless it affects how I proceed in future, so this solution forms the basis of a *branching* plan structure.

2 Previous Approaches

Previous work has tackled KAAP in three quite different ways, each of which has its problems. This section gives a brief overview of each.

2.1 Modal Epistemic Logic

Arguable the most common approach has been to combine a modal logic of knowledge (e.g. KD45) with a modal logic of action to provide a KAAP framework (Moore, 1980; Moore 1984). The resulting theory is then typically compiled down to some first order representation. (Approaches along similar lines include the linear planning work on decidability by Rayner and Janson (1989) and the quotational theory advanced by Haas (1986)). Figure 1 gives the axioms of KD45 for

single agent knowledge, which forms the basis of most epistemic logics (K(P) is used to denote the agent in question knowing that P).

The resulting logic is highly expressive and can be used to solve quite complex KAAP problems. However, there are a number of problems with it, including the assumption of logical omniscience (agents are aware of all the logical consequences of any explicitly represented knowledge they have), restrictions on quantification and, indeed, the whole basis for accepting KD45 as a logic of knowledge is open to question (Slaney, 1996).

Basis:	A	- A is a non-modal axiom
N:	K(A)	- A is an axiom of KD45
K:	$K(A \Rightarrow B) \Rightarrow (K(A) \Rightarrow K(B))$	
D:	$K(\neg A) \Rightarrow \neg K(A)$	
4:	$K(A) \Rightarrow K(K(A))$	
5:	$\neg K(A) \Rightarrow K(\neg K(A))$	

Figure 1: The axioms of KD45.

However, by far the biggest problem with this line of attack is that nobody has yet found a way to build tractable automatic reasoners for modal epistemic logics in planning domains.

2.2 Interleaving Action with Planning

An alternative approach, implemented in planners such as XII (Etzioni et al. 1996), is to interleave planning with action. In this scheme, the planner is intimately tied in with the executive agent. The planner keeps track of the completeness of its knowledge of the initial conditions.

The planner records K(P) if P is listed in the initial conditions. The planner records the decidability of P as KW(P) where $KW(P) \equiv K(P) \vee K(\neg P)$. The planner also keeps track of completeness, as $\forall x.\ KW(P(x))$, indicating that an entire class of propositions P(x) can be decided from the initial conditions. Existence information may also be recorded as $\exists x.\ K(P(x))$.

When an XII type planner considers a subgoal that can be neither supported by the existing partial plan nor decided from the initial conditions, it performs the following steps:

1. it suspends planning for the current problem;
2. it looks for a plan that will provide the planner with the required information;
3. it executes the new subplan;
4. it resumes planning for the original goal in possession of the required information.

There are a number of unfortunate drawbacks with this scheme. In the first place, information gathering plans are generated *and executed* without knowing whether this will lead to a successful plan for the top-level goal. Of course, in some situations one cannot avoid this tactic; however in many others this can lead to unnecessary expenditure of resources and, in the worst case, can *prevent* the planner from finding a solution for the top-level goal if the interleaved execution changes the world in such a way as to prevent success. Another criticism is that information gathering tasks have to be treated specially and are emphatically not like ordinary, physical actions. Philosophically, it's not clear that this distinction is necessary or desirable.

3.3 Sensory Graph Plan

SGP, or Sensory Graphplan (Weld et al., 1998), is a variant of Blum & Furst's (1995) Graphplan algorithm that deals with the KAAP problem by identifying the sets of possible worlds (i.e. sets of initial conditions) the agent might be in and running Graphplan separately on all of them. SGP has to find a plan that will succeed in all possible worlds, which it does by introducing the notion of sensory actions that allow the executing agent to prune some possible worlds from the set it has to consider at runtime. Sensory actions in SGP merely decide the truth of some propositional formula at the time they are executed. Plan extraction is more complex since execution of parts of the final plan may depend upon the outcome of sensory actions (some actions may only be necessary or even executable in certain possible worlds).

As Weld et al. admit, SGP is only capable of solving very small problems, albeit doing so fairly quickly – a significant improvement over previous contingency planners – but it has a number of severe limitations. In the first place, sensory actions are assumed to have no preconditions and so can always be executed. However, this simply isn't the case in most KAAP situations (e.g. I can't find out how many elephants there are in the office next door without committing to some active course of investigation). Secondly, in order for SGP to work it has to be possible to enumerate all the possible worlds that the agent might start off in, which means that SGP is restricted to finite propositional domains (this is a general criticism of Graphplanners). Finally, unless the degree of uncertainty in the initial state is tiny, SGP has to consider an enormous number of possible worlds. The upshot of all this is that SGP is not yet able to tackle real KAAP problems.

3 A New Approach

Attempts to solve the whole problem (or, at least, a very broad part of it) using highly expressive logics founder because it is currently not known how to make effective automated reasoners for them. The simpler,

more pragmatic approach of interleaving planning with execution is philosophically unsatisfying and also has serious efficiency issues in that unnecessary, and even harmful, activity can take place before a complete plan has been derived.

This section outlines an alternative method that treads a path between the extremes of earlier methods that sits nicely within the framework of classical planning, solves a large and interesting class of KAAP problems, and does not suffer from crippling efficiency issues or engage in premature physical activity.

The core of the idea is to assume that the executive agent has some internal state (e.g. a bank of hardware registers) that changes as a consequence of performing information gathering actions. For this state, and 'finding out' actions, to be of any use, the agent should also be able to make control decisions based upon its current state.

The connection between the contents of a 'register' (or some state component) in an agent and the truth of some proposition depends upon how those contents came to be there. For example, if register R contains some number x as a consequence of performing the look-up-Fred's-phone-number action then x must be some representation of Fred's phone number. If, on the other hand, x arrived there as a consequence of the observe-the-colour-of-the-litmus-paper action then x must be some representation of the colour of the litmus paper.

It should be noted that this approach was inspired by the work of Rosenschein (1985) who demonstrated that by identifying "the agent knows that P" with "the agent is in a state S and the agent being in state S implies that P", one can construct a framework that supports all the standard axioms of propositional epistemic logic[1]. The great advantage of this interpretation is that it has an obvious and natural connection to computing systems in general and planning systems in particular.
Adopting the scheme just described provides several new opportunities for planners and presents us with a number of new problems to overcome. The new opportunities include the ability to construct plans which exploit the information provided by 'finding out' actions, either by using that information directly (e.g. by dialling the phone number) or to make a decision ("shall I declare the solution to be acidic or alkaline?"). Most interestingly, it is possible to both construct plans that solve universally quantified goals using iteration, rather than expanding the quantified formula to its universal base (Weld, 1994) and planning for each instance separately, and construct plans that involve searching for something ("have I found the box with the bomb in it yet?").

The framework described in this paper has been implemented in a planner called Baldric, about which more detail can be found in (Becket, 1998).

4 Formalism

Since the underlying idea is so simple, this section will be short.

Let S be some component of the executive agent's state and let us write S(t) to denote the value of S at time t. We will take it as axiomatic that S is always defined and unique:

$$\forall t.\ \exists!\ x.\ S(t) = x$$

Now let A be some observation action that changes the value of S, and let do(A, t, t') denote A being carried out over the interval between t and t' (that is, the preconditions of A are required to hold at t and the effects of A are deemed to hold at time t').

4.1 Deciding Observations

We now have to consider two types of observation action. The first is that an action A *decides the truth* of some proposition P by setting the value of S to be some value k iff P is true:

$$do(A, t, t') \Rightarrow (S(t') = k \Leftrightarrow P)$$

4.2 Value Observations

The second type of observation action *observes the value* of some aspect of the world, such as the colour of some litmus paper. If P(x) denotes "the value of the aspect of the world in question has a value *represented by* x" then we have

$$do(A, t, t') \Rightarrow (\forall x.\ S(t') = x \Rightarrow P(x))$$

Note that value observations may impute 'bulk knowledge'. For instance, P(x) could denote "x is a representation of the set of conference attendees" and be a precondition for projecting out (representations of) members of that set via some other action.

And that's about it for the 'knowledge' part. The remainder of this paper is concerned with how such basic tools can be used to tackle more interesting problems and the obstacles that have to be overcome in order to do so.

[1] Note that this paper is only concerned with agent *knowledge* in the sense that if an agent knows that P then P is indeed true – issues of belief and incorrect belief are not considered.

5 Knowledge and Action Planning

This section presents a variation on hierarchical task network (HTN) planning (Erol et al. 1993) adapted for KAAP. Two novel characteristics of this framework are that (a) plans may include disjunction and (limited forms of) quantification and that (b) actions can have different 'determinisms' in the Mercury (Somogyi et al. 1995) sense that actions can legitimately 'fail' – failure, in this case, being detected by the executive agent's program interpreter and used to make control flow decisions.

5.1 Events, Actions, Tasks, Propositions and Fluents

Events (i.e. the start and end points of tasks) are labelled with the times at which they occur[2]. If event t occurs before event t' then we write $t < t'$.

If A is an action, then do(A, t, t') is the *task* of carrying out A starting at time t and finishing at time t'. It is a requirement that the start event precede the finish event:

$$do(A, t, t') \Rightarrow t < t'$$

Actions may be concrete or abstract. A concrete task can be directly translated into a program step that can be carried out by the executive agent. An abstract task must first be decomposed, by the planner, into an implementation consisting only of concrete tasks before it can be executed by the agent.

A plan is a logical formula positing the execution of some number of tasks, the achievement of some number of subgoals, and various constraints on, amongst other things, the order of execution of the tasks in the plan.

Propositions whose truth can vary over time are called fluents; fluents take an extra, final, argument indicating the time at which they hold true: if P(x) is a fluent then P(x, t) says that P(x) is true at time t whereas ¬P(x, t) asserts that P(x) does not hold at time t.

If a fluent P(x, t'') appears in a plan then it is a subgoal of that plan. The subgoal is said to be supported if the plan includes some task do(A, t, t') s.t.

$$do(A, t, t') \Rightarrow P(x, t')$$

and a causal link constraint link(P(x), t', t'') where

$$link(P(x), t', t'') \equiv t' < t'' \wedge \forall\, t.\ t' \leq t \leq t'' \Rightarrow P(x, t)$$

In other words, a subgoal is supported if it is established by some preceding task and that that effect is required to persist up to the point where the subgoal is required.

A plan is said to be complete if it contains no unsupported subgoals. A plan is said to be concrete if it contains no abstract tasks. The objective of the planner is to find a complete, concrete, consistent plan formula.

5.2 Goals, Abstract Tasks and Methods

Goals and abstract tasks are replaced in the planning process by methods that achieve or implement them respectively. Methods are described simply as implications:

$$P \Leftarrow \text{Method}$$

where P is a goal or an abstract task. The correctness of generated plans, of course, depends upon the correctness of the method axioms in the domain definition: for instance, causal links should not emanate from tasks that do not achieve the stated effect and tasks should not be included in methods without their preconditions being either supported or listed as subgoals.

Example: Looking up a Phone Number

Methods

talking_to('Fred', t) ⇐
    do(dial_no_in_reg, t1, t2) ∧
    know_phone_no_for('Fred', t1) ∧
    link(talking_to('Fred'), t2, t)

know_phone_no_for('Fred', t1) ⇐
    do(lookup_phone_no_for('Fred'), t3, t4) ∧
    link(know_phone_no_for('Fred'), t4, t1)

Starting Plan (Top-Level Goal)

talking_to('Fred', t)

Final, Expanded Plan

do(lookup_phone_no_for('Fred'), t3, t4) ∧
link(know_phone_no_for('Fred'), t4, t1) ∧
do(dial_no_in_reg, t1, t2) ∧
link(talking_to('Fred'), t2, t)

Figure 2: Looking up a phone number.

Say we are constructing a plan for an agent with a single register R, the value in R at time t being R(t). Let know_phone_no_for(x, t) denote "R(t) is a representation of x's phone number".

[2] Events in this paper are named t0, t1, t2, ... but the numerical suffixes imply no temporal ordering.

Our goal is for the agent to set up a telephone connection with Fred, so the initial plan is just

```
talking_to('Fred', t)
```

(unless otherwise stated, all variables in plans are deemed to be existentially quantified.) Figure 2 describes the methods for the lookup task, which deposit's a representation of a phone number in R, and the dial task which uses it to decide what number to dial.

5.3 Conditional Execution

By allowing plans to include disjunction and semideterministic actions – actions that may fail, usually if some run-time condition is not met – then we can construct plans that feature conditional execution.

The idea here is that the final plan will be turned into a program with two or more branches; which branch is actually executed by the agent is decided by semideterministic test actions at the start of each branch. In Prolog fashion, when a semideterministic task fails the executive agent's interpreter will backtrack to the choicepoint and try an alternative branch. If-then-else control flow, for instance, can be obtained using a two-armed disjunction, one arm of which is guarded by a test for the truth of the condition and the other of which is guarded by a test for its negation.

Example: Performing a Litmus Test

This time we want our agent to tell us whether a particular solution is acidic or alkaline; figure 3 shows the basic methods and final plan structure. The starting plan contains the disjunctive structure in this example, although there is no reason why disjunction should not be introduced through method expansion. Note that this example uses a deciding observation task, as opposed to a value observation task in the phone number problem.
Simple plan expansion treats each branch of the disjunction separately, so the same 'setting up' operations will appear in each branch in the first instance. After the expansion phase, the planner attempts to simplify the plan structure by 'factoring out' components common to all arms of a disjunction. Indeed, this is a necessary precursor to being able to extract a workable program for the agent to run – this aspect of the KAAP scheme presented here is discussed in more detail later in the section on pragmatics.

5.4 Searching Problems

Often, one has to search for something before being able to proceed. This can be done in two ways. If the set of candidates is known to the planner in advance then the planner can simply describe the searching problem as a large disjunction in which each candidate is tested before the appropriate action is taken. If, on the other hand, the planner does *not* know the candidate set at plan-time then it can attempt to find a solution to the problem by constructing an *iterative* plan in which each candidate is tested in turn until the required member is identified.

Methods

```
solution_is_acidic ⇐
      do(test(R(t3) = red), t3, t4) ∧
      know_colour_of_dipped_litmus(t3)

solution_is_alkaline ⇐
      do(test(R(t3) ≠ red), t3, t4) ∧
      know_colour_of_dipped_litmus(t3)

know_colour_of_dipped_litmus(t3) ⇐
      do(observe_colour_of_litmus, t5, t6) ∧
      litmus_dipped_in_solution(t5) ∧
      link(know_colour_of_dipped_litmus, t6, t3)

litmus_dipped_in_solution(t5) ⇐
      do(dip_litmus_in_solution, t7, t8) ∧
      link(litmus_dipped_in_solution, t8, t5)
```

Solution

```
      (solution_is_acidic ∧
       do(say("it's an acid!"), t1, t2))
∨
      (solution_is_alkaline ∧
       do(say("it's an alkali!"), t1, t2))
```

Fully Expanded and Factorised Plan

```
do(dip_litmus_in_solution, t7, t8) ∧
link(litmus_dipped_in_solution, t8, t5) ∧
do(observe_colour_of_litmus, t5, t6) ∧
link(know_colour_of_dipped_litmus, t6, t3) ∧
(     (do(test(R(t3) = red), t3, t4) ∧
       do(say("it's an acid!"), t1, t2))
∨     (do(test(R(t3) ≠ red), t3, t4) ∧
       do(say("it's an alkali!"), t1, t2))
)
```

Figure 3: Performing a litmus test.

Iterative plans of the repeat-until variety can be described using a combination of existentially quantified plan fragments, semideterministic tasks (for the termination test) and *non*deterministic tasks to enumerate the candidate set. A nondeterministic task is much like a nondeterministic predicate in Mercury or Prolog: it may 'succeed' an arbitrary number of times, each time assigning to a register, or some other part of the executive agent's state, a value representing a new member

of the candidate set. The outline structure of such plans is, therefore

$$\exists x.\ \text{generate}(R, t1, t2) \wedge \text{link}(R(t) = x, t2, t3) \wedge \text{test}(R, t3, t4)$$

where generate(...) is the nondeterministic component and test(...) the semideterministic termination condition.

Consider McDermott's bomb-in-the-toilet problem (McDermott, 1987): the agent is locked in a room with a single toilet and a number of packages, one of which contains a ticking bomb. The agent has to defuse the bomb by flushing it down the toilet. Unfortunately, the agent only gets one chance to solve the problem since flushing a package down the toilet blocks it up. Figure 4 gives the solution in the framework presented here.

$$\text{link}(\text{toilet\_is\_unblocked}, t0, t7) \wedge \exists x.\ \text{do}(\text{choose\_package}, t1, t2) \wedge \text{link}(\text{holding\_package}(x), t2, t3) \wedge \text{do}(\text{listen\_for\_ticking\_sound}(x), t3, t4) \wedge \text{link}(\text{know\_whether\_ticking}(x), t4, t5) \wedge \text{do}(\text{test}(R = \text{is\_ticking}), t5, t6) \wedge \text{link}(\text{is\_bomb}(x), t6, t7) \wedge \text{do}(\text{flush\_down\_toilet}(x), t7, t8)$$

Figure 4: Iterative solution to the bomb-in-the-toilet.

In the solution, choose_package is a nondeterministic action that causes the agent to pick up the packages, one at a time on each iteration. The listen_for_ticking_sound action sets the agent's R register depending upon whether it hears a ticking sound from the box currently being held. The test(...) action is semideterministic as before and a successful result, along with the final action, supports the top-level goal of flushing the bomb down the toilet (the initial conditions, labelled with time t0, are used to support the flushing action precondition that the toilet be free of blockages.)
The combination of the nondeterministic choosing task and the semideterministic test task is spotted by the planner's program extraction stage and encoded as some kind of repeat-until loop in the agent's execution language. Again, the planner has to take care over the pragmatics of the plan (see later) in order to derive a correct program from it.

5.5 Iteration over an Unknown Set

In a similar fashion, we can construct plans describing iteration over a set of objects, this time using universal quantification, nondeterministic actions and implication:

$$\forall x.\ \exists t1, t2, t3, t4.\ \text{generate}(R, t1, t2) \wedge \text{link}(R(t) = x, t2, t3) \Rightarrow \text{perform\_task\_on}(R, t3, t4)$$

Tasks in the consequent of the implication may take support from tasks in the antecedent, but tasks outside the implication may not.

The applications to KAAP arise when the set to be iterated over cannot be decided at plan-time – for example, when the information has to be obtained from some external database or other agent. Indeed, this structure is quite flexible: there is no reason why the generation step cannot also include nondeterministic tasks. This could be used, for example, to ask each attendee at a forthcoming conference whether they need accommodation arranged or not and to go ahead and arrange it for those answering in the affirmative.

6 Program Extraction

This section addresses a number of important points that were glossed over in the preceding sections.

The key point is that, in the context of the more expressive plan specification language presented here, program extraction is no longer a trivial issue. Classical plans are essentially simple conjunctive formulae specifying what tasks are to be carried out and a partial order on execution – program extraction merely involves producing a sequence of instruction corresponding to the tasks in the plan in some linearisation of the partial order.

In the presence of local quantification, disjunction, and semi- and nondeterministic actions, this approach simply will not work. The problem is that now a plan is a formula presenting *sufficient, but not necessary, conditions* for achieving the goal. The crux of the matter is that normal (i.e. deterministic, or classical) actions cannot be backtracked over since they change the state of the world, making the program step ordering problem much more complicated.

For example, consider the following plan:

$$(\text{do}(\text{test}(\text{is\_spy}(\text{'Fred'}), t1, t2) \wedge \text{do}(\text{shoot}(\text{'Fred'}), t3, t4)) \vee (\text{do}(\text{test}(\neg\text{is\_spy}(\text{'Fred'}), t1, t2) \wedge \text{do}(\text{ask\_questions}(\text{'Fred'}), t3, t4))$$

We want our agent to shoot Fred if he is the spy and to ask him some questions if he is not. However, there is no causal dependency between the tasks in either of the

branches, so *logically* the goal would be satisfied if the agent shoots Fred *and then* determines that he is, indeed, the spy. Unfortunately, if the agent gets it wrong and finds out that it should have picked the other branch then it's too late to ask Fred questions. (It might be argued that, in this case, the goal is underspecified; however, since more subtle versions of this situation arise in iterative plans as well it seems that some robust way of dealing with it is necessary.)

The program extraction problem, then, is to find some ordering of the various tasks such that backtracking (or whatever control mechanism is used in the target language) will not lead to a situation where an alternative course of action has been clobbered by the attempted execution of another.

6.1 Disjunctive Plans

First, some terminology. A plan is sound if it is complete, consistent and concrete. A plan is simple if the tasks specified therein are all deterministic. A plan is directly implementable if the corresponding program is 'easily' derived. Simple, linear plans are directly implementable using the obvious strategy.

Now, consider a disjunctive plan $\Gamma \vee \Delta$ where Γ decomposes into the sequence[3] A, Test, Γ' s.t. A is sound, linear and simple, Test is semideterministic and Γ' is sound and directly implementable. If Δ is directly implementable, then so is the whole plan – for example, as

A, if Test then Γ' else Δ

provided that A, Δ is also sound. Note that it may be necessary to factor out tasks common to Γ and Δ if they are not idempotent – see figure 3 for an example.

6.2 Repeat-Until Plans

A plan describing a repeat-until structure must be ordered so that it decomposes into the following form:

A, $\exists$ x. (Generate, B, Test, C)

where the components A, B and C are directly implementable, Generate is nondeterministic and Test is semideterministic. The plan as a whole is directly executable, then, if A, (Generate, B)+, Test, C is sound, taking Δ+ to mean an arbitrary sequence of one or more instances of Δ (recall that Test only affects the state of the world if it succeeds). This can be verified by checking that the propositions protected by causal links supporting the loop body also hold after execution of the loop body. In other words, it must be the case that

A, $\exists$ x. (Generate, B,
$\exists$ y. (Generate, B, Test, C)[y/x])

is a sound plan. One possible implementation[4] of Γ is

A, while(Generate) {B, if Test then {C, break}}

if we posit that nondeterministic tasks such as Generate are implemented by operations that *either* change the state (of the agent and/or its environment) and return true *or* change nothing and simply return false.

6.3 Universally Quantified Plans

Universally quantified plans are very similar to existentially quantified (repeat-until) plans, the only difference being that the loop body is executed for *all* members of the set enumerated by the Generate step. Hence a plan that decomposes as

A, $\forall$ x. (Generate $\Rightarrow$ B)

might be implemented as

A, while(Generate) {B}

assuming A and B to be directly implementable.

6.4 Pragmatics: the Real World

In many real-world situations, knowledge sources such as people and databases are typically incomplete, imperfect and can be expensive to consult. It is important for a practical planning application to take these issues into account: it may well be the case that consulting a complete knowledge source may have such a high expected cost that it is better to go to a cheaper, less reliable source. For example, say I want to get hold of an individual on the telephone, but I don't have sufficient information about them to find their number in the telephone directory. One method is to call every number in the directory in turn until I reach the person I'm after. However, this procedure, while guaranteed to succeed (under various simplifying assumptions) is likely to be prohibitively expensive. A far better plan is to first ask someone we *expect* to be able to furnish us with sufficient information to carry out a direct lookup in the phone book.

[3] The notation Γ, Δ denotes the plan composed of the conjunction of Γ and Δ where the tasks in Γ are constrained to occur before the tasks in Δ.

[4] This pseudo-code glosses over the complications of failure, since if no solution is found in the loop body then the whole plan should fail because the follow on part of the plan almost certainly assumes a successful search. If the Generate operation runs out of options, program execution should be aborted.

A proper examination of pragmatics in this framework is beyond the scope of this paper and, indeed, is still an open research topic. A more detailed discussion of the problem (and the rest of the material covered in this paper) can be found in Becket (1998).

7 Conclusion

A new style of KAAP has been described which suffers from few of the difficulties that previous methods have fallen victim to. In particular, only relatively minor extensions to standard HTN planning are required to support the construction of concrete plan specifications. Program extraction is the phase which incurs the bulk of the penalty paid for greater expressive power, in that there is now no longer a trivial relationship between plans and programs that implement them. Extraction involves factorisation where necessary in disjunctive subplans, pragmatic reordering of conjuncts to ensure that 'branch mispredictions' at runtime don't clobber recovery, and compilation to a more complex programming model involving explicit representation of the agent's state and non-linear control flow.

The great advantage of the method outlined in this paper is that it can be implemented efficiently, mainly using well understood technology, and that it is capable of solving a broad class of interesting and practical KAAP problems. An appealing aspect is that implementations do not need to consider possible worlds, quotational theories or manipulate modal formulae.

This work has been implemented in a planner called Baldric which is currently being rewritten with the intention of conducting KAAP experiments to improve the strategic and interactive intelligence of agents in computer games; it is hoped that this will give a better feel for whether the approach really is applicable to real-world problems and, if so, what holes are left to be plugged.

Acknowledgements

Thanks have to go to Manny Rayner at NASA and Sam Steel at Essex University for their valuable insight and advice; both have been a great help in developing this work.

References

Becket, R. *Efficient Knowledge and Action Planning in First Order Logic.* SRI Technical Note CRC075, available at http://www.cam.sri.com/tr/crc075/paper.ps.Z

Blum, A. & Furst, M. *Fast Planning Through Planning Graph Analysis.* In *Procedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1636-1642. Pub. Morgan Kaufmann. 1995.

Erol, K., Nau, D. & Hendler, J. *Toward a General Framework for Hierarchical Task Network Planning.* In *Papers from the 1993 AAAI Spring Symposium*, pages 20-23, AAAI Press. 1993.

Etzioni, O., Golden, K. & Weld, D. *Sound and Efficient Closed World Reasoning for Planning.* University of Washington Dept. of C.S. and Eng. technical report UW-CSE-95-02-02. 1996.

Haas, A.R. *A Syntactic Theory of Belief and Action*, Artificial Intelligence, 28:245-292. 1986.

McDermott, D. *A Critique of Pure Reason*, in *Computational Intelligence*, 3:151-160. 1987.

Moore, R. *Reasoning about Knowledge and Action*, SRI AI Centre Technical Note 191. 1980.

Moore, R. *A Formal Theory of Knowledge and Action*, SRI AI Centre technical note 320. 1984.

Rayner M. & Janson S. *Finding out = Achieving Decidability*, in *IJCAI Workshop on Knowledge, Perception and Planning*, Davis, Morgenstern & Sanders (eds.) 1989.

Rosenschein, S.J. *Formal Theories of Knowledge in AI and Robotics*, New Generation Computing, 3:345-357. 1985.

Slaney, J. *KD45 is not a Doxastic Logic*, CISR technical report TR-SRS-3-96, Australian National University. 1996.

Somogyi, Z., Henderson, F., Conway, T. *Mercury: an Efficient, Purely Declarative Logic Programming Language.* In *Proceedings of the Australian Computer Science Conference*, pages 499-512. 1995.

Weld, D.S. *An Introduction to Least Commitment Planning.* AI Magazine, 15(4):27-61. 1994.

Weld, D.S., Anderson C.R., Smith, D.E. *Extending Graphplan to Handle Uncertainty & Sensing Actions.* AAAI, 1998.

Merging Planning and Path Planning: On Agent's Behaviours in Situated Virtual Worlds

Marc CAVAZZA
School of Computing and Mathematics
University of Teesside
TS1 3BA
United Kingdom
M.O.Cavazza@tees.ac.uk

Éric JACOPIN
CREC-Saint-Cyr
Écoles de Coëtquidan
56380 Guer Cedex
France
ejacopin@acm.org

M. Shafie ABD LATIFF
EIMC Department
University of Bradford
BD7 1DP
United Kingdom
M.S.Abdlatif@bradford.ac.uk

1 Introduction

Intelligent agents that populate virtual environments should be able to perform autonomous tasks. This performance assumes some capabilities to carry out plans whose elementary operators affect the state of affairs in the virtual world. We here discuss the implication that the spatial nature of the tasks has on the kind of planning technologies that can be used to implement agents' behaviour. Following previous work in situated AI, we first introduce the notion of spatialised planning environments; we then investigate the conditions under which the spatialised planning problem can actually be reduced to a geometrical path planning problem coupled with simple action selection, for agents possessing complete knowledge of their environment but not controlled by intentional representations. Finally, we outline a resolution procedure for spatialised planning environments.

2 Related works

Many different planning techniques have been proposed for controlling agents in virtual environments. The common ground for this work is however that agents have to carry tasks of world object manipulation that depend on some spatial properties and configuration. The interaction of the agent takes place with its environment as a whole and with the specific objects that this environment includes. [5] describes a planning system for the humanoid agent Jack . Some of the features in ItPlanS make explicit reference to the theory of situated actions, while its incremental planning capabilities are compared with those of PRS [6]. [7] identify some specific problems in the control of an agent's behaviour, drawing a distinction between low-level actions (where the agent reacts to its environment) and high-level plans that the agent should carry. This has served as a basis for the development of the "Microsoft Agent" system. However, no theoretical framework is developed to bridge the gap between high-level and low-level actions. In [3] have described path planning for agent behaviour, but do not address high-level plans that would make use of such paths. Following previous work in situated AI [1], we have investigated the implications for plan formalization and resolution of the spatial nature of planning tasks [4].

3 Spatialised planning environments

Let us first introduce the notion of Spatialized Planning Environment (SPE). A SPE is made of a physical layout which is more than the reference world upon which the agent performs its plans. It can rather be said that the spatial relations between various objects acting as resources for the solution of a plan carry a direct meaning for the resolution of the plan itself, or actually encode part of the solution.

Furthermore, as described in the framework of situated AI, spatial environments have the unique property to act both as targets for operation and as a mean to store knowledge. The combination of these two properties can be illustrated by an analogy with blackboard techniques for problem-solving. Like in a blackboard, the knowledge source (resource) triggered by relevant data forms a Knowledge Source Activation Record awaiting its application by the control mechanism, the spatial proximity of a object to a specialized tool creates the local conditions for reactive object manipulation. As transformations performed on the object also have a spatial translation, they can affect both the search space and the further application of operators.

For instance, the domestic appliances in the kitchen [4], can all be associated different meanings according to their spatial arrangements relative to the object upon which they are supposed to operate as the plan unfolds. SPEs are more than theoretical constructions or artifacts resulting from an ethnomethodological perspective. Actually, most of modern computer games can be described as SPEs, in which there is a direct mapping between actions to be carried and the spatial disposition of tools or weapons that are required to perform the elementary actions of which the plan solving the game level is to be made. There is significant evidence of this through the notion of "ideal path" to solve a game level, although this is specific to adventure/Role Playing Games. In recent computer games, the notion of objective (or sub-goal) becomes explicit. The collection of tools as preconditions for specific actions is also made apparent. The fact that a game level has a specific solution that consists in performing the correct actions at given stages in time and space is of course a direct evocation of planning technologies. The solution plans appear for instance as part of the textual solutions (also called "spoilers"), which detail the set of actions to be taken together with their proper justifications. They are just a textual description of a plan from which the plan can be formalised more traditionally, e.g. through and AND/OR graph. However, the same solution is often presented under the form of an "ideal path" joining various landmarks that correspond to objects to be collected or problems to be solved. This is only possible because of the spatial nature of part of the application semantics.

We can now introduce a basic framework for agents in SPEs. The solution of a plan can be seen as the combination of path planning and local reactive actions that are fully determined by the local preconditions of operators. The specific locations at which operators are to be actions behave as landmarks in the traversal of the environment. These can be set as part of the design of the SPE. For example, the design of video- games involves a purposeful spatial arrangement of objects bearing a specific meaning for the player; many games, especially in the adventure genre, can actually be reduced to the arrangement of tools and reactive objects on which to operate. The hypothesis behind our work is that landmarks serve the programming of agents in virtual worlds: they thus are part of the resources of plans in the tradition of situated action [1, 10]. When basing plan resolution on path planning, the essential step is to determine which landmark should be visited next.

This also suggests the important conclusion that SPEs require an initial searching step, or "priming" step, which is crucial to finding an appropriate solution. In the context of spatialised tasks, this is a consequence of the fact that path planning is a monotonous process, hardly accommodating backtracking: the search space for the first action should be small enough to allow for an exhaustive search. However, this approach should be consistent with the general set of incremental planning techniques[6, 5].

We have seen that in SPEs the solution could be amenable to geometrical path planning, which consists in finding an optimal path from a source position to a goal position along which the agent will move in the virtual environment. This is a well-described problem in robotics, with the important difference that the virtual agent can be conferred an absolute knowledge of its environment and does not have to rely on sensor data to establish the correct path in real time. Stated in these terms, path planning can be implemented through traditional search techniques. Several of these can be used, but A* provides acceptable solutions for static environments, i.e. environments in which the goal does not change during the search process itself. Also, because the operators are applied one at a time and the results of the local reactive actions taken might impact on subsequent events, the path planning process only plans from one landmark to the next one.

Identification of the next landmark thus is a crucial step which can be achieved through various techniques: (i) proximity heuristics and (ii) domain dependent heuristics prioritizing landmark transitions.

The solution to a SPE can thus be represented by the following algorithm:

```
LandMark ← GetFirst();
While LandMark ≠ Goal Do
        LandMark ← FindNext();
        Plan_path_to(LandMark);
        Action ← GetReactiveAction(LandMark);
        Area ← GetBoundaries(LandMark);
        Perform(Action, Within(Area));
End while
```

In some cases the environment can provide a hint (proximity to the starting point). However, in the

general case a specific search procedure is need. This might be similar to the notion of depth-bounded search experts introduced by [5].

We wish to illustrate the notion of SPEs a little further with discussion of the DigitVille problem [9, page 160-161]:

> (Opportunistic search). The figure 1 shows a map of the hypothetical city of DigitVille, used in this exercise to illustrate aspects of the task of errand planning. Suppose that Barbara works at General Heuristics and lives in the Ocean View Apartments. On the way home from work Barbara intends to carry out several errands: getting groceries, picking up the kids at the day-care center, picking up some clothes that are ready at the dry cleaner's, and perhaps picking up some treats for company that evening. For simplicity, assume that each task takes 10 minutes to perform, that parking can only take place at designated parking lots and takes 2 minutes, that driving from point to point takes 2 minutes per block, that walking takes 3 minutes per block, and that Barbara needs to be home 75 minutes afters she leaves work. Also assume that walking a partial block or crossing a street takes no time at all and that the time to "unpark" is included in the 2 minutes for parking. Other constraints are that groceries are too heavy to carry farther than the grocery-store parking lot and that the young kids will fuss too much if they are taken on more than one errand.
>
> (This exercise was inspired by one used by Barbara and Frederick Hayes-Roth in testing cognitive models of planning by human subjects. It is a variation of the travelling salesman problem. This exercise is intended to promote discussion on human strategies of search for a familiar task.)
>
> (a) Find a route home for Barbara that enables her to accomplish all of her primary tasks and as many of her secondary tasks as possible. The primary tasks are picking up the kids at day care, getting groceries, and picking up a prescription for the kids at the drugstore. The secondary tasks are picking up some forms at vehicle registration, getting cash at the bank, getting a widget at the hardware store, picking up some pastries at the Elaborate Pastry Shop, and picking up the clothing at the dry cleaner's. Present your solution showing the elapsed trip time after each leg of the trip.

The important point here is the relation between the primary and secondary tasks: none. Except that "kids [should not be] taken on more than one errand", whatever this errand is. Everything is compatible with everything: an AND/OR graph of the tasks would just consider everything possible (see figure 2). If we look at the map of DigitVille, we understand that only the distances between the locations where the tasks are to be performed and their duration are crucial to this problem. This is not the case in SPEs where the agent can move and perform tasks freely. In fact, Barbara may as well pick a better map of DigitVille and find the shopping center 3 where she could do everything she needs to, in the way she decides to: the map of figure 3 still is coherent with the graph of the tasks 2 but is much better for the performance of the tasks than the map of DigitVille1. However, we can push the freedom a little further again by distributing spatially the possible performance of the tasks: we place Barbara into the new DigitVille Hypermarket where you can find cash machines in several stores and where the grocery store possess an Elaborate Pastries section. Now, Barbara might as well get cash while choosing groceries or buying a widget at the hardware store. Getting cash clearly is a routine task and Barbara would prefer spending time thinking of the pastries than of the performance of the "Get money from the cash machine" task. Not only SPEs provide freedom in movements and in the order in which the tasks are performed, but they also allow routine performance of some tasks. Finally, in the hypermarket of DigitVille, Barbara shall apply her A* algorithm on the and-or graph to design a path according to this evening's mood, which can also be very dynamic (i.e. changing from one locations to another). This also illustrates the shopper's progress through a supermarket where the sight and then pick-up of a necessary item only locally alter the wandering of the shopper [8, pages 152–155].

4 Conclusions

We have introduced Spatialised Planning Environments which constitute a framework for virtual agents in which the spatial layout encodes part of the solution, where the resolution process can be based on path planning from one landmark position to another. While this framework is supported by practical evidence and previous studies in planning theory, we do not claim however, that this approach should be suitable for any kind of planning application. However we do claim it opens interesting directions for the simulation of autonomous behaviour, where part of the design of an experiment lies in the spatial distribution of world objects with their specific semantics. It would help exploring the behavioural consequences of spatial layouts by avoiding biases due to specific encoding of high-level be-

haviours in the agents themselves.

References

[1] AGRE, P. Computation and Human Experience, Cambridge University Press (1997), 371 pages.

[2] ALEXANDER, R. Construction of Optimal-Path Maps for Homogeneous-Cost- Region Path-Planning Problems, Ph.D. Thesis, Dept. of CS, US Naval Postgraduate School, Monterey, CA, U.S.A. (1989).

[3] BANDI, S. and THALMANN, D.. Space Discretization for Efficient Human Navigation, Computer Graphics Forum, 17(3): 195-206, (1998).

[4] CONEIN, B. and JACOPIN, É. Proceedings of the AAAI Fall'96 Symposium on Embodied Action, AAAI Press Report FS-96-02, (1996).

[5] GEIB, C. 1994. The intentional planning system: ItPlanS. In Proceedings of AIPS'94, AAAI Press, pages 55–60.

[6] INGRAND, F.-F., GEORGEFF, M. and RAO, A. An Architecture for Real-Time Reasoning and System Control, IEEE Expert 7(6): 33-44 (1992).

[7] KURLANDER, D. andLING, D., Planning-Based Control of Interface Animation, Proceedings of CHI'95.

[8] LAVE, J. Cognition in Practice, Cambridge University Press. 214 pages.

[9] STEFIK, M. Introduction to Knowledge Systems. Morgan Kaufmann. 870 pages.

[10] SUCHMAN, L. Plans and Situated Action: The problem of human/machine communication, Cambridge University Press (1987), 204 pages.

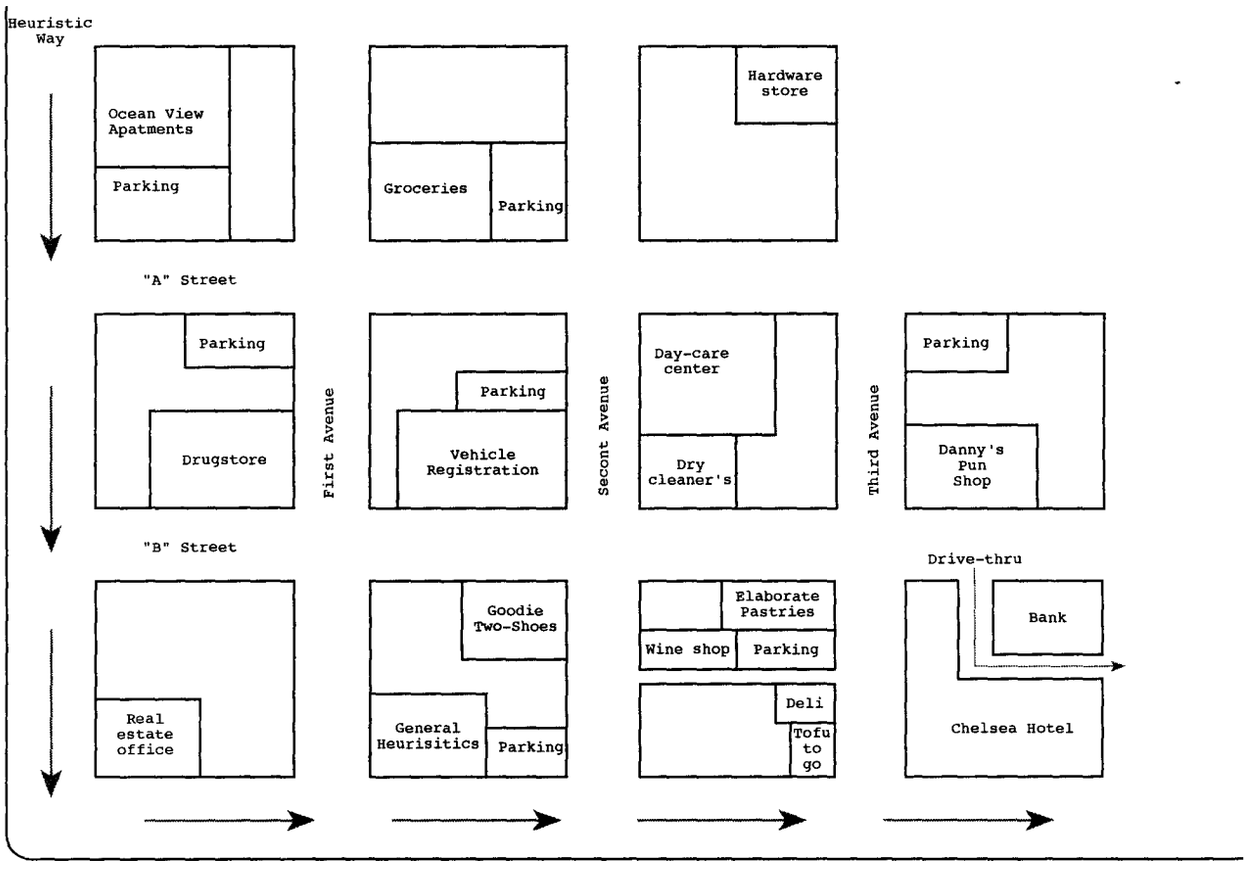

Figure 1: Map of DigitVille.

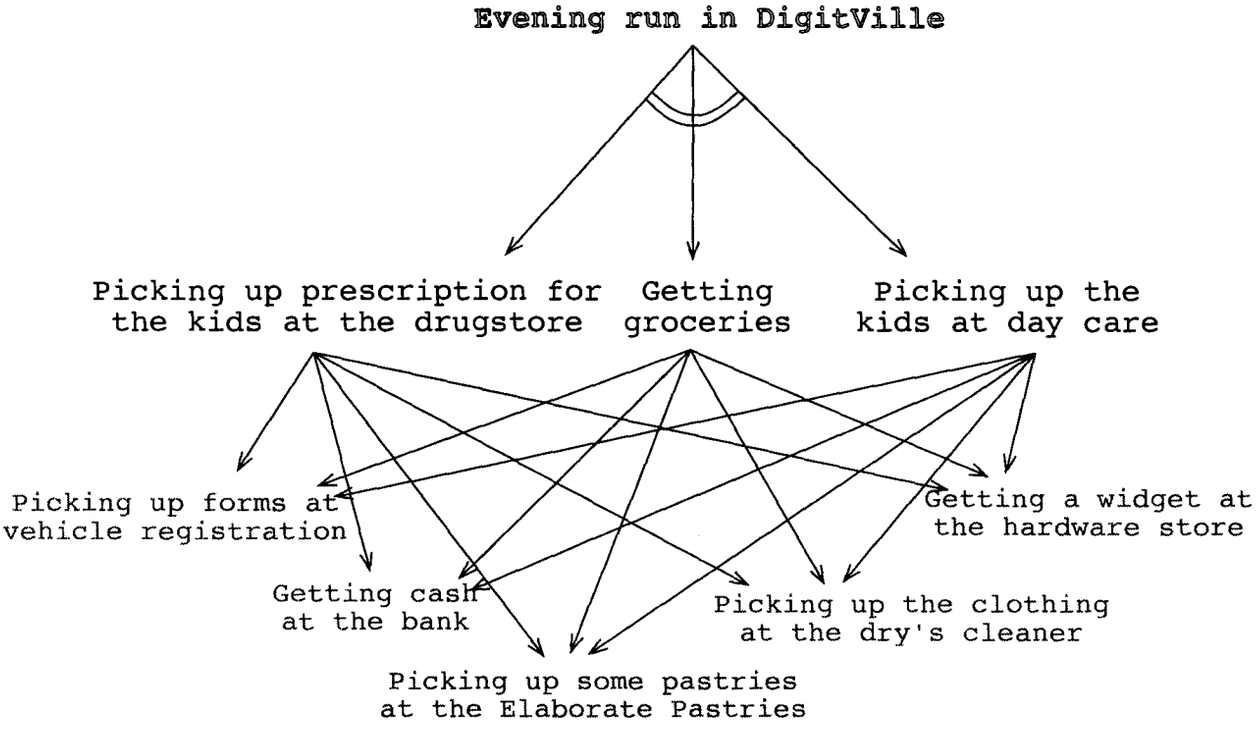

Figure 2: The and/or graph of Barbara's primary and secondary tasks this evening in DigitVille.

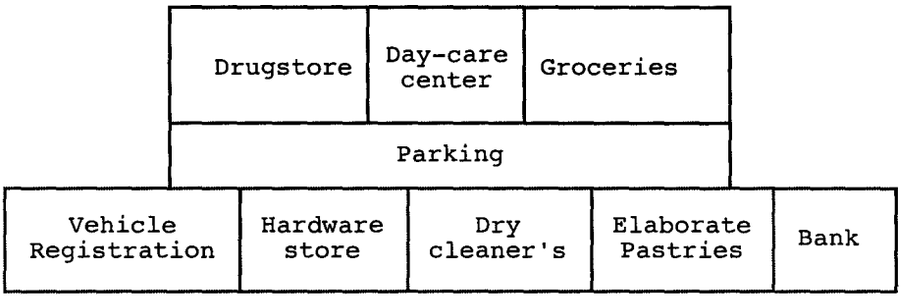

Figure 3: DigitVille as a Shopping center.

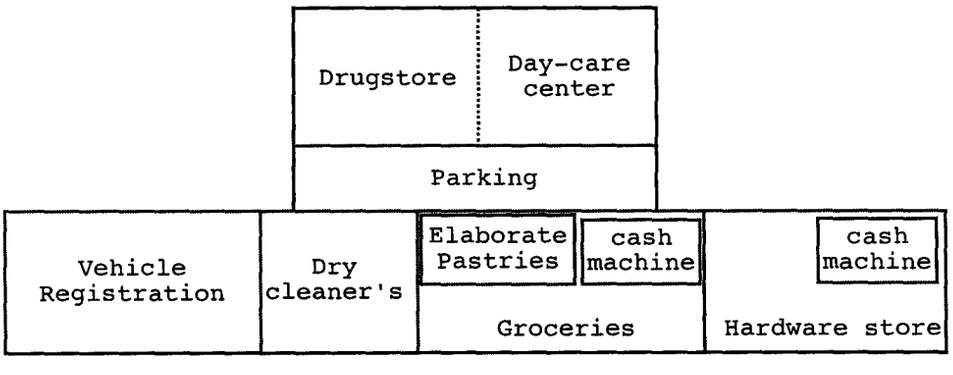

Figure 4: DigitVille as a Hypermarket.

Scheduling for an Uncertain Future with Branching Constraint Satisfaction Problems

David W. Fowler; Ken Brown
Department of Computing Science, University of Aberdeen
dfowler@csd.abdn.ac.uk; kbrown@csd.abdn.ac.uk

Abstract

Agents which schedule tasks must deal with problems that change as time progresses, often while the problem is being solved. In this paper, we assume that the agent does not blindly react to events, but uses knowledge of likely changes to make decisions in the present that will facilitate future actions. To solve this problem, we introduce the branching CSP, a new variant of the constraint satisfaction problem, along with two algorithms for solving it. Some experiments are presented, showing that constraint propagation improves search efficiency.

1 Introduction

In this paper, we consider the problem of an agent that periodically receives requests for action, and must make a decision for each action as it is received. The agent also has a simple model of what requests are likely to occur. We believe that this knowledge can enable the agent to make decisions that will better facilitate its future actions.

The approach used here involves extending the framework of constraint satisfaction to include the model of future events to give a new form of constraint satisfaction problem, (or CSP) (Kumar, 1992; Tsang, 1993), called a *branching CSP* (BCSP).

A related area is that of dynamic constraint satisfaction (Dechter and Dechter, 1988; Miguel and Shen, 1999). This approach models a dynamic environment as a series of constraint satisfaction problems, but does not include the model of future events that we use here.

Wallace and Freuder (1997) do consider future events in problems which can change over time, which they call recurrent CSPs. However, the changes are frequently recurring and temporary deviations from a normal state of affairs, and the aim is to find solutions that require as little adjustment as possible.

Fargier et al. (1995) examine constraint satisfaction problems where knowledge of the world is uncertain. The main difference with the work described here is that we examine problems where a sequence of decisions must be planned in advance, instead of just a single decision.

We have concentrated here on problems that involve hard constraints that can not be violated, and with variables that can be left uninstantiated (at a cost). Leaving a variable uninstantiated corresponds to denying a request. Other approaches, such as fuzzy scheduling (Kerr and Slany, 1994), and partial constraint satisfaction (Freuder and Wallace, 1992) deal with constraints that can be violated (at a cost), but where all variables are instantiated. We intend to look at ways to combine these approaches in the future.

2 An Example Problem

To illustrate the type of problem we are considering, we will use a simple example. Consider a scheduling problem in which new tasks arrive during the process. The aim is to schedule as many tasks before their due date as possible. There are two identical resources that the tasks may use. The tasks for this problem are described in Table 1.

Task	*Duration*	*#Resources*	*Due*	*Utility*
A	2	1	4	1
B	2	1	4	1
C	3	1	4	3
D	1	2	4	8

Table 1: Tasks for Example Problem

We start with tasks A and B. We know that one of the other tasks will arrive at time 1. It will be C with probability 0.6, and D with probability 0.4. A final restriction (future work will look at how to overcome this) is that tasks A and B must be scheduled before it is known which of tasks C or D arrives next. How should we schedule tasks A and B?

It can be seen that there are three reasonable possibilities, shown in Figure 1. Solution (a) allows task C to be scheduled, but not task D, whereas the solution (b) allows task D but not task C. Solution (c) allows both of tasks C and D to be scheduled, at the price of omitting task B. Which solution is best depends on the probabilities of the tasks, as well as their utilities.

For this example, the three solutions have expected utilities of:

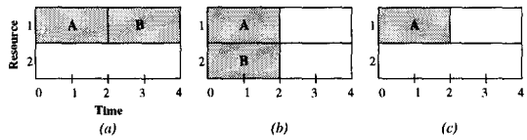

Figure 1: Two Possible Solutions

(a) $1 + 1 + (0.6 \times 3) + (0.4 \times 0) = 3.8$

(b) $1 + 1 + (0.6 \times 0) + (0.4 \times 8) = 5.2$

(c) $1 + 0 + (0.6 \times 3) + (0.4 \times 8) = 6.0$

It is therefore best to schedule task A only, as this gives the highest expected utility.

We will now present a method to represent and solve this type of problem.

3 Branching CSPs

The initial definition of a Branching CSP involves the following components: variables, constraints, and a state tree.

The set of variables can be represented as:

$$X = \{X_1, \ldots, X_n\}$$

Each variable X_i has an associated finite domain of values D_i, and a non-negative utility value u_i. It is possible in this formulation for a variable to be left unassigned. In which case the utility gained from that variable will be 0. We have used the standard notion of utility from decision theory (see, for example, French (1986) or Lewis (1997)), with the difference that the utilities are non-negative.

A constraint is a restriction on the values that a subset of the variables can simultaneously take. For the purposes of definition, the set of constraints can be represented as:

$$C = \{C_1, \ldots, C_m\}$$

where each constraint C_j is a pair (J_j, P_j); J_j being an ordered subset of X: $\langle X_{j1}, X_{j2}, \ldots, X_{jt}\rangle$, and P_j being a subset of $D_{j1} \times D_{j2} \times \ldots \times D_{jt}$.

The state tree represents the possible development paths of the dynamic problem. Each edge in the tree is directed, and is labelled with a transition probability. Each node contains a variable from the set X, with the restriction that a variable can appear at most once in any path from the root node to a terminal node.

For our definition we have a set of states

$$S = \{S_1, \ldots, S_n\}$$

with each state S_i having an associated variable X_{S_i}. There are transition probabilities p_{ij} labelling the edge (if it exists) between S_i and S_j.

For any path through the state tree from root to terminal node, a series of constraint satisfaction problems is produced, involving all variables that have been encountered at each node in the path so far. At each node, a value must be found for the corresponding variable, or the variable may be left unassigned. If a constraint involves variables that are all assigned values, then those values must satisfy the constraint.

A solution to the overall problem is an assignment of a value to each variable (alternatively leaving the variable unassigned) at each node in the state tree, so that at each state all relevant constraints (those that involve only variables that have been assigned values in the path from the root to the node in question) are satisfied.

A solution represents a plan for each possible sequence of variable additions, and we have assumed that the total utility of the solution can be found by summing the utilities of the assigned variables in the path that actually occurs. However, we must try to find a solution before we know which path will actually occur, and so we define the optimal solution to be the one with the highest expected total utility. Note that there are other notions of optimality, such as minimal regret, but we have concentrated on maximising the expected utility.

The expected utility from a node can be defined recursively as follows. The E.U. from a terminal node S_i is:

$$\begin{cases} u_i & \text{if } X_{S_i} \text{ is assigned} \\ 0 & \text{if } X_{S_i} \text{ is unassigned} \end{cases}$$

For a nonterminal node S_i, the E.U. is:

$$\sum_j p_{ij} EU_j + \begin{cases} u_i & \text{if } X_{S_i} \text{ is assigned} \\ 0 & \text{if } X_{S_i} \text{ is unassigned} \end{cases}$$

where the sum is over all the child nodes of S_i.

For our example problem, the variables will be the four tasks A, B, C and D. Their domains will be the possible locations on the time/resource diagram. There will be six pairwise constraints between the tasks to ensure that they do not overlap. The state tree is shown graphically in Figure 2.

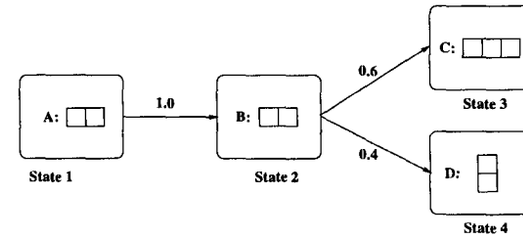

Figure 2: The State Tree

It would be possible to represent our model as a 'game against Nature', where the player's moves are the possible assignments to the variables, and Nature's moves are the different branches of the tree in Figure 2. However, we assume here that the choices of variable assignments do not affect which paths will be taken through the tree, and therefore our simpler model is sufficient.

4 Solution Algorithms

In this section two complete algorithms, depth first search and forward checking search, are described briefly. These have been implemented in C++, and the results of some preliminary experiments are given in the next section.

4.1 Depth first search

The depth first algorithm finds the maximum expected utility for the problem by trying all possible values for the variable in the initial state, and for each value recursively finding the maximum expected utility for each of the children states of the initial state. At each state, only those values consistent with the assignments previously made are considered.

A branch and bound approach can be used here: as there may well be more than one optimum solution, it is best to stop as soon as the first is found. This is achieved by calculating at each state an upper bound on the expected utility from that state. This can be done by relaxing all the constraints and assuming that each variable can be assigned a value. During search, if any assignment to a variable attains this upper bound, no more values for the variable need be tried.

The algorithm is given in Figure 3. It returns the maximum utility achievable from a given node, and also returns the set of assignments to achieve this utility.

4.2 Forward checking search

The forward checking algorithm (Figure 4) uses constraint propagation to reduce the domains of variables lower down in the tree, by eliminating values that conflict with the current variable assignment, in an analogous way to that of forward checking in standard constraint satisfaction (Haralick and Elliott, 1980).

It also uses the idea of branch and bound, in a more advanced form, as the algorithm that performs the propagation of constraints also returns a new upper bound on the expected utility that can now be achieved from the current state. If this bound is lower than a previously found value, then the branch of the tree can be pruned immediately.

The forward checking algorithm uses constraint propagation to reduce the domains of variables lower down in the tree. The function `propagate` performs this reduction, and returns an upper bound on the expected utility that can now be achieved. If this bound is lower than an existing solution then the propagation can be undone (using `retract`), and the next value tried for the current variable immediately.

```
DF_Search(input S_i:node,
          output bestTEU:float,
          output bestPlan:set of ⟨S_i, x_i⟩)
Local vars:
x:domain type of X variables
TEU:float // total expected utility for S_i
plan, result: set of ⟨S_i, x_i⟩
stop:boolean

bestTEU := 0;
for x in domain(X_{S_i}) do
   begin
   plan := ∅;
   TEU := utility_of(X_{S_i});
   if ⟨X_{S_i}, x⟩ and current assignments satisfy C then
     begin
     for S_j in children_of(S_i) do // recursively calculate TEU
         begin
         result := DF_Search(S_j);
         TEU := TEU + (result.utility × prob(S_i → S_j));
         plan := plan ∪ result.bestPlan;
         end;
     if TEU > bestTEU then
           begin
           bestTEU := TEU;
           bestPlan := plan ∪{⟨X_{S_i}, x⟩};
           end;
     if TEU = maxUtility(S_i) then
           begin // stop if no better solution can exist
           stop := true;
           break;
           end;
     end;
   end;

if not(stop) then
  begin // try leaving X_{S_i} unassigned
  plan := ∅;
  TEU := 0;
  for S_j in children_of(S_i) do
      begin
      result := DF_Search(S_j);
      TEU := TEU + (result.utility × prob(S_i → S_j));
      plan := plan ∪ result.bestPlan;
      end;
  if TEU > bestTEU then
        begin
        bestTEU := TEU;
        bestPlan := plan ∪{⟨X_{S_i}, ⊥⟩};
        end;
  end;
return bestTEU, bestPlan;
```

Figure 3: Depth First Algorithm

5 Experiments

To test and compare the performances of the algorithms, they were both used to solve the same range of randomly produced problems, and the number of constraint checks examined. There are a large number of parameters that can be altered; for these initial experiments most parameters were fixed. These parameters and their values are:

```
FC_Search(input S_i:node,
          input futureVars:set of X_i,
          output bestTEU:float,
          output bestPlan:set of (S_i, x_i))
Local vars:
x:domain type of X variables
TEU:float
plan, result: set of (S_i, x_i)
stop:boolean

bestTEU := 0;
futureVars := futureVars \ X_{S_i};
for x in X_{S_i} do
   begin
   plan := ∅;
   TEU := utility_of(X_{S_i});
   if propagate(⟨X_{S_i}, x⟩,futureVars,S_i) < bestTEU then
      begin
      retract(⟨X_{S_i}, x⟩,futureVars);
      continue; // Continue with next value for x
      end;
   for S_j in children_of(S_i) do
      begin
      result := FC_Search(S_j,futureVars);
      TEU := TEU + (result.utility × prob(S_i → S_j));
      plan := plan ∪ result.bestPlan;
      end;
   if TEU > bestTEU then
         begin
         bestTEU := TEU;
         bestPlan := plan ∪{⟨X_{S_i}, x⟩};
         end;
   retract(⟨X_{S_i}, x⟩,futureVars);
   if TEU = maxUtility(S_i) then
         begin
         stop := true;
         break;
         end;
   end;
if not(stop) then
  begin
  plan := ∅;
  TEU := 0;
  for S_j in children_of(S_i) do
      begin
      result := FC_Search(S_j,futureVars);
      TEU := TEU + (result.utility × prob(S_i → S_j));
      end;
  if TEU > bestTEU then
         begin
         bestTEU := TEU;
         bestPlan := plan ∪{⟨X_{S_i}, ⊥⟩};
         end;
  end;
futureVars := futureVars ∪X_{S_i};
return bestTEU, bestPlan;
```

Figure 4: Forward Checking Algorithm

- The number n of variables (10);
- The size m of the domains (10);
- The maximum utility value for any variable (50). Each utility value is an integer selected uniformly from the range [1,50];
- The maximum depth of the state tree (8). Each path from the initial state to a terminal state can have a maximum of 8 states. The tree is generated according to a branching process (described below) which is terminated if it reaches the maximum depth.

The parameters which were varied are:

- The proportion p_1 of constraints that exist (out of the total possible, $n(n-1)/2$) between a pairs of variables (only binary constraints were used for these tests). p_1 is also called the density of the constraint graph, and for low values of p_1 there is a high probability that the constraint graph will be disconnected. p_1 was varied from 0 to 1 in steps of 0.2;
- The proportion p_2 of tuples that are *disallowed* in any constraint (this is known as the tightness of the constraint). There will be $p_2 m^2$ such tuples in each constraint. This parameter was also varied from 0 to 1 in steps of 0.2;
- The branching process that produces the state tree. A distribution gives the probability that a state will have 0,1,2 etc. children states. Two distributions were used, as shown in Table 2.

Number of children	0	1	2	3
Dist1	0.05	0.55	0.80	1.00
Dist2	0.05	0.85	1.00	

Table 2: Probability Distributions

The transition probabilities from each state were selected as follows: a total value of 1 was shared out between each child state, with each child receiving a random probability selected uniformly from 0 to whatever value was left to be shared; the last child receiving the remainder.

Note that the parameters n,m,p_1 and p_2 are standard in constraint satisfaction, and our choice for these parameters corresponds to model B in MacIntyre et al. (1998).

For each experiment, 20 problems were generated for each combination of p_1 and p_2, and the median number of constraint checks required to solve the problems was recorded.

The results of these experiments are shown in Figures 5-8. The first thing to note is that forward checking involves many fewer constraint checks than depth first search (by a factor of around 10). Depth first search is better on the highly unconstrained problems (on the left

hand side of the graphs), as in these problems there is a very high density of optimum solutions, and the propagation and retraction of forward checking is a waste of time.

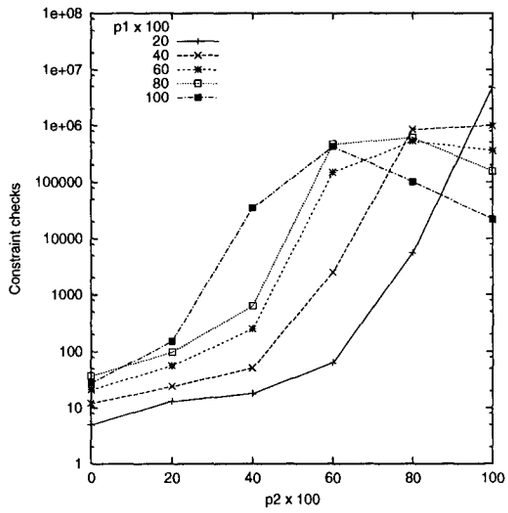

Figure 5: Depth First Search Using Dist1

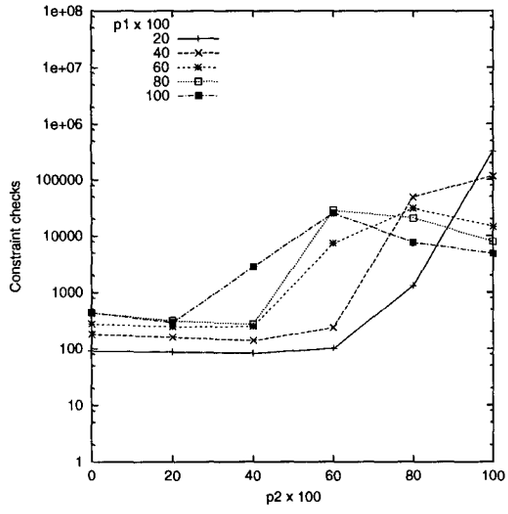

Figure 6: Forward Checking Search Using Dist1

As forward checking is more efficient for most problems (and especially for the hardest problems), we have concentrated on using it instead of depth first search. The previous experiments were repeated using forward checking, but varying p_1 in steps of 0.1, and p_2 in steps of 0.02. 100 problems were generated for each combination of p_1 and p_2. The results are shown in Figures 9 and 10, each curve having a peak further to the left as p_1 increases.

There has been much work in recent years on the 'phase transition' phenomenon in CSPs and related problems, where there is a region in the space of problems which are much harder to solve than elsewhere (Cheeseman et al.,

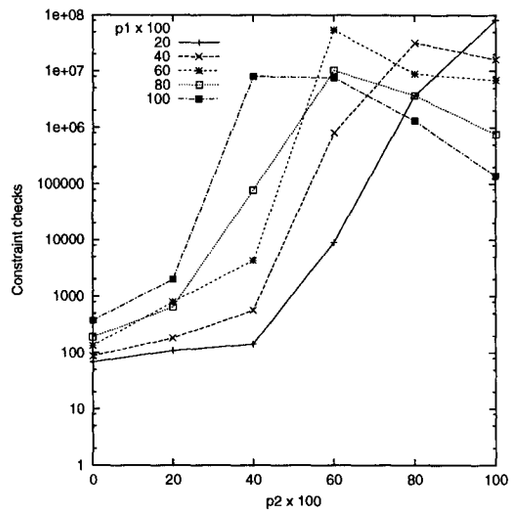

Figure 7: Depth First Search Using Dist2

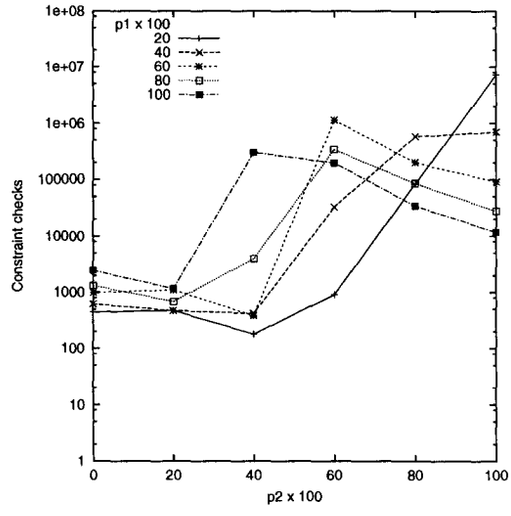

Figure 8: Forward Checking Search Using Dist2

1991; Prosser, 1994). Our results also show a hardness peak, but there are some important differences. It is interesting to compare the hardness of branching CSPs with that of static CSPs, where all variables must be assigned values (if this is possible). This is done in Figure 11 for $p_1 = 60$ (other values of p_1 show similar behaviour). The static CSP has the same number of variables (10) and domain size (10).

For values of p_2 below the hard region, the two problems are equally hard. These problems are simple to solve, as the constraints are very loose, and most or all of the variables can be assigned values easily. The steep climb in difficulty occurs at around the same value of p_2, but for higher values of p_2 the BCSP problems are harder to solve than the corresponding static CSP problems, as we

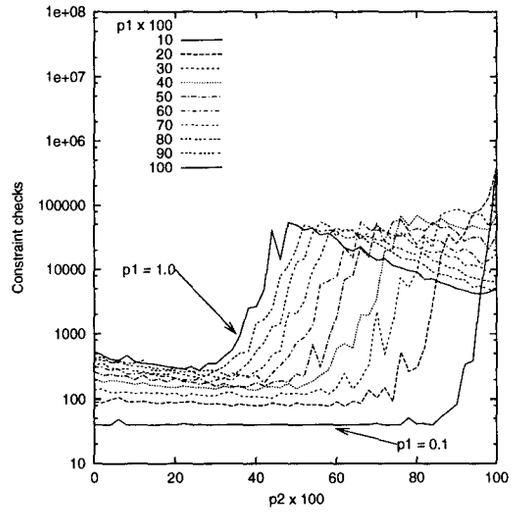

Figure 9: Forward Checking Search Using Dist1

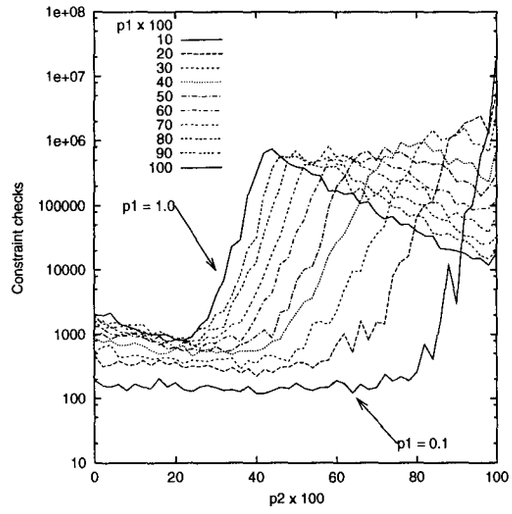

Figure 10: Forward Checking Search Using Dist2

would expect. A static CSP is either solvable or unsolvable, and for high p_2 it is easy to determine that a problem is unsolvable. On the other hand, a BCSP solver must find a 'best' solution, so there is more work to be done in this case, and the hardness declines more slowly.

6 Future Work

In addition to the search algorithms described in this paper, two other algorithms have been implemented, but for which experimental results have still to be produced:

- A thresholding method, which can be used in conjunction with either of the two existing search algorithms. This simply prunes the tree of any branches which have an expected utility less than a certain threshold. In this way, solutions can be quickly found for the most likely and most profitable branches, at the cost of sacrificing the chance to find solutions for the least likely and least profitable branches.

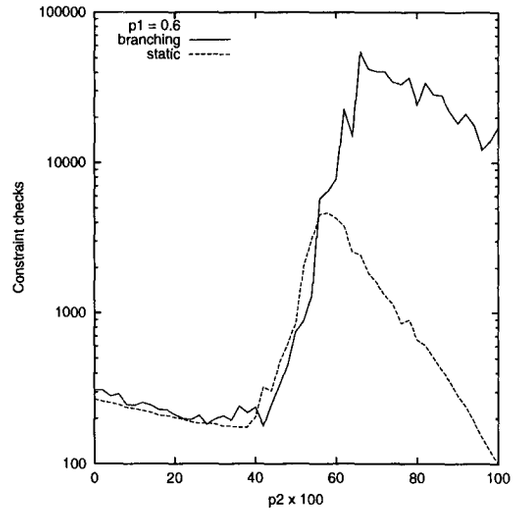

Figure 11: Comparison of BCSP v CSP hardness

- An algorithm to find solutions to problems where the postponement of assignments is allowed. For example, in the original example problem, it would be better to schedule task A, and postpone scheduling B until it was known whether either task C or D would be next. It can be checked that the expected utility would then rise to 7.0. At the moment, the algorithm can only postpone assignments for one step (so it must assign a value to a variable when the identity of the next variable becomes known). Future work may consider how to extend this limit further.

Other future work will concentrate on:

- Investigating new methods for tackling the problems. For example, it seems possible that the MAC algorithm (Maintaining Arc Consistency) (Sabin and Freuder, 1994) could be adapted in a similar manner to forward checking.

- Extending the model to cope with more realistic scenarios. For example, the current model can only represent events as happening in a sequence. It would be useful to be able to specify the times of events, as this would allow for better planning of the solution process. (If the next event is known to be far in the future, more time can be used in finding a good assignment for the current variable).

- Finding an 'anytime' algorithm to solve the problems. This is an algorithm that finds progressively

better solutions to a problem, and can be interrupted to find a reasonably good solution to the entire problem. The algorithms presented here work in a depth first manner, which economises on storage, but may well ignore important branches if they are interrupted.

- Combining our approach of allowing variables to be unassigned with that of standard partial constraint satisfaction.
- Comparing our algorithms with existing methods for scheduling under uncertainty, for example Just-In-Case Scheduling (Drummond et al., 1994).

Acknowledgements

The first author has been funded by a joint studentship from the Faculty of Science and the Department of Computing Science, University of Aberdeen.

We would like to thank Patrick Prosser of Glasgow University for reviewing an earlier version of this work; and also the members of the Computing Science Department of the University of Aberdeen for their comments and discussion; and last, but not least, the two anonymous reviewers for their comments.

References

Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of IJCAI-91*, pages 331–337, San Mateo, CA, USA, 1991. Morgan Kaufmann.

Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In *AAAI-88*, pages 37–43, Saint Paul, Minnesota, USA, August 1988. American Association for Artificial Intelligence.

Mark Drummond, John Bresina, and Keith Swanson. Just-in-case scheduling. In *AAAI-94:The Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, USA, 1994.

Hélène Fargier, Jérôme Lang, Roger Martin-Clouaire, and Thomas Schiex. A constraint satisfaction framework for decision under uncertainty. In *Proceedings 11th International Conference on Uncertainty in AI*, Montreal, Canada, 1995.

Simon French. *Decision Theory: an introduction to the mathematics of rationality*. Ellis Horwood Ltd, 1986.

Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

M. Haralick and G. L. Elliott. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

Roger M. Kerr and Wolfgang Slany. Research issues and challenges in fuzzy scheduling. Technical Report CD94/68, Technische Universität Wien, December 1994.

Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, pages 32–40, Spring 1992.

H. W. Lewis. *Why Flip a Coin? : the art and science of good decisions.* John Wiley and Sons, Inc., 1997.

Ewan MacIntyre, Patrick Prosser, Barbara Smith, and Toby Walsh. Random constraint satisfaction: Theory meets practice. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98*, pages 325–339, Pisa, Italy, October 1998. Springer.

Ian Miguel and Qiang Shen. Hard, flexible and dynamic constraint satisfaction. *The Knowledge Engineering Review*, 14(3):199–220, 1999.

Patrick Prosser. Binary constraint satisfaction problems: Some are harder than others. In *11th European Conference on Artificial Intelligence*, pages 95–99, 1994.

Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *11th European Conference on Artificial Intelligence*, 1994.

Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, 1993.

Richard J. Wallace and Eugene C. Freuder. Stable solutions for dynamic constraint satisfaction problems. In *Workshop on The Theory and Practice of Dynamic Constraint Satisfaction*, Salzburg, Austria, November 1997.

Evaluation of Algorithms to Satisfy Disjunctive Temporal Constraints in Planning and Scheduling Problems

M. A. Salido, A. Garrido, F. Barber

Dpto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, Camino de Vera s/n 46071
Valencia, Spain
msalido@dsic.upv.es, agarridot@dsic.upv.es, fbarber@dsic.upv.es

Abstract

The management of constraints either implicit or explicit in planning and scheduling environments is commonly a very hard task. The way to manage them in the proper way is becoming an important area of study. Moreover, the common approach in planning and scheduling problems is that the constraints to satisfy are disjunctions on intervals (i.e., they imply several possible alternatives). Hence, the complexity of their management is vastly increased (NP-complete complexity). If we use a closure process, there exist two possible solutions to manage these constraints: algorithms that maintain the input and derived constraints and algorithms that only maintain the input constraints. The former require large amounts of memory to store all the generated constraints, whereas the latter do not require large amounts of memory. Here, we present an analysis and evaluation of algorithms to satisfy temporal constraints on metric-disjunctive intervals in scheduling environments using this last kind of algorithms.

1 INTRODUCTION

Planning and Scheduling are two active and relevant areas in Artificial Intelligence which have become of great interest to researchers because of their applications in real problems. There exist many problems (manufacturing, transport problems, planning of production processes, etc.) which should be treated as planning problems with temporal constraints and resource usage constraints. Classical methods for solving them are based on resource allocation.

In Operational Research techniques, the partial sequence of actions is already known, and the actions are principally based on resource allocation. In contrast, Artificial Intelligence methods are used to determine the correct plans. The planner builds a plan as a partially-ordered sequence of actions to reach a goal and the scheduler must ensure that this plan is executable (resource allocation and temporal constraint satisfiability). Nevertheless, the increasing complexity of current problems obligates us to use new, more flexible and more powerful approaches.

1.1 Integrated Planning and Scheduling System

In this section, we present a high-level general view of our integrated system (Figure 1). Our work is developed from an integrated architecture of planning and scheduling (Garrido et al. 1999). The main goal of this integration is to guarantee plan executability and satisfy the problem constraints through the planner and the scheduler in a simultaneous and interactive way. The planning system searches through several alternative partial plans, dispatching the constraints and resource requirements that the scheduler must check. Hence, the scheduler guarantees data constraints and resource assignments are satisfied. Due to this interactive behaviour, the need for improving both planning and scheduling process efficiency becomes more important. In this integrated system, the planning improvements are focused on techniques to reduce the search space, solve conflicts (threats among actions), grouping primitive actions in macro-actions, etc. On the other hand, the main improvements in the scheduler deal with a more efficient management of constraints: resource usage constraints and metric-disjunctive temporal constraints (Barber, 2000). We will focus specifically on the scheduling process and on the temporal constraint management.

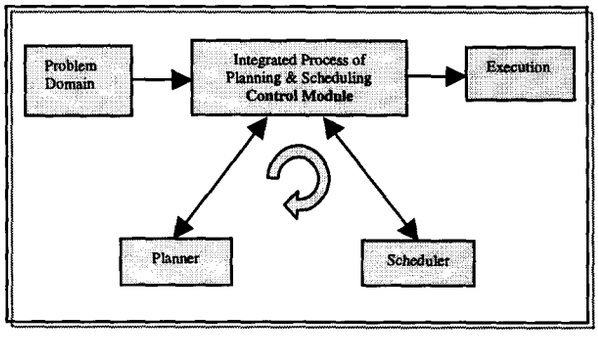

Figure 1. Integrated Planning and Scheduling environment

Since scheduling processes imply temporal constraints about actions and resources, one of the main improvements in the schedulers deals with a more efficient management of constraints. These constraints represent temporal intervals, which can indicate an order of execution, allowing us to represent a wide set of possible solutions. Moreover, these constraints can be disjunctive or non-disjunctive. If the constraints are non-disjunctive, only one order of execution will be possible. However, if the constraints are disjunctive, different (and alternative) orders of execution will be feasible (Baptiste & Le Pape, 1995; Dechter et al., 1991).

Following, we are going to define a simple, typical example in which there exist metric-disjunctive temporal constraints (Dechter et al., 1991). In Figure 2, the input constraints (the explicit ones) of the example are shown in a temporal graph. This example will help us to explain the nature of the problem which is described in this paper:

Michael goes to work either by bus (at least 60') (R_1), or by car [30', 40'] (R_2). Anthony goes to work either by train [40', 50'] (R_3), or by car [20', 30'] (R_4). Today Michael left home (t_1) between 7:10 and 7:20 (R_0), and Anthony arrived (t_4) at work between 8:00 and 8:10 (R_0). We also know Michael arrived (t_2) at work about [10', 20'] (R_0) after Anthony left home (t_3).

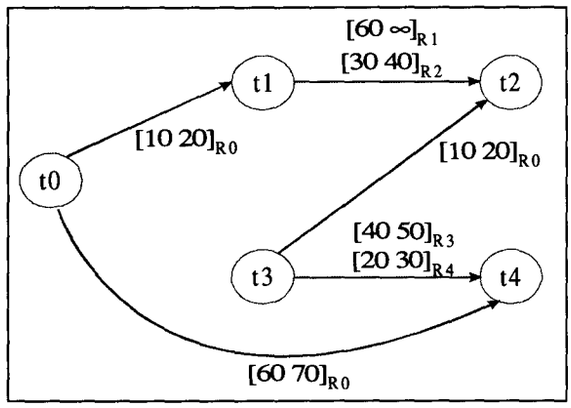

Figure 2. Input constraints of the example

Some queries can be made about the information in the example:

- Who arrives at work earlier? Michael or Anthony?
- Could all these disjunctions be satisfied at the same time?
- Which kinds of vehicles are going to be used in the final solution?
- Is this example's information consistent, i.e., is there a solution?

We might use several techniques in order to answer this last question. One of these techniques is based on traditional methods of *Constraint Satisfaction Problems* (CSPs) that work on a previously known set of constraints obtaining a final solution which answers the previous questions. Nevertheless, the behaviour of this technique is not adequate enough due to its lack of flexibility. For each new set of constraints (which are the result of including or excluding constraints), a CSP process must resolve the entire problem in order to obtain a new solution. However, when the set of constraints is modified, the previous solution may become invalid.

On the other hand, other techniques that can be used to solve problems with metric-disjunctive temporal constraint are based on closure processes. The main goal of closure techniques is to guarantee the consistency of the existing constraints, which may be included and/or excluded by means of an interactive behaviour. There exist several levels of consistency depending on the solution exigency, from the lowest exigent levels (for instance, path consistency) to the most exigent ones (global consistency) and two alternative to guarantee this consistency:

Algorithms that maintain derived constraints

In this case, derived constraints which are obtained from the set of input constraints are explicitly represented in the temporal network. Consequently, these algorithms require large amounts of memory to store all the derived constraints in the closure process. For instance, in the previous example (Figure 2), the derived constraints would be t0→t2, t0→t3, t1→t3, t1→t4 and t2→t4 (see Figure 3). When a constraint between two temporal points is asserted into the system, these algorithms check its consistency with the existing constraint. If the new constraint is consistent, the resulting constraint to maintain will be the most restrictive combination between them, and the input constraint will not be maintained. Due to the fact that the graph is always propagated, retracting a constraint is a very complex task (it is difficult to determine the input constraint) and it obligates us to repeat the closure process for every input constraint.

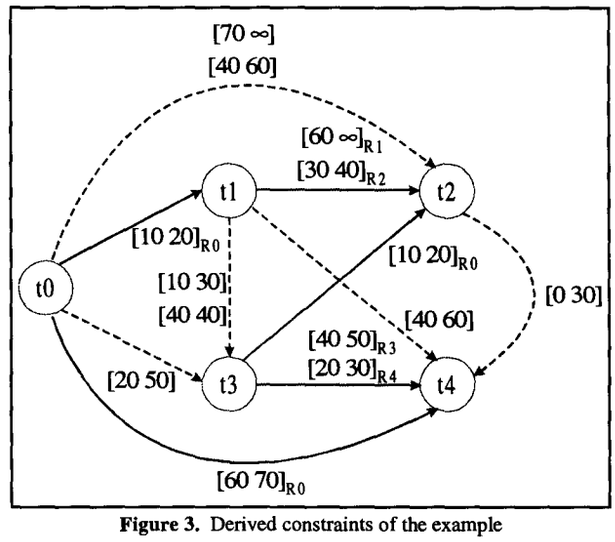

Figure 3. Derived constraints of the example

Algorithms that only maintain input constraints

These algorithms only maintain input constraints (the explicitly asserted ones which appear in Figure 2). The main advantage of these algorithms is that they do not require large amounts of memory to store the derived constraints because they do not carry out any propagation process between the new constraint and the existing ones. For this reason, these algorithms may be used in an attempt to reduce the complexity of asserting new constraints and they ease the process of retracting some asserted constraints into the system. When a new constraint between two temporal points (nodes) is inserted into the system, the algorithm retrieves the most restrictive constraint (the minimal one) between these two points. Next, the algorithm checks whether the new constraint is consistent with the retrieved one. If it is consistent, the new constraint is accepted, and if not, it is rejected. The main difficulty is to calculate the minimal constraint in a nonpropagated disjunctive graph in the most efficient way. It is a complex task because there exists an exponential number of paths that represent the constraints to be calculated. Since the graph may contain both negative arcs in parallel (disjunctions) and circuits, we must find all paths in the associated graph in order to obtain the *shortest path* which represents the minimal temporal constraint (Goldfarb, 1991).

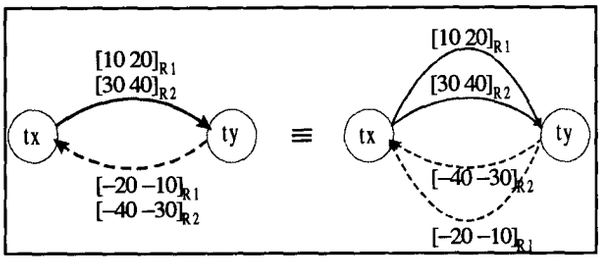

Figure 4. Equivalence between disjunctions and arcs in parallel

As can be observed, problems with metric-disjunctive temporal constraints may be represented by means of a directed graph with negative weights and arcs in parallel (Figure 4). Nodes represent time points, and arcs between nodes represent metric-disjunctive temporal constraints among time points. The negative weights are temporal intervals, and arcs in parallel are denoted by means of disjunctive weights. For instance, let x_i and x_j be two nodes. The metric-disjunctive constraint might be

$$x_i \{[d_k, D_k], [d_l, D_l], \ldots\} x_j : d_k, D_k, d_l, D_l \in \Re .$$

Hence, these problems may be treated as problems over temporal graphs. One of the most studied problems in graph theory is the shortest path problem or the *k*-shortest paths problem in a network (Ford & Fulkerson, 1974; Shier, 1979; Goldfarb, 1991). Over the last four decades, several algorithms have been proposed for solving the shortest path problem or the *k*-shortest paths problems (Ravi et al., 1992). Many of these methods are variants of the well-known Bellman-Ford algorithm (Aho & Ullman, 1992; Ford & Fulkerson, 1974). Our problem attempts to generate algorithms for finding all paths (without circuits) from a node source x_1 to a node sink x_n, which represent time points. Therefore, we have proposed a *unidirectional search algorithm* and a *bidirectional search algorithm*. The *unidirectional search algorithm* generates a spanning tree to find all paths between two nodes. This algorithm obtains all paths by means of a strategy which is similar to the *inorder* strategy. The number of generated nodes is exponential according to the number of nodes of the graph, although its spatial cost is linear. In contrast, the *bidirectional algorithm* does not generate the entire spanning tree to find all paths between two nodes. The search process starts forward from a node (the source node) and backward from the other node (the sink node). When the partial paths meet, a new complete path is found. Thus, the number of generated nodes is decreased with regard to the *unidirectional algorithm* (it is theoretically the square root), but the spatial cost is increased because both forward and backward trees must be simultaneously stored in memory.

In section 2, we present the specification of the *unidirectional* and *bidirectional search algorithm*. The analysis of the algorithms and their evaluation are presented in section 3 and section 4, respectively. Conclusions and future lines of work are discussed in section 5.

2 SPECIFICATION OF THE ALGORITHMS

In this section, both the *unidirectional* and *bidirectional search algorithm* are presented. We begin introducing the necessary notation used in the algorithms.

2.1 Preliminaries. Notation

Definition 1

Let $G = (V, E)$ be a connected graph. V is a finite non-empty set of nodes. E is a set of pairs of nodes called arcs. $V(G)$ and $E(G)$ represent the set of nodes and arcs of graph G respectively. We assume $V = \{x_1, x_2, \ldots, x_n\}$, $|V| = n$ and $|E| = l$. Node x_i of the arcs (x_i, x_j), or (x_j, x_i), is said to be adjacent to node x_j. We denote $Adj(x_i)$ as the set of x_i adjacent nodes, and we say $deg(x_i)$ is the x_i degree.

Definition 2

A *path* from node x_1 to the node x_n in a connected graph is an ordered sequence

$[(x_1,x_2),(x_2,x_3),\ldots,(x_i,x_{i+1}),\ldots,(x_{n-1},x_n)]$ of nodes such that there is an arc from each node to the next one, i.e., (x_i,x_{i+1}) is an arc for $i=1,2,\ldots,n-1$. Without loss of generality, we denote this sequence by $(x_1,x_2,\ldots,x_n)$. A *valid path* is a path without repeated nodes. The maximal *length* of a valid path $(x_1,x_2,\ldots,x_n)$ in a graph of n nodes is $n-1$, i.e., the number of arcs along the path.

Definition 3

Let $G=(V,E)$ be a connected graph and let (x_i,x_j) be an arc. We can say x_i is a *predecessor* of x_j and x_j is a *successor* of x_i.

Definition 4

We denote the ordered list of the x_i forward predecessors as

$P_f(x_i)=(x_0,x_1,\cdots,x_{i-1}): x_j \neq x_p \ \forall j \neq p, 0 \leq j,p \leq i-1$.

Analogously, we denote the ordered list of the x_i backward predecessors as

$P_f(x_i)=(x_{i+1},x_{i+2},\cdots,x_n): x_j \neq x_p \ \forall j \neq p, i+1 \leq j,p \leq n$.

Definition 5

$\overline{P}_f(x_i)$ is 'the x_i direct predecessor' in forward exploration, whereas $\overline{P}_b(x_i)$ is 'the x_i direct predecessor' in backward exploration.

Definition 6

The set of x_i valid predecessors in forward exploration (nodes which do not form circuits) is $Suc_f(x_i)=Adj(x_i)\setminus P_f(x_i)$, and the set of x_i backward true predecessors is $Suc_b(x_i)=Adj(x_i)\setminus P_b(x_i)$.

Definition 7

We denote $(a_1,a_2,\ldots,a_n)^T$ by $(a_n,\ldots,a_2,a_1)$.

2.2 The unidirectional search algorithm

The *Unidirectional Search Algorithm* is a well-known algorithm which finds all paths between nodes x_1 and x_n by generating the spanning tree by means of a unidirectional strategy. It starts from the node source x_1 in level 0. Level 1 is formed by the adjacent nodes of the node x_1. Thus, level i is formed by the adjacent nodes of level i-1 which have not been processed in that branch (i.e. $Suc(x_1)$). Hence, the algorithm terminates when all paths have been found.

2.3 The bidirectional search algorithm

The *Bidirectional Search Algorithm* is an algorithm which finds all paths between nodes x_1 and x_n and, therefore, it finds the minimum constraint between x_j and x_n. This algorithm works on a bidirectional strategy in which it finds paths starting from the node source x_j in a forward direction and from the node sink x_n in a backward direction and varying the direction of the expansion alternatively. Hence, the complete path will be found when both semi-paths meet.

Notation

F_f = Set of frontier nodes generated in the forward spanning.
F_b = Set of frontier nodes generated in the backward spanning.
H_f = Set of leaf-nodes generated in forward exploration.
H_b = Set of leaf-nodes generated in backward exploration.
C_f = Set of reached nodes in forward exploration.
C_b = Set of reached nodes in backward exploration.
C = Set of found paths.
$\oplus$ = List concatenation operator.

Algorithm
Bidirectional Search Algorithm (*Graph*, x_1, x_n)

; Initialisation stage
$C_f := \{x_1\}$, $H_f := \{x_1\}$, $F_f := \phi$;
$C_b := \{x_n\}$, $H_b := \{x_n\}$, $F_b := \phi$; $C := \phi$;
for $i = 1$ **to** n **do**
  $P_f(x_i) := nil$, $P_b(x_i) := nil$
endfor
if $n\ MOD\ 2 = 0$ **then** $z = n\ DIV\ 2$
else $z = (n-1)\ DIV\ 2$
endif

; Search stage
for $k=1$ **to** z **do**
  Search $(f,b,F_m,H_m,C_v,C_m,C;C_m,H_m,F_m,C)$;
forward
  Search $(b,f,F_m,H_m,C_v,C_m,C;C_m,H_m,F_m,C)$;
backward
endfor

; Odd stage
if $n\ MOD\ 2 = 1$ **then**
Search $(f,b,F_m,H_m,C_v,C_m,C;C_m,H_m,F_m,C)$;
return C
endAlgorithm

Algorithm 1. The Bidirectional Search Algorithm

Algorithm *Search* $(\mathbf{I}: m, v, F_m, H_m, C_v, C_m, C; \mathbf{O}: C_m, H_m, F_m, C)$

$\forall x_i \in H_m$ **do**
  if $x_i \in C_v$ **then**
    $\forall x_j \in Suc_m(x_i)$ **do**
      If $x_j \in \overline{P}_v(x_i)$ **then**
        If $\{P_m(x_i) \oplus \{x_i\} \oplus \{P_v(x_i)\}^T\} \not\subset C$ **then**
        $C \leftarrow C \cup \{P_m(x_i) \oplus \{x_i\} \oplus \{P_v(x_i)\}^T\}$; *new path has been found*
  $F_m \leftarrow F_m \cup Suc_m(x_i)$; *we update* F_m
  $C_m \leftarrow C_m \cup Suc_m(x_i)$; *we update* C_m
  $\forall x_j \in Suc_f(x_i)$ **do**
  $P_m(x_j) := P_m(x_j) \oplus \{x_i\}$; *we update the* x_j *predecessors*
$H_m \leftarrow F_m$; *we assign to set of leaf-nodes* H_m *the set* F_m
$F_m \leftarrow \phi$; *we empty the set of frontier nodes*
end;

Algorithm 2. The Search Algorithm

Theorem 1

Let $G=(V,E)$, $|V|=n$, $|E|=l$ be a connected graph. The bidirectional search algorithm finds all paths from a node source x_1 to node sink x_n.

Proof

We assume a path $(x_1,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_n)$ exists, and the bidirectional search algorithm does not find it. We will reason over this premise and we will demonstrate that this path has been found by the algorithm (arriving at a contradiction).

if $\{x_1,x_2,\ldots,x_n\} \not\subset C \Rightarrow \exists x_k \in \{x_1,x_2,\ldots,x_n\}$:

$$\begin{cases} \text{a) } x_k \notin H_f \wedge x_k \notin H_b \text{ for any } H_f, H_b \\ \text{or} \\ \text{b) } x_k \in H_f \vee x_k \in H_b, \text{but } Suc_{(f,b)}(x_k) = \phi \end{cases}$$

a) If $x_k \notin H_f \wedge x_k \notin H_b \Rightarrow \neg\exists\, x_i : x_k \in Suc_{(f,b)}(x_i)$
$\Rightarrow Deg(x_i) = 0$ (contradiction) because the graph is connected.

b) $x_k \in H_f \vee x_k \in H_b$ but $Suc_{(f,b)}(x_k) = \phi$
We have assumed that $\{x_1,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_n\}$ is a path $\Rightarrow Suc_f(x_k) = x_{k+1}$ and
$Suc_b(x_k) = x_{k-1} \Rightarrow Suc_{(f,b)}(x_i) \neq \phi$ (contradiction).

All paths are going to be found because the algorithm expands both the forward and the backward tree until the $n/2$ level. Consequently, no path will be lost due to the fact that the algorithm finds all the paths of length $n-1$, the maximal length of a valid path without circuits in a graph with n nodes.

3 ANALYSIS OF THE ALGORITHMS

In this section, we present both a temporal and spatial analysis of the proposed algorithms in section 2. Their respective complexities are based on the number of nodes in the initial graph.

3.1 The unidirectional search algorithm

Let n be the number of nodes in the graph. The branching factor is determined by the number of descendent nodes from each node. Thus, the effective branching factor B_f is the maximum of these numbers. The maximum depth of the generated tree is bounded by the maximum length of all the valid paths. Since the valid paths do not contain circuits, the maximum length is lower than the number of nodes (specifically, the maximum length will be $n-1$). Therefore, the maximum number of generated nodes is given by the following formula:

$$1 + \sum_{i=1}^{n-1} \prod_{j=0}^{i-1} (B_f - j) \in O({B_f}^{n-1}) \in O({B_f}^{n})$$

However, the real number of nodes is lower because a new node can not be generated due to a cycle in the current path. Moreover, as B_f is always lower than n, the temporal complexity in terms of n is $O(n^n)$.

The unidirectional search algorithm obtains all paths by means of a strategy which is similar to the *inorder* strategy. Therefore, it only maintains one complete path each time as maximum. Hence, the spatial complexity is O(n).

3.2 The bidirectional search algorithm

In order to calculate the temporal complexity, the bidirectional algorithm behaves as two unidirectional algorithms. On the one hand, it starts from the source node towards the sink node. On the other hand, it starts from the sink node towards the source node. Thus, the two generated trees have exactly half the depth (instead of generating a *n*-depth tree, two *n/2*-depth trees are generated). Consequently, the number of nodes is given by the next formula:

$$2\left(1+\sum_{i}^{(n-1)/2}\prod_{j=0}^{i-1}(B_f-j)\in O(B_f{}^{(n-1)/2})\right)\in O(B_f{}^{n})$$

As can be deduced of this formula, the asymptotic spatial complexity is $O(n^n)$. Its complexity is the same as that of the unidirectional search algorithm, but its practical behaviour is much better (because it approximately generates the square root of nodes).

On the other hand, the spatial complexity is greater than the unidirectional complexity. In order to be able to join the partial paths found, it is necessary to maintain the two whole trees. Hence, the required memory is proportional to the number of generated nodes, and the spatial complexity is

$$O(2\cdot n^{(n-1)/2})\in O(n^{n/2})\in O(n^{n})$$

4 EVALUATION OF THE ALGORITHMS

In order to evaluate the performance of the two proposed algorithms, we implemented them in *Common Lisp*. The computer used in our tests was a Sun Ultra 10 Sparc with 256 Mb. of memory and with the SunOS 5.7 operating system.

We evaluated the algorithms by their execution time and the number of nodes generated. Once the graph is defined, the algorithms carry out a search process between two nodes which are randomly chosen. The selected graphs were random graphs which consisted of a set of nodes (from 20 to 100) and a set of arcs, which represented 2-disjunctive and non-disjunctive constraints. We limited the number of disjunctive constraints to 10. This implies having an equivalent number of 2^{10}=1024 different non-disjunctive graphs for each generated graph. In addition, we varied the branching factor, from 1.1 up to 1.5 which implies varying this factor from 2.2 up to 3.

The mean and variance of the execution time (in hundredths of a second) and the number of nodes generated in a 20-node graph for the unidirectional algorithm are shown in Table 1. We modified the effective branching factor B_f from 1.1 up to 1.5. Table 2 shows the same values for the bidirectional search algorithm as are shown in Table 1 for the unidirectional search algorithm.

Unidirectional (20 nodes)	Time (hs)		Nodes Generated	
	Mean	Variance	Mean	Variance
$B_f = 1.1$	0	0	111.6	72.4
$B_f = 1.2$	0	0	148.4	46.4
$B_f = 1.3$	40	54.8	636	229.4
$B_f = 1.4$	40	54.8	3225.2	1959.3
$B_f = 1.5$	60	54.8	3927.4	2424.3

Table 1. Execution time and nodes generated in the Unidirectional Search Algorithm (20-node graph).

Bidirectional (20 nodes)	Time (hs)		Nodes Generated	
	Mean	Variance	Mean	Variance
$B_f = 1.1$	0	0	86.8	14.3
$B_f = 1.2$	20	44.7	125.8	26.8
$B_f = 1.3$	20	44.7	471	77.8
$B_f = 1.4$	480	164.3	1378.4	297.7
$B_f = 1.5$	940	634.8	2167.8	1179.2

Table 2. Execution time and nodes generated in the Bidirectional Search Algorithm (20-node graph).

As can be observed in the previous tables, the mean of the execution time is higher in the bidirectional algorithm although the number of nodes generated is lower. This average time is higher due to the fact that, in the computation time in each node is much higher in the bidirectional search algorithm: it is necessary to check whether a path has been found by joining two semi-paths (one from the forward tree and the other semi-path from the backward tree). As analysed in section 3, the number of nodes generated is lower in the bidirectional search algorithm. Theoretically, the bidirectional algorithm asymptotically generates the square root of the nodes generated by the unidirectional search algorithm. However, the number of nodes generated in practice is approximately only half. The variance values are quite high due to the fact that the results obtained are greatly dependent on the topology of the graph.

In Figure 5 and Figure 6, we present the graphics of the nodes generated in the two search algorithms for each graph of our tests.

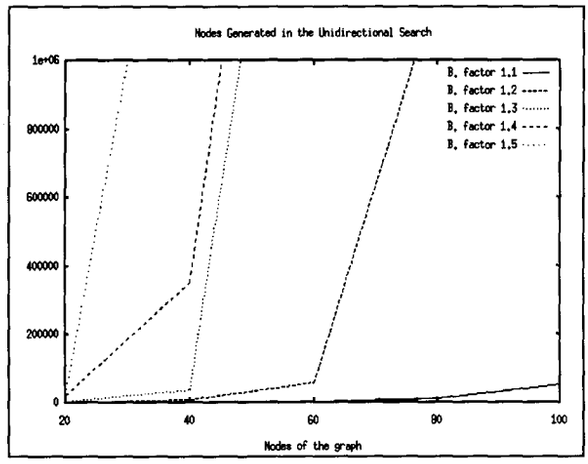

Figure 5. Nodes generated in the unidirectional search

As can be observed in Figure 5, the higher the branching factor, the greater the number of nodes generated. For instance, since a higher branching factor implies a greater number of arcs, , the unidirectional algorithm is not able to manage 40-node graphs with a branching factor higher than 1.3.

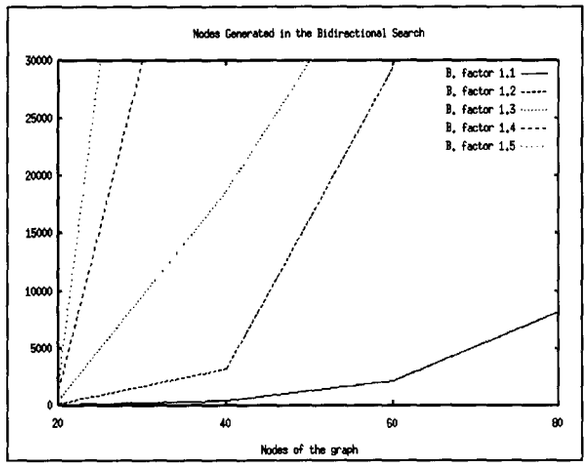

Figure 6. Nodes generated in the bidirectional search

In Figure 6, the number of nodes generated is lower than in Figure 5 because the bidirectional search algorithm generates less nodes for the same graph. In this case, the bidirectional search algorithm has generated less than 3500 nodes for a 40-node graph with branching factor 1.2, whereas the unidirectional search algorithm has generated more than 7000-nodes for the same graph.

The graphics of the execution time are presented in Figure 7 and Figure 8. In these figures, the execution time demonstrates that the bidirectional search algorithm takes much longer than the unidirectional search algorithm for the same graphs and the same branching factors. For instance, the bidirectional search algorithm takes 13420 hundredths of second in the 40-node graph with branching factor 1.3 and, on the contrary, the unidirectional search algorithm takes only 460.

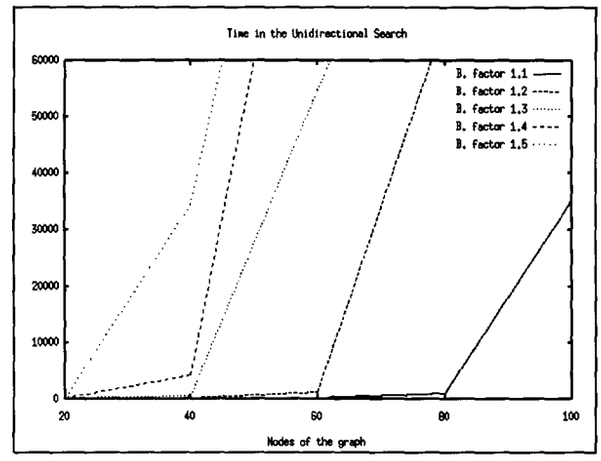

Figure 7. Execution time for the unidirectional search

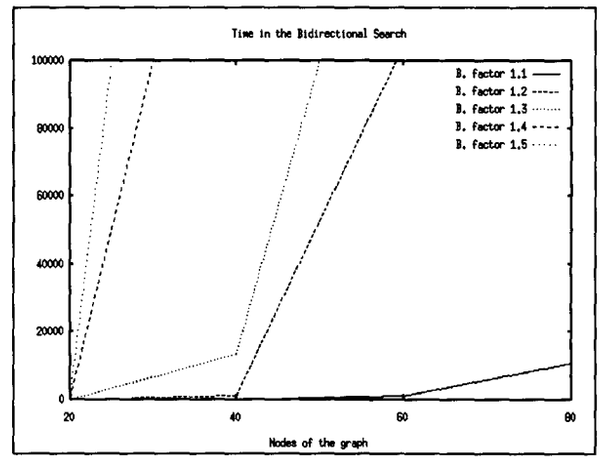

Figure 8. Execution time for the bidirectional search

Finally, it is important to note that these algorithms are not applicable to complete graphs with many nodes (in which there exists an arc between each pair of nodes). Due to the huge number of existing paths, it is impossible to obtain all the paths. For instance, in a graph with only 20 nodes, the number of existing paths is approximately 10^{16}.

5 CONCLUSIONS

In this paper, we have presented two algorithms to find the most restrictive constraint between two temporal points in a temporal graph. Since each arc represents a constraint on temporal disjunctive intervals, the arc's weight can either be positive or negative. Therefore, there does not exist an admissible heuristic that avoids having to find all the paths in order to obtain this minimal constraint.

These proposed algorithms can also be used in other problems based on graph exploration. However, even though the number of generated nodes is decreased by

using the bidirectional search algorithm, the needed storage space is increased.

As can be deduced from the results presented, these kinds of algorithms are not applicable to graphs with many nodes. Graphs with more than 100 nodes become unmanageable, specially if the branching factor is higher than 1.5 or 2. As can be seen in the results of the comparative study, the bidirectional algorithm generates fewer nodes than the unidirectional one. However, the spatial cost of the unidirectional algorithm is lower than the bidirectional search algorithm cost. The results demonstrate the suitability of each algorithm for use in the management of temporal constraints. These methods are not adequate enough in many real problems due to the difficulty of finding the optimal solution. Thus, our work is focused on reducing this complexity by means of the following techniques:

- Heuristic techniques in order to solve real problems in polynomial time. We can reduce the complexity of the search processes by using a heuristic that helps us to decide which path must be generated, and which must be discarded, according to some criteria. These criteria might lead to a path which does not represent the most restrictive constraint and we would have to choose between a nonminimal constraint obtained in a faster way or a minimal constraint obtained in a slower way.
- Combined CSP and closure techniques to improve the behaviour of planning and scheduling processes (Alfonso & Barber, 1999).
- Other techniques that do not guarantee the total consistency, for example path consistency.
- The use of new data structures which might improve the behaviour of these processes allowing us to have more powerful tools for solving real problems of planning and scheduling.

We can also use other methods that diminish the consistency of the generated graph (Freuder, 1982) and, therefore, decrease the complexity of the process.

Another interesting idea for the application of these algorithms to scheduling processes is to work on nondisjunctive graphs, which can be solved in polynomial time. When a disjunctive constraint appears, the method will select one of the disjunctions (according to criteria such as slack, due time, etc.) and it will ignore the other ones (Alfonso & Barber, 1999). If the selection has been appropriate the other disjunctions will be discarded. However, if the selection has been inappropriate, a backtracking process will be necessary in order to continue through another disjunction. In this case, the criteria are focused on minimising the number of backtracking stages.

ACKNOWLEDGEMENTS

This work has been proposed in the *Intelligent Planning & Scheduling Group* of the Polytechnic University of Valencia (http://www.dsic.upv.es/users/ia/gps) and partially supported by the grant CICYT/TAP98-0345 from the Spanish government.

REFERENCES

A.V. Aho and J.D. Ullman, 'Foundations of Computer Science', *Computer Science Press*, (1992).

M.I. Alfonso and F. Barber, 'Combinación de Procesos de Clausura y CSP para la Resolución de Problemas de Scheduling', *Proceedings of the VIII Conferencia de la Asociación Española para la Inteligencia Artificial*, 1(3), 35-42 (1999).

P. Baptiste and C. Le Pape, 'A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling', *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 600-606, Morgan Kaufmann, (1995).

F. Barber, 'Reasoning on complex disjunctive temporal constraints', *Journal of Artificial Intelligence Research*, (2000).

R. Dechter, I. Meiri, and J. Pearl, 'Temporal constraint networks', *Artificial Intelligence* 49, 61-95 (1991).

L.R. Ford, D.R. Fulkerson, 'Flow in Network', *Princeton University Press*, (1974).

E.C. Freuder, 'A sufficient condition for backtrack-free search', *Journal of ACM* 29(1), 24-32, (1982).

A. Garrido, E. Marzal, L. Sebastiá and F.Barber, 'Un Modelo de Integración de Planificación y Scheduling', Proceeding of CAEPIA'99 1(3):1-9, (1999).

D. Goldfarb, 'Shortest Path Algorithm Using Dynamic Breadth-First Search', *Network*, 21, 29-50 (1991).

R. Ravi, V. Madhav, and C. Pandu, 'An Optimal Algorithm to Solve the All-Pair Shortest Path Problem on Interval Graphs', *Network*, 22, 21-35 (1992).

D.R. Shier, 'On Algorithm for Finding the *K*-Shortest Paths in a Network', *Network*, 9, 195-214 (1979).

Scheduling Activity in an Agent Architecture

Ignacio Soto
Dpt. Tecnologías de las Comunicaciones
Universidad Carlos III de Madrid
c/Butarque 15
28911 Leganés (Madrid)
Spain
isoto@it.uc3m.es

Abstract

Agents for applications in dynamic environments require artificial intelligence techniques to solve problems to achieve their objectives. For example, they must develop plans of actions to carry out missions in their environment, in other words, to achieve some state in the world. But also, the agents must fulfill real-time requirements that arise because the characteristics of the applications and the dynamism of the environment. In this paper we analyze the use of a schedule of activity in an agent architecture to control the resources (time) needed by agents to accomplish their objectives.

1 Introduction

An agent must achieve objectives in dynamic and complex environments. To achieve these objectives it must carry out a series of tasks. We call task to a schedulable and executable procedure. A task can be computational, i.e., one that tries to find out other tasks which once executed will eventually let the agent achieve its objectives. Or a task can embody actions in the real world and/or perceptions of the environment.

On the other hand the activity of the agent is conditioned by real-time requirements:

1. The application can have real-time constraints: the agent must fulfill each objective before its deadline.
2. The agent must be reactive in front of events in the environment. Some will need an immediate response by the agent to guarantee its own security, others will allow for deliberation to deal with them (to find out which tasks to execute associated with them).
3. The behavior of the agent must be robust in the sense of always doing useful work. If it has not resources to fulfill all its objectives, it must try to fulfill its most important ones, while not being distracted by objectives it cannot achieve.

Requirement 2 has been the main aim for agent architectures that have been used to build agents that need to interact with a real world environment (for example, controlling robots). Less effort seems to have been made to deal with requirements 1 and 3 (but see section 5 in which we compare our work with other approaches).

In section 2 we describe an agent architecture to fulfill the requirements mentioned above. This agent architecture is based on the blackboard model. We identify the characteristics that this model offers that, we believe, are useful for building intelligent agents that combine the use of different artificial intelligence techniques with real-time requirements. And then, we propose modifications to the basic model that are needed to fulfill these requirements. In particular, we propose that, to be able to deal with resource constraints of high level objectives (missions) of the agent, the agent architecture can benefit from having an schedule of the predicted activity to achieve those objectives. In section 3 we describe the role of the schedule of tasks that defines the activity of the agent and how can be built under real-time constraints. In section 4 we present experimental results about the behavior of the architecture using the schedule. In section 5 we compare the role of the schedule in our agent architecture with the role that plans play in other agent architectures, and comment on other related work. And finally, in section 6 we summarize our results and give directions for future research.

2 Agent Architecture

Our research group has been working in developing an agent architecture to fulfill the requirements mentioned in the introduction. This architecture is called AMSIA.

AMSIA is based on the blackboard model (Corkill, 1991; Carver and Lesser, 1992; Hayes-Roth, 1988; Pfleger and Hayes-Roth, 1997). Using this model, we can divide the knowledge of our agents in a series of Knowledge Sources (KSs). This division has several advantages:

1. **Distribution**: first, of course, we are dividing the activity needed to solve a problem. The parts should be easier to build than the complete solution. Moreover, incremental and/or hierarchical reasoning is natural in this model.

2. **Software reuse**: each part solves a problem and so, it can be reused in different situations where the problem appears and/or in different applications (Hayes-Roth et al., 1995). Application programmers can take the basic architecture and bring or build knowledge sources to deal with their domain problems.

3. **Flexibility**: it allows the agent to use different reasoning methods. Each knowledge source is independent from the others and can be built in any form needed by the application. The knowledge sources doesn't communicate directly. The only restriction is that a knowledge source must be capable of understanding the representation of the knowledge in which it is interested and that will have been left in the blackboard by other knowledge sources.

4. **Estimation of resource requirements**: the division of the activity needed to solve a problem in parts makes easier to estimate resource requirements. The agent can do this estimation separately for each part, and it can compensate the resource use of different parts. Also, real-time artificial intelligence techniques, such as anytime algorithms or approximate processing, can be integrated smoothly in knowledge sources.

In AMSIA, we have refined the traditional blackboard model with two new properties:

1. All the activity in the system is explicitly scheduled. With the term activity we refer both to actions in the real world and to actions internal to the agent (i.e. reasoning activities including planning). This is the base to control the use of resources.

2. We make independent in the agent the following of a line of activity which, at the same time, generates possibilities of activity for the future, from the decision of what line of activity must be followed.

We believe that the second property defines an important division needed to achieve real-time performance. The line of activity of an agent represents its committed resources. It defines a behavior with some profit for the agent. Choosing future lines of action is the act of committing resources to achieve some profit. The separation of these two activities allows the agent to control its opportunism.

In the past we have explored achieving this division using a multiprocessor architecture for our agent (Soto et al., 1997, 1998). We used a processor to follow a line of activity and offer new ones; and another to analyze the possibilities that were created by the agent by following its line of activity, and to choose the future line of activity of the system. We continue working in this architecture but, in this paper, we explore another approach to the problem, namely we study how AMSIA achieves the mentioned division in time, and not with the use of two processors. In this architecture the own schedule of future activity of the agent must include time to consider and choose among possibilities of future activity. This is not easy because there are situations in which the agent doesn't know when possibilities for future activity are going to be opened. We study how to deal with this situation in next section.

To predict future activity the agent must use planning techniques. In AMSIA, reasoning tasks can create plans of objectives; and control tasks can translate those to plans of tasks (to achieve the objectives), assign them resources, and introduce them in the schedule. Decisions can be delayed simply by using a reasoning task to decide what to do about an objective in the right moment, perhaps extend it in a series of sub-objectives. Changes in the plan of objectives are easy because they are in the blackboard and can be accessed by any task. Changes in the method (task) to achieve an objective are also easy because the alternative tasks are kept associated with the corresponding objective.

Figure 1 shows the conceptual model of AMSIA. Notice:

1. Control and execution are independent activities according with property two above, but both of them get its time of execution from the schedule that defines the activity of the system.

2. Both control and domain actions have preconditions. This is a check to ensure that the conditions expected by the task to be executed are really so when it is going to be executed. If they are not, the task is not executed and an external (see bellow) event is generated. Soto et al. (1998) presents a more detailed discussion of this issue.

3 Scheduling Tasks in AMSIA

3.1 Construction of the Schedule

To have a schedule of activity allows AMSIA to control the use of resources. The problem is how to build this schedule.

In AMSIA, activity is triggered by events. These events signal that something interesting has happened. They represent changes in the blackboard that can be consequence of a reasoning activity or of perceptions in a broad sense: we consider perceptions readings from sensors but also messages from other agents or a timer that expires.

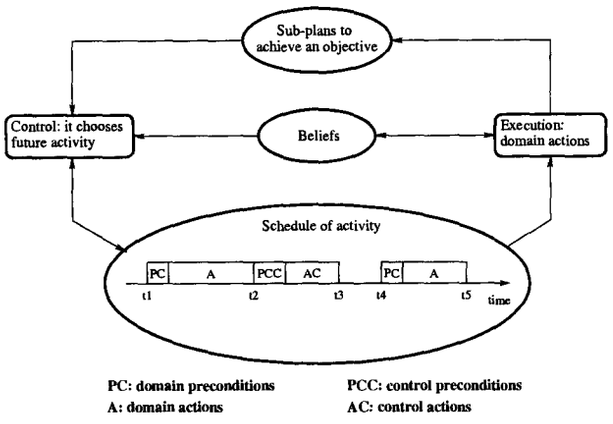

Figure 1: The Conceptual Model of the Agent Architecture

For each event there will be a number of KSs whose knowledge can be useful in that situation. The agent identifies those KSs, creates tasks based on them, builds possible sequences of those tasks to do the work needed in front of the event, and then it must add one of the sequences to the global schedule that defines its (of the agent) future activity. Different sequences will make different trade-offs in resource usage and quality of expected results. The schedule registers the resources allocated to the tasks. In our implementation the only resource considered is time and so, it is kept in the schedule the instants before which the execution of each task must begin and end.

The activity needed to deal with an event (identify KSs, create tasks, build sequences of tasks, and introduce one in the schedule) is too complex to be done in a fixed or negligible time. Instead, this activity must be scheduled itself, i.e., a task to deal with the event, to do that activity, must be included in the schedule. To do so, we divide the events in two different kinds:

- internal: events internal to the reasoning flow of the agent;
- external: events external to that flow.

Internal events are created by the reasoning activity of the agent. They show the need/possibility of using new tasks to develop the reasoning work in which the agent is involved. For example, the execution of a task in certain level of abstraction can discover that it is needed the execution of several tasks in a lower level of abstraction. So, internal events can be anticipated by the agent and it must include in the schedule of activity a task to deal with them.

But there are also events that aren't produced by the reasoning activity of the agent. We call them external events. Examples are certain situations perceived in the environment, or a message from other agent. The situation is the same as before in the sense that the agent needs to execute a task to deal with the events. The difference is that the agent cannot anticipate these events and so, it cannot have in the schedule tasks to deal with them. The solution is that, when an external event is received by the agent, asynchronously, it must include a task in its schedule to deal with it.

The agent can control its openness and reactivity in front of events because it decides when and how it is going to deal with them.

The scheduler works with the algorithm that is shown in figure 2.

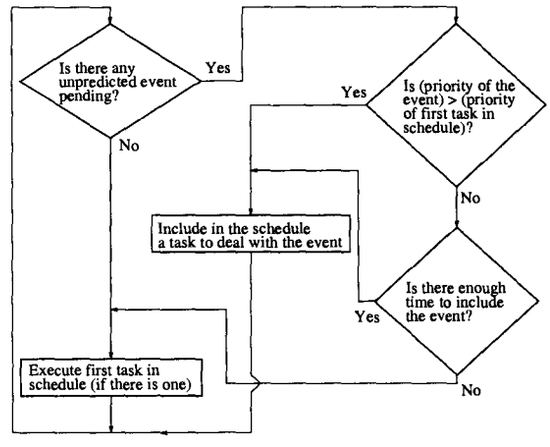

Figure 2: Algorithm of the scheduler

The scheduler is non-preemptive (it works between tasks, not when an event is received, which is reasonable in deliberative tasks but see section 6 conclusions and future work) and dynamic (of course it doesn't know the future time of arrival of new events to the agent).

3.2 Example of Schedule Construction

In figure 3 it is shown an example of the algorithm working. We begin with an empty schedule. An external event is received and, hence, the scheduler adds a task to the schedule to deal with it. To assign time to this task the scheduler has the information of the kind of event and (possibly) the time that has spent in tasks to deal with the same kind of events in the past (more on this later). This task is then executed resulting (in this example) in the scheduling of two new tasks. The scheduler algorithm is run, as there is no new external events, the next task in the schedule (task number two in the figure) is run. This is a deliberative task and as a result of its reasoning activity internal events are generated.

There are not external events and so next task (task three) is executed. This task is in the schedule to deal with the internal events generated by task two. As a result, new tasks are added to the schedule.

Task four is again a deliberative task that generates internal events. A point to notice is that the agent can predict the time that is going to need to execute not only task four but also the tasks that task four identifies for

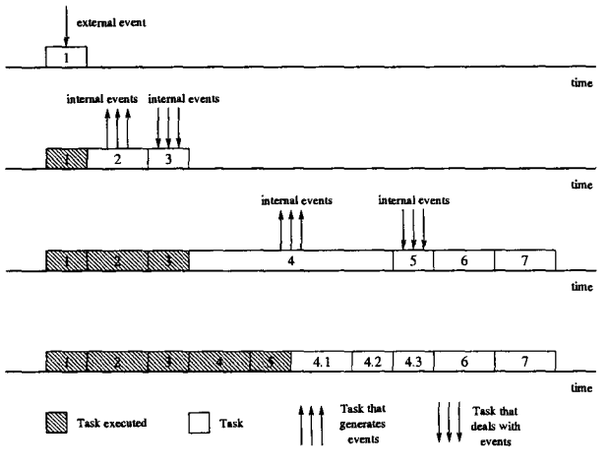

Figure 3: An example of a schedule

execution. This is useful because is a reserve of resources that allows to know early if the agent is going to have resources enough to execute the plan, and it simplifies the work of scheduling the tasks identified by the execution of task four.

3.3 Estimations of Execution Time of Tasks

An important problem is how to assign time to the tasks in the schedule, mainly because most of them are deliberative or represent complex actions in the environment (not a primitive action but a reactive module to achieve some state in the environment). We are not trying to answer this question here. Our architecture offers the means to apply the solutions proposed elsewhere. For example:

- Anytime algorithms: they can be interrupted at any moment and they guarantee to offer a result, although more time of execution will mean results with more quality. They have associated performance profiles that indicate the expected quality of results in function of the time of execution. Tasks can be constructed as anytime algorithms giving the tasks that add them to the schedule the flexibility of assigning them time to get certain quality. And tasks to deal with external events can be anytime algorithms so they can be executed the available time.

- Approximate processing: our architecture integrates very easily the possibility of having several methods to do the same thing. The task that deals with the event will choose according to resource constraints and quality requirements. We can also have several methods to deal with external events and use an heuristic in the scheduler to choose among them.

The control mechanism of AMSIA schedules sequences of tasks (and not individual tasks) and so, real-time artificial intelligence (Musliner et al., 1995; Garvey and Lesser, 1994) techniques can be applied.

Usually we will use estimations for the execution time of tasks. These estimations will be based in the history of the agent and can be changed dynamically. This is necessary both because the dynamism of the environment that can condition the time needed to do some task, and because, using learning techniques, the agent can learn to do certain tasks faster.

Of course estimations can be wrong. There are two protections to errors in the estimations of time of execution of tasks in our architecture:

1. Little deviations can be compensated with available time in the schedule or with execution time of other tasks of the same plan.

2. Greater deviations can be dealt by using monitoring. A great deviation will be detected and an external event will be generated to repair the schedule. Currently we do monitoring between tasks because we do not consider preemption. The tasks themselves must be build so that they have a maximum execution time (but see future work in section 6).

Moreover, AMSIA supports an hierarchical application of knowledge using internal events to identify tasks to work in other level of abstraction. This is interesting also because when the agent has a plan at a certain level of abstraction, it has resources (time) assigned to it. The execution of the tasks at that level generates tasks in a lower level that define more exactly the resource needs (possibly inside the resources previously reserved, see tasks 4, 4.1, 4.2, 4.3 in figure 2, although perhaps with some kind of adjustment). Then, as the agent spends more time in a plan, it has more exact idea of the resource requirements of that plan and, so, it is less probable that the agent had to abort the plan due to underestimation of resource requirements.

Also it is important that the reasoning model of the agent is incremental, the agent has a plan (schedule) and it works adding and removing pieces to that schedule. Resource estimations are not global, hence, they are easier to do and to compensate in case of error.

3.4 Conflicts in Resources Assignment

It is possible that, when the control mechanism of AMSIA tries to introduce a sequence of tasks in the schedule, there are not resources (time) enough to do it. To solve these conflicts, the control mechanism of AMSIA scores all the sequences of tasks. The score depends on the plan the sequence of tasks is trying to achieve, and the particular tasks that are part of the sequence. When there is a conflict, the control mechanism tries to free time in the schedule by removing the sequences of tasks with the smallest score and that are in conflict with the one that it is being introduced. External events are generated to signal

the removing of these sequences of tasks, and so, later it can be considered their re-introduction. This is an heuristic process, but it only happens when there are resource conflicts and it favors the most important plans.

4 Experimental Work

In this section we are going to show the results of an experiment developed to study the robustness of our agent in front of errors in the estimations of the duration of the tasks of the schedule.

We have implemented the proposed agent architecture modifying BBK (Brownston, 1995), a C++ implementation of the blackboard architecture for control (Hayes-Roth, 1988), and adding the mechanisms described in this paper. We have applied it to control a simulated robot (a modified version of the Khepera simulator (Michel, 1996)) that receives requests to carry out missions in the environment. The missions have the following characteristics:

- A deadline: each mission must be accomplished by the agent before its deadline.
- An importance: each mission has an associate importance. Not all the missions are of the same importance to the agent, in case of resource shortage it is better for the agent to abandon missions with low importance to favor the accomplishment in time of missions of higher importance.
- A destination: the environment presented by the simulator is a collection of rooms. Missions consist of going to a room (destination) and make a fault diagnosis and repair there. Information needed by the robot to do the diagnosis can be obtained only if it is in the destination room.

To operate in this environment and to successfully accomplish its missions the agent needs to implement several functionalities. It must be able to act: to move (using its two motors), and to repair faults. It must be able to sense: obstacles in its path, the state of a fault, and messages telling the agent the missions that it must accomplish. It must be able to *reason*: planning how to accomplish its missions, path planning for discovering how to go to its destinations, and diagnosis of faults (using an expert system). All this functionality is implemented as knowledge sources in our architecture. For example, the agent has a knowledge source for going from one point to another, this knowledge source controls the speed of the motors of the robot and attends to its sensors. Robot sensors offer raw data that must be processed by the knowledge source to deliver symbolic information.

First, we identify the factors that can influence in the performance of the agent:

1. Dynamism: the dynamism is configured in the simulator by two parameters:
   (a) missions dynamism: the ratio of appearance of new missions. Modeled by an exponential distribution with mean $\bar{t}_M$.
   (b) obstacle dynamism: the ratio of appearance of obstacles that can make more difficult or make impossible the accomplishment of some missions, modeled by an exponential distribution with mean $\bar{t}_O$. And the life of those obstacles, modeled by an exponential distribution with mean $\bar{t}_{OD}$.
2. Deadline: how is the deadline associated with missions. The deadline is modeled by an exponential distribution shifted to the right t_{MD} and with mean $\bar{t}_{MP}$.
3. Range of importance: the importance of missions is distributed uniformly between **0** and I_{max}.

The variables that we use to measure the performance of our agent in a certain interval of time are:

1. *Effectiveness* $= \frac{\textit{Score obtained by the agent}}{\textit{Total score offered to the agent}} \times 100.$

   where,

   $$score = \sum_{\text{missions accomplished}} (importance_{mission} + 1)$$

   Missions accomplished refers to those accomplished *before* their deadlines.
2. *Mission effectiveness* $= \frac{M_{aa}}{T_{oa}} \times 100$

   where, M_{aa} is the number of missions accomplished by the agent, and T_{oa} is the total number of missions offered to the agent.
3. *Importance effectiveness* $= \frac{M_{aahi}}{T_{oahi}} \times 100$

   where, M_{aahi} is the number of missions accomplished by the agent of the highest importance, and T_{oahi} is the total number of missions offered to the agent of the highest importance.

We wanted to measure the performance of the agent in stationary state, so we did preliminary experiments and use them to decide the time of the simulation (15000 seconds), the number of samples in each condition (5), and the suppressed samples to avoid the transitory state. Also we used the preliminary experiments to determinate interesting values of the factors that influence the performance of the agent in the experiment. The values chosen for the experiment are shown in table 1.

The categories in table 1 correspond to the following values (in tenths of second) of the parameters in the simulator:

Factor	values
Mission dynamism	high, low
Importance range	medium
Deadline	big
Obstacle dynamism	low
Time estimation	high, medium, low, very_low

Table 1: Independent variables in the experiment

Missions dynamism = $high \Rightarrow \bar{t}_M = 275$
Missions dynamism = $low \Rightarrow \bar{t}_M = 600$
Importance range = $medium \Rightarrow I_{max} = 5$
Deadline = $big \Rightarrow t_{MD} = 3000$ and $\bar{t}_{MP} = 10000$
Obstacle dynamism = $low \Rightarrow \bar{t}_O = 1000$ and $\bar{t}_{OD} = 100$

Time estimation *medium* means that the average execution time of each task (measured in the preliminary experiments) is used as estimation of the expected execution time of that task. Time estimation *high* means that estimations 15% over the average values are used, *low* means 15% under the average values, and *very_low* 25% under the average values.

The results of the experiment are shown graphically in figure 4, where we have separated the situation with dynamism high and low. An analysis of variance shows that the factor *time estimation* has significant influence in the three dependent variables: effectiveness (for missions dynamism=low F=4.5673, P=0.0171; and for missions dynamism=high F=4.4002, P=0.0194), missions effectiveness (for missions dynamism=low F=5.2067, P=0.0031; and for missions dynamism=high F=4.0605, P= 0.0253), and importance effectiveness (for missions dynamism=low F=7.0520, P=0.0031; and for missions dynamism=high F=7.9768, P=0.0018).

The shape of the curves in figure 4 is what we expected. The architecture achieves a profit of its time estimations, hence, the effectiveness measurements have a maximum at one point, and go down at both sides of that point. If time estimations are too high, this results in that missions which could have been tried are not, because the agent thinks that it has not enough resources. If estimations are too low, the agent tries missions that finally are not achieved because of lack of resources (or they are achieved after their deadlines).

However, when the missions dynamism is low, the maximum of effectiveness and mission effectiveness is not achieved using as time estimations the average time of execution of tasks, but a lower value. The reason for this is the flexibility that the agent architecture has to deal with errors in time estimations. If missions dynamism is high the agent architecture has more problems to deal with error in time estimations, there are few time available in the schedule and the missions in it are of high importance.

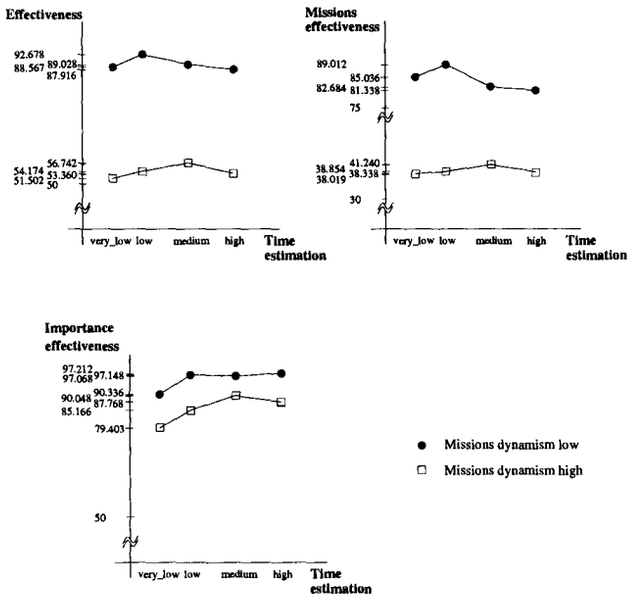

Figure 4: Results of the experiment

The only solution left is to use tasks with less quality (but that need less time) to achieve the missions. The problem is that these tasks sometimes are going to fail preventing the achievement of the mission.

We can conclude the following from this experiment:

1. The estimation of execution time of the tasks has influence in the performance of the agent architecture. Hence, a better estimation improves the performance. However, errors in estimations doesn't provoke an abrupt fall in performance because the mechanisms that the architecture has to deal with these situations.

2. As the missions dynamism (the number of missions that the agent is facing) is decreased, it is better to be optimistic in time estimations. These allows the agent to try more missions, and it has enough flexibility to deal with situations of error in the time estimations. If mission dynamism is increased, time estimations must be more exact to get higher performance. Notice that the agent architecture can calculate dynamically the estimations of the time of execution of its tasks; for example, it can be more or less conservative according to the perceived missions dynamism.

5 Related Work

Plans or schedules have different roles in different agent architectures.

Reactive architectures, as the subsumption architecture (Brooks, 1985), don't use plans, and so, it doesn't

seem easy, using this kind of architecture, to build an agent to fulfill certain real-time requirements of high level objectives.

Hybrid architectures as InteRRaP (Fischer et al., 1995; Müller, 1996), TouringMachines (Ferguson, 1992), or RemoteAgent (Gamble Jr. and Simmons, 1998), use a reactive module to ensure the security of the agent in front of events in the environment that can mean a risk to the agent. The reactive layer offers actions quickly to ensure the survival of the agent while the deliberative layer/s makes plans to achieve the high level objectives of the agents, negotiate with other agents, etc. These plans are built off-line and, afterwards, executed. But deliberative actions are not scheduled themselves and so it is difficult to offer guarantees of global real-time requirements (specifically, it is difficult to adapt the reasoning to real-time constraints). Nonetheless, the idea of a reactive layer to manage the direct interaction with the environment seems a good one (see future work in section 6).

IRMA (Bratman et al., 1988; Pollack et al., 1994) is a deliberative architecture thought to deal with resource-boundedness in the reasoning of the agent. The main procedure to do this is to use the plan of intentions that defines what the agent intends to do as a guide for the reasoning of the agent, limiting in that way its possibilities of reasoning. Options for deliberation are filtered to avoid loosing much time in deliberation. The idea is that the less promising options are discarded faster with the filtering process than if the agent deliberate about them. Options incompatible with the current plan of intentions are filtered this way. But, to keep openness in front of external events, an override process allows options incompatible with the current plan but highly promising to pass the filtering process to let the agent deliberate about then (about changing the current plan). Much of the work with IRMA is to show the advantages of the filtering mechanism for a resource-bounded agent. Notice that in our agent architecture the global schedule effectively directs where the agent is going to spend its reasoning resources. The role of the filtering-override processes is played by the scheduler and how it deals with external events. But reasoning activity is scheduled and so the agent has the flexibility of choosing among different reasoning methods according with the circumstances, of deciding when to deliberate and how about a particular event, and of integrating several objectives and divide the resources among them.

Our work differs from recent advances in planning and scheduling (as for example in Chien et al. (1998)) in that our main aim is in the integration of planning and execution. In fact, in our system, planning is an activity as any other and must compete for the resources of the agent, the result of this activity are plans that guide the future behavior of the system. Plans keep its causal structure and can be analyzed or modified at any time, but the schedule is highly committed to simplify control operations and because replanning is based on the plans, not on the schedule. AMSIA can adapt its planning activity to the circumstances (for example it can choose a predefined plan because there is not time to generate a better one).

As it was mentioned before, techniques such as anytime algorithms (Garvey and Lesser, 1994) and how to build a solution to a problem using a number of anytime algorithms (Zilberstein, 1996), and approximate processing (Lesser et al., 1988) and how to build a solution to a problem based on different methods of different tasks (Garvey and Lesser, 1993), are easily integrated in AMSIA.

6 Conclusions and Future Work

In this paper we have analyzed the role of a schedule of activity to guide the behavior of an agent. This agent must use different reasoning methods under real-time requirements associated with its high level objectives.

All the activity in AMSIA is explicitly scheduled as a way of controlling the use of resources. Also, the activity to choose a line of action is separated from the activity of following that line of action and offering new possibilities for future action. We believe this is an important property for agents that must fulfill real-time requirements. The line of action focuses the attention of the agent that, independently, considers changing that line of action, i.e., it keeps its opportunism. In other work (Soto et al., 1997, 1998) we have explored the idea of separating these activities in hardware. In this paper we explore the division of these activities in time. To do so, the activity needed to choose a line of action must be included as a series of tasks in the schedule of the agent. A mechanism (external events) is added to deal with unexpected events, i.e., to include tasks in the schedule to consider what to do in front of those events.

Also, there are options for AMSIA that we want to explore:

- The use of a preemptive scheduler. This means that we need to be able to interrupt the execution of tasks. The problem is that it is not easy to keep the consistence of the knowledge in the blackboard when a reasoning task is interrupted. There are solutions as using sections of code where an interrupt is impossible to make changes in the knowledge state of the system.
- We have used our agent architecture to control a simulated robot. In a real environment we will need a reactive layer to augment the reactivity in face of contingencies.
- We want to extend the information that is kept in the schedule. For example, it will be interesting to register other temporal constraints for the execution of tasks. Although this will complicate the heuristics used in schedule construction, this is not a critical problem because this activity is also scheduled.

Acknowledgements

The author is grateful to Mercedes Garijo and Carlos Ángel Iglesias for useful comments on this work. The author also wishes to thank the anonymous reviewers for useful comments on the abstract of this paper.

References

Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computacional Intelligence*, 4:349–355, 1988. http://bert.cs.pitt.edu/pollack/distrib/guide.html.

Rodney A. Brooks. A robust layered control system for a mobile robot. Technical Report A. I. Memo 864, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, September 1985. http://www.ai.mit.edu/people/brooks/papers.html.

Lee Brownston. *BBK Manual.* Knowledge Systems Laboratory, Stanford University, September 1995. Report No. KSL 95-70.

Norman Carver and Victor Lesser. The evolution of blackboard control architectures. Technical Report UM-CS-92-071, University of Massachusetts, Amherst, 1992.

Steve Chien, Benjamin Smith, Gregg Rabideau, Nicola Muscettola, and Kanna Rajan. Automated planning and scheduling for goal-based autonomous spacecraft. *IEEE Intelligent Systems*, September/October 1998.

Daniel D. Corkill. Blackboard systems. *AI Expert*, 6(9), September 1991. Also as *Technical Report, Blackboard Technology Group Inc.*

Innes A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents.* PhD thesis, University of Cambridge, October 1992.

Klaus Fischer, Jörg P. Müller, and Markus Pischel. Unifying control in a layered agent architecture. In *Proceedings of the IJCAI 1995 Workshop on Agent Theories, Architectures, and Languages*, 1995. Also as DFKI GmbH Technical Memo RR-94-05, http://www.dfki.uni-sb.de/mas/papers.

Edward B. Gamble Jr. and Reid Simmons. The impact of autonomy technology on spacecraft software architecture: A case study. *IEEE Intelligent Systems*, September/October 1998.

Alan Garvey and Victor Lesser. A survey of research in deliberative real-time artificial intelligence. *Real-Time Systems*, 6(3):317–347, May 1994.

Alan J. Garvey and Victor R. Lesser. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), November/December 1993.

Barbara Hayes-Roth. A blackboard architecture for control. In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 505–540. Morgan Kaufmann Publishers, 1988.

Barbara Hayes-Roth, Karl Pfleger, Philippe Lalanda, Philippe Morignot, and Marko Balabanovic. A domain-specific software architecture for adaptative intelligent systems. *IEEE Transactions on Software Engineering*, 21(4):288–301, April 1995.

Victor R. Lesser, Jasmina Pavlin, and Edmund Durfee. Approximate processing in real-time problem solving. *AI Magazine*, 9(1):49–61, Spring 1988.

Olivier Michel. *Khepera Simulator Package version 2.0.* University of Nice Sophie-Antipolis, March 1996. Freeware mobile robot simulator downloadable from http://wwwi3s.unice.fr/~om/khep-sim.html.

Jörg P. Müller. *The Design of Intelligent Agents, A Layered Approach*, volume 1177 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, Berlin, 1996.

David J. Musliner, James A. Hendler, Ashok K. Agrawala, Edmund H. Durfee, Jay K. Strosnider, and C. J. Paul. The challenges of real-time AI. *IEEE Computer*, January 1995.

Karl Pfleger and Barbara Hayes-Roth. An introduction to blackboard-style systems organization. Technical Report KSL-98-03, Knowledge Systems Laboratory, Stanford University, 1997. http://www-ksl.stanford.edu/publications/index.html.

Martha E. Pollack, David Joslin, Nunes Arthur, Sigalit Ur, and Eithan Ephrati. Experimental investigation of an agent commitment strategy. Technical Report 94-31, Department of Computer Science, University of Pittsburgh, June 1994. http://bert.cs.pitt.edu/~pollack/distrib/tileworld.html.

Ignacio Soto, Manuel Ramos, F. Javier González, and Ángel Viña. Arquitectura multiprocesador para sistemas inteligentes adaptativos. In Martín Llamas, José J. Pazos, and Manuel J. Fernández, editors, *Actas de las V Jornadas de Concurrencia*, pages 161–175, Vigo, Junio 1997. Servicio de publicaciones de la Universidad de Vigo. (In Spanish).

Ignacio Soto, Manuel Ramos, and Ángel Viña. A control mechanism to offer real-time performance in an intelligent system. In Ethem Alpaydin, editor, *Proceedings of the International ICSC Symposium on Engineering of Intelligent Systems (EIS'98).* ICSC, February 1998.

Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996. http://anytime.cs.umass.edu/.

Collaborative Personal Agents for Team Working

Simon Thompson; Brian Odgers[1]
Intelligent Business Systems Group
Advanced Communications Research Department
BT. Advanced Communications Technology Centre
PP12, MLB1
Ipswich, UK, IP5 3RE
44 1473 605531
simon.2.thompson@bt.com;brian.odgers@bt.com

Abstract

This paper describes a workforce management system, implemented with intelligent agents, that we call SAMBA (Software Agents for Mediating Business Activities). Workforce management is cast as a collaborative information system, implemented using software agents. We will distinguish this approach from the traditional "management by scheduling" view of controlling the operational activities of large workforces. The commercial and practical motivations for this change of perspective are discussed. A detailed example of the use of SAMBA is presented. The technical details of our system are reported: an *extended request* and *constrained contract net* protocol are motivated and described as is the use of the *Zeus Toolkit*, a *mediated architecture*, and *deliberative collaborative agents supporting mixed initiative reasoning*.

1. Introduction and Overview

SAMBA is an agent-based system that enables distributed teams to collaborate for business applications such as team information, overtime and work rosters. An intermediary agent is used to facilitate the collaboration of human team members using personal agents, with their human manager.

In recent research, employee performance has not only been linked to skill levels but also to motivation [14]. If enterprises could improve their employees' teamwork and motivate their workforce, the result would be a better working environment with increased performance, and therefore better customer satisfaction.

Within SAMBA employees, managers and the enterprise are represented by autonomous collaborative agents. An intermediary agent is introduced to act as a go-between for the employees, managers and the enterprise's information infrastructure. The managers are able to add localised business rules to the intermediaries, covering for example geographic or weather conditions, as well as team issues. When an agreement is reached it may be necessary to pass this information, or other results of the agreement, back to the enterprise to be used in its next evaluation cycle. The intermediary can also act as a "virtual coach" passing information to appropriate employees when required.

In the process of developing these ideas we have developed some novel technology, specifically a mediated agent level architecture and some innovative interaction protocols; this paper reports these innovations.

We also report a detailed example of this technology in a system that allows field engineers to book and revise their leave and overtime schedules. In the next section we will discuss the motivations for our approach to managing workforces.

1.1 Commercial Motivations

In a customer service environment engineers are required to install and maintain the enterprise's services at consumers' homes or businesses. This leads to a distributed workforce working both individually and as a team. As technology has increasingly been used to obtain efficiency gains the engineers have become more isolated, receiving job information through hand held devices or laptops, with little contact with their fellow workers.

[1] We would like to acknowledge the efforts of (in alphabetical order) Paul O'Brien, Mark Buckland, Dean Jones and John Shepherdson who committed much time and effort to the review and development of this paper.

SAMBA has been developed to consider how engineers in the "field" can work more as a team, increasing morale and, as a by-product, increasing customer satistfaction [11], without sacrificing the efficiency gains derived from the highly successful dynamic scheduling systems already deployed.

In any human-human interaction trust is a key issue and with new stricter legislation for companies concerning their employees [2], companies can no longer impose working conditions and practices on thier employees. Because of this corporations are now realising that high quality management systems enabling closely formed teams with strong team identities and structures of trust and obligation between the members can increase motivation.

By engineering social commitments and obligation between the enterprise and the employee we hope to increase the loyalty of the workforce and improve our response to our customers, while building on the functionality of previous innovations.

1.2 Related Work

Three particular areas of work are relevant to the research described here.

- Systems developed to support workforce management, concentrating on the scheduling or enactment of a particular business process.
- Agent technology for managing business processes.
- Work investigating methods of co-ordinating teams of agents in order that they can achieve a goal.

1.2.1 Workforce Management Systems

For many years businesses have considered the issues of supporting employees as they do their work. Process centric workflow systems such as Oracle workflow [18] or Plexus workflow [17] frame the workforce management problems in terms of managing the movement of items of work through a business process. Alternatively products such as Freeflow [1] address this requirement by providing flexible support for workers' activities rather than enforcing the process model. Freeflow lacks, however, facilities for work negotiation and team-building required for effective support of distributed team-work. It also lacks means for managerial influence and oversight. Indeed, this is a weakness in most workflow systems; using a single "correct" process model controlled by a single person. Collaboration while planning a business process is only supported by the research system Regatta VPL [22], and its commercial offshoot TeamWARE Flow. VPL and TeamWARE allow business professionals to define and change "their" section of the process (their perspective) by manipulating a visual representation.

Workforce management systems have often employed scheduling technology as the basis of their approach to managing employees. Dynamic scheduling [12] is used within the BT operational support systems (OSS's) In order to manage the operational workforce, and is estimated to save the company $150,000,000 annually. Dynamic scheduling is currently the subject of one of the technical co-ordination units (TCU's) in the PLANET [16] network of excellence funded by the EU. However despite their extraordinary success dynamic scheduling systems fail to capture the social and "soft" aspects of an ideal work management system; this is the focus of the SAMBA project.

1.2.2 Process Management Agents

As we have stated SAMBA is an agent based project. In the field of agent technology work continues to develop agent based collaborative systems. Relevant work includes the ADEPT business process management system [10] and the agent based workflow system developed by Harker and Ungar in [8], considered business to business collaboration through agents constantly negotiating to achieve their business goals. The personal assistant paradigm has become a much used flagship application for agent co-ordination [4].

While some of these projects consider the co-ordination of information [25][9] others consider the co-ordination of peer groups [26][27]. Neither systems co-ordinating information nor those co-ordinating peer groups consider the issues of complex teams with both peer to peer *and* command and control management structures. And although learning packages such as Tapped In by SRI [23] have interactions between both teacher and student, and student to student they are based around white boards as a medium for information exchange.

1.2.3 Agent Teams

Recently work on the construction of teams of agents that work together to achieve goals has become a topic of research in the agent community [24]. The work on agent teams, for example the Robo-cup competitors that have been so popular at agent and AI conferences in recent years, addresses only the communication and interaction of the agents within an artificial society, however.

In contrast, the work reported here investigates software agents that must interact with each other, *and* human users, within a society with rules and parameters imposed by a human system. The objective of SAMBA is not to construct a team of software agents, rather it is to construct, develop and support, a team of humans, by using software agents.

2. Example Scenario

Ten workshops involving engineering teams and their managers were held to elicit the requirements for the system. One of these was the automatic resourcing of overtime, and we have chosen this as an appropriate

demonstrator scenario and as the example agent service that we will discuss in this paper.

Due to the new European Union Working Directive [2] employers cannot force employees to work more than forty-eight hours a week. Therefore a manager is required to request that the employee do overtime. Even if an engineer has asked to be considered for overtime the manager still has to confirm with the engineer that they are still available for overtime. At anytime in this process the engineer can refuse to work overtime if they have worked more than the specified forty-eight hours. This presents a considerable problem for the managers, and a significant element in their workload appears to involve the resourcing of overtime rosters.

In the SAMBA system, the intermediary takes the resourcing request from the manager agent and sends a set of requests for overtime to the agents who have shown an interest in doing overtime that day. If an employee is not able to comply with the request they may try to solicit one of their "friends" to do the overtime for them by sending a call for proposal to their associates. The engineers can form an agreement which can be returned to the intermediary. The agreement between two engineers (engA and engB) cannot just be simple off-loading of work as it involves personal information. The engineer being solicited (engB) must first ask the intermediary if he is permitted to take the overtime from the initiating engineer (engA) before giving a response. By separating the negotiation into two phases, it possible for the engB to conceal from engA that the intermediary deemed him unfit to do the work.

Note, that engA who contracted to do overtime with the system is deemed to have responsibility for arranging a replacement resource if unable to meet their obligation. This is a significant point in this example because the explicit adoption of obligations toward the enterprise and toward each other is one of the key ideas that we have to foster a team spirit and increased co-operation.

The intermediary evaluates the replies using a set of global business rules along with local rules defined by the manager. The manager may define the order in which the employees are awarded overtime from a set of functions defined by the intermediary, such as, first person to respond gets the overtime or overtime is given to the employee with the highest ratings.

If the intermediary finds a conflict of interest it cannot resolve, the unresolved issues are passed to the manager for final evaluation. Once this stage is completed the intermediary sends appropriate overtime accepts and rejects to the field engineers and an overtime overview to the manager and any other systems that require this information.

3. Technical Details

In this section we describe the technical elements of the SAMBA system. We chose to implement SAMBA using the abstraction of intelligent agents for a number of reasons:

- SAMBA is inherently decentralised, and agents are closely associated with decentralised problems
- Several of the examples of systems that we would like to construct using SAMBA were physically distributed: again pointing to an agent based approach
- It was apparent by inspection that a number of autonomous and discreet computational entities existed within the architecture

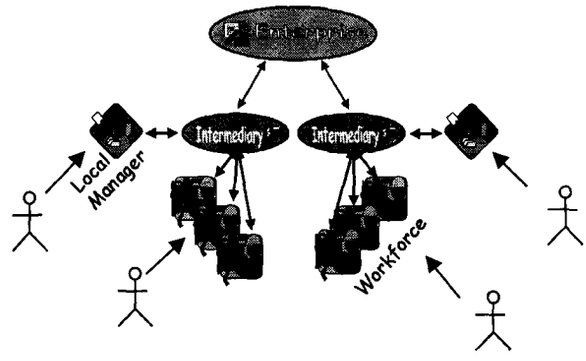

Figure 1. The intermediated architecture used by SAMBA.

In Figure 1 we show a schematic of the mediated agent based architecture that we have developed for SAMBA. There are three types of agents in SAMBA. It might appear on first inspection that systems like SAMBA should be decomposed at the agent level into "manager agents" and "worker agents" (or "team agents" as we like to call them). However, further analysis shows that in reality it is necessary to implement "mediator agents" as part of the architecture if it is to be practical for real world use.

3.1 Mediated Architecture

The system uses an intermediary as a go-between for the engineers, their managers and the corporation as a whole. The agents act either automously or semi-autonomously, according to the engineer's wishes. An agent can gather information, react to requests and even attempt to collaborate with "friends" to share work. The intermediary acts as an autonomous neutral observer, reasoning over the team's goals and wishes and, if possible, providing a fair judgement. The intermediary also holds a representation of the legal obligations of the enterprise.

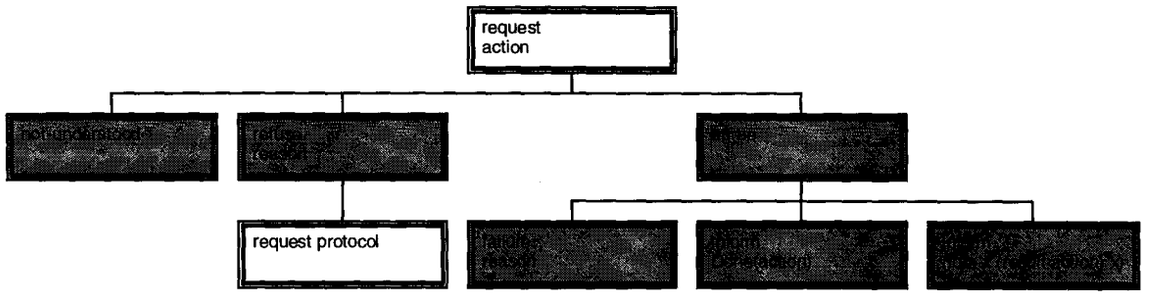

Figure 2.The extended request protocol. The protocol is read from top to bottom, with each branch representing a separate messaging possibility.

In effect intermediary acts as a facilitator between the goals of the manager and the personal requirements of the engineers and the goals and objectives of the larger enterprise. This enables the engineer's working environment to be more agile and flexible by enabling the engineers to respond to requests from the intermediary via their automous agents. The engineers can use their agents to formulate agreements which can then be validated by the intermediary, enabling the engineers to work more closely as a team within legal requirements. The intermediary can act as an auditor considering engineer-engineer and engineer-manager interactions to make sure they are within the constraints of corporate culture.

In [19] it is argued that intermediaries in an electronic medium were obsolete. The SAMBA system was implemented using an intermediated architecture because we were constrained by the characteristics of the service applications that were required. In the example we discuss in section 2 the global behaviour of the system is determined by goals that are set on the team derived from the model of the business used by managers higher in the organisation, and therefore operating at a higher level of abstraction. This requires a knowledge based decomposition of the goals into sub-goals that can be executed by team members, and the re-composition of the achievements of the delegated processing into the requested goal, which we have implemented within the intermediary.

Also, for our purposes the intermediary can be considered to instantiate a knowledge base of constraints on the interactions of the agents in the team. As individual agreements are made by team agents, and sent to the intermediary, the scope of the agreements that can be made in the future is gradually constrained. It is important for the intermediary to prevent the schedule that is derived in this way from becoming over optimised because over optimised structures tend to be very brittle [7]. If the knowledge of agreements made in the system so far were to be decentralised, and held on the individual agents in the system, it is likely that the communication overhead required to obtain sufficient information to analyse the risk associated with the state of the system, would be prohibitive.

The use of a mediated architecture was one of the drivers for the development of some novel interaction protocols that are described in the following section.

3.2 Novel Protocols

The hierarchical commercial environment that systems using the SAMBA architecture must operate within is very different from the peer-peer environments that agents engaged in distributed problem solving [21] or agent applications such as meeting scheduling or e-commerce tend to operate in.

The hierarchical nature of the operating environment required that we construct some novel interaction protocols. We call these protocols "Two tiered" because they require both the permissions' of the agents engaged in the agreement and the authority of a third party (in this case the intermediary) to be granted. These protocols can also be seen as mechanisms for agents to obtain a relaxation of constraints imposed on their problem solving behaviours by other agents.

Two protocols were implemented. In section 3.2.1 we describe the *Extended Request Protocol* (see Figure 2) which we implemented for requests between peer agents that share a superior which has ordered some action. In section 3.2.2 we describe the *Constrained Contract Net Protocol* (see Figure 3) which we implemented in order to provide an intermediated contract net.

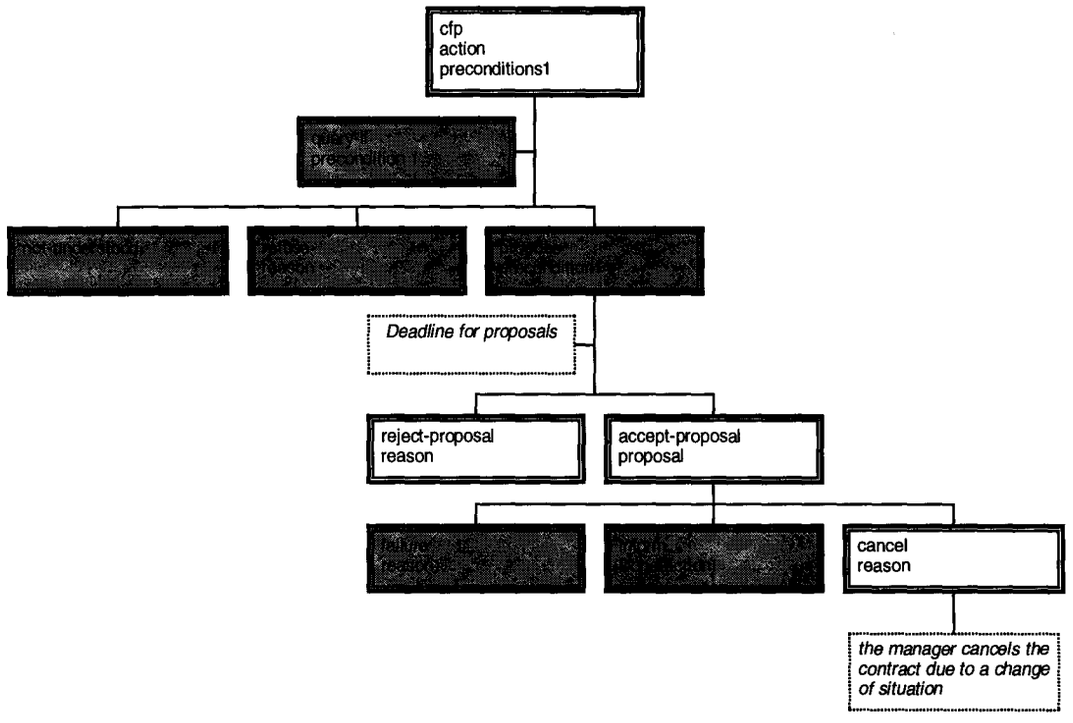

Figure 3. The Constrained Contract Net Protocol

3.2.1 Extended Request Protocol

In the environment of collaborative interface agents using an intermediary, the intermediary may not be able to resolve all the issues raised within a request. In this case, the intermediary will reply with a set of relaxation constraints or set of alternative actions. The requesting agent can then choose the most appropriate

action and initiate a new request within the same context. This extension to the classic request protocol [3] is shown in Figure 2. This may seem similar to the contract-net protocol [20] but contract-net is a delegation protocol between a "consumer" agent and a community of "suppliers". This is not the relationship that exists between the agents in SAMBA.

Extended Request captures the "worker" to "manager" relationship of the agents, and retains the context of the iterated request that is needed for the intermediary to decide the degree to which it is prepared to relax the constraints on the agreements already made.

In the overtime arranging scenario described in section 2, the interaction between the manager and the intermediary is based around the extended request protocol. This protocol enables the manager's agent to request the intermediary to perform some action, and the receiving agent to respond that the action has been performed with some additional information or that the action cannot be performed and a reason. In the case of the request being refused the intermediary will pass the manager's agent a set of constraints as part of the reason. These constraints can be used to relax the rules imposed by the system. The manager's agent considers these constraints and sends another, amended, request to the intermediary within the same message context of the original request.

3.2.2 Constrained Contract Net Protocol

The Constrained Contract Net Protocol is based on a contract-net tied in with a query-if protocol, see Figure 3. Whereas the contract-net is performed between "suppliers" and "consumers" here a third party, the intermediary, must be involved. Essentially this is because the constraints that must be reasoned over in order for the contract to be made, must in this scenario, be distributed across several different agents in order to protect the privacy of team-members and retain the control privileges of management. If the proposed "supplier" agents do not have the privileges to provide an instant response to the proposal they must request the appropriate privileges from the intermediary.

The constraints that dictate the finalisation of the contract are held on separate agents to protect privacy. In the example discussed in section 2 an engineer, engA, who has contracted to do overtime, but due to circumstances cannot, can try to acquire overtime from a peer. This is initiated via the contract-net protocol.

The engineer's agent's two options are to reply directly to the proposal, or seek confirmation on the feasibility of the proposal from the intermediary. If the former route is taken the proposing agent may be able to deduce information about its peers, by taking the latter option the proposed agent can find out the feasibility of the proposal before making a decision thus greatly reducing the initiating agents likelihood of deducing its personal details. In this way two-tier negotiation can reduce the likelihood of the initiating agent deducing information about its peers, by enabling the peers to only return information that has been validated by the third party.

3.3 Agent implementation

Originally SAMBA was implemented using Java and CORBA [12]. Our experiences showed us that the Team Agents, Manger Agents and SAMBA Intermediary Agents would all be required to exhibit both reactive and goal based reasoning over a common knowledge base. Because of this we chose to use the Zeus toolkit [28] for the current implementation. Although originally developed as a proprietary research tool by BT, Zeus is now an open source, publicly available toolkit for the development of multi-agent systems.

In the rest of this section we describe the agent level architecture of Zeus so that the reader can see what tools a SAMBA agent has at its disposal for action and reasoning in its world. In section 3.4 we describe which of the mechanisms in Zeus we used to implement the reasoning in SAMBA. In section 3.5 we describe other functionality that is specific to SAMBA.

Zeus Agents have the following components:

- A Resource Database (ResourceDb) which is used to store a set of "facts" that the agent "knows".
- A mailbox that they use to send and receive messages.
- A co-ordination engine that defines the problem solving behaviour that the agents use in order to construct and execute their plans
- A planner & scheduler which is used to define the set of tasks that the agent must complete in order to achieve a goal and to schedule and control the execution of those steps. The planner operates under the control of the co-ordination engine.
- A rule engine that executes chains of CLIPS [6] style rules.
- An "agent external" which is a piece of procedural (Java) code which interacts with the above components in order to achieve behaviours that are beyond their scope as implemented.

We discuss each of these components in detail in sections 3.3.1 to 3.3.5 below. We have drawn on the material in the Zeus distribution documentation [28] and [15] as the source for this discussion, as well as our first hand experiences with the toolkit.

In addition it is worth noting that each Zeus agent has the address of (and therefore can communicate with) several local utility agents that provide infrastructure for the agent system. In the language of FIPA these agents define the Zeus "agent platform". These utiltity agents are

- The Agent Name Server (ANS), which provides a "white pages" service. The ANS provides mechanisms for agent registration and de-registration. If requested this agent can provide the address to any agent that is registered with it.
- The Facilitator agent, which provides a "yellow pages" service location mechanism. The facilitator polls all Zeus agents registered with the ANS requesting lists of the services that they are prepared to offer. It will then provide a list of agents offering particular services on demand.
- The Visualiser agent, which provides visualisation, agent platform interaction and application debugging services to the platform user.

3.3.1 The ResourceDb

The ResourceDb in a Zeus agent stores the agents set of current beliefs about both its internal state and the state of the world it is operating in.

3.3.2 The Mailbox

Zeus agents implement communication via a FIPA-like agent communication language. This language uses a number of performatives, that one agent sends to another in an attempt to change the state of the receiving agent. For example; agent A may have some data that agent B has requested. Agent A will send agent B a message using an *inform* performative. The semantic of this transaction is that A wants B to become "informed of" the data. Specifically A wants B to put the data into its ResourceDb, which is the default behaviour for Zeus agents on receipt of an *inform*.

The arrival of a message in the Mailbox of an agent triggers a number of actions within the agent. As we noted above, default behaviours encoded in the agent in the form of protocols can be triggered. Examples of this are the ResourceDb update behaviour triggered by the arrival of a message and the contract net protocol behaviour triggered by the arrival of a message containing a *Call For Proposals* (CFP) performative. Another important side effect of the arrival of a message is the dispatch of events in the Zeus External.

These events can be used to trigger other behaviours as discussed in 3.3.5.

The content of messages in Zeus can be strings, serialized Java objects or Zeus Facts. Much of the content that is exchanged in SAMBA is coded as a string containing an XML document. This is because complex data structures can be encoded in XML and powerful, easy to use and flexible parsers are available on open source terms from a number of sources. (we use XERCES from www.apache.org). We have found that the non XML encoded content messages that we have actually implemented in SAMBA were exclusively part of primitive call and response type exchanges.[2]

3.3.3 Co-ordination and Agent Level Planning

Zeus utilizes two components to achieve goal-based behaviour.

1. The co-ordination engine is used to devise and execute the sequence of actions that the agent needs to execute in order to construct a plan,
2. The planner & scheduler is used to construct that plan and execute it.

This is a rather different approach to that which a centralised planner requires, and stems from the fact that in order to satisfy goals Zeus agents must interact with one another.

Problem solving behaviours are represented in Zeus as recursive transition network graphs that are traversed from their start node until a terminal node is reached. The nodes in the graph are code fragments that are interpreted and executed by the agent in order to generate behaviour actions. The execution of the nodes yields one of three results:

- The node can return OK in which case the node processing has succeeded and traversal can continue to the next node.
- The node can return WAIT, which is associated with either a timeout value or a message-reply-key (essentially a conversation context indicator). In this case the processing of the node will be suspended until the timeout expires, or a message with the associated key is received. This allows the co-ordination engine to query some other agent, or resource, for the possibility of the execution of some behaviour and to prevent further action until a reply has been received.
- The node can return FAIL, in which case the agent backtracks from the node by calling a reset() method that undoes the actions taken while execution was attempted. Any untried arcs on the preceding node are then traversed. This process continues recursively until there are no more arcs to traverse on the start node, and the graph traversal fails.

A detailed account of this behaviour and the default goal-processing graph in the Zeus distribution can be found in [15].

3.3.4 Rule Chaining

CLIPS rules can be implemented in Zeus so that the presence of facts in the ResourceDb triggers either chains of deduction or side-effects. For example a rulebase might contain rules :

```
{ruleOne{condition
  ?factid <-
       factTypeName
             (attributeOne            val1)
             (attributeTwo val2)}
{action
  assert (otherFactTypeName
             (attributeOne val1)
  assert (thirdFactTypeName
             (attributeTwo val2))
}}
```

and

```
{ruleTwo
{condition
       ?factid    <-    otherFactTypeName
       (attributeOne val1)}
{action
sendMessage
       (content
             (factTypeName
                   (attributeOne val2)),
             type (Inform),
             receiver (otherAgent))
}
```

These rules have two meanings within the context of the Zeus interpreter . Firstly they have the straight forward condition-> action meaning which implies that the presence of factTypeName (val1,val2) will lead to otherFactTypeName(val1) and thirdFactTypeName(val2) being added to the ResourceDb; and that the presence of otherFactTypeName(val1) will lead to a message being sent. Secondly, the other semantic is that if ruleOne is fired then on the next evaluation cycle of the agent ruleTwo could be fired. Only one rule will be fired in any reasoning phase, and numeric priorities can be associated with rules to decide which should be fired if more than one pre-condition is matched.

3.3.5 The "agent external"

The "agent external" in a Zeus agent is a set of Java classes that are able to access an AgentContext object.

[2] We think that this is interesting because it implies that messages that are part of more complex interactions tend to have higher information levels.

This object provides references to proxy objects for each of the major components of the Zeus agent. These proxy objects provide methods that allow client code to listen for events occurring within the agent and then to take procedural actions. Examples of this are the FactAdded events, which are fired when new facts are added to the ResourceDb, and Message events which are fired when Mail arrives or is sent. It is possible to set goals for the agent; create and add facts to the agent and explicitly send messages via these interfaces.

3.4 Decision Making In SAMBA Agents

As we described in sections 3.3.3 to 3.3.5, there are three basic mechanisms that can be implemented in a Zeus agent that enable it to make decisions within its environment.

- The agent can utilise a knowledge base, reasoning over it using the RETE [5] engine in Zeus to decide on some action based on the state of the world according to its ResourceDb.
- The co-ordination engine can use the planner & scheduler to decide on, and execute a behaviour sequence defined as the execution of a set of tasks.
- The "agent external" element, written in Java in the current implementation of Zeus, can be used to trigger sequences of procedural behaviour. These behaviour sequences are typically triggered by the arrival of some new element in the agent's world (a new message or fact for instance) and execute without reference to the sensors of the agent.

In this section we will describe how we used these mechanisms to achieve the behaviours that we required from the system.

3.4.1 Controlling the Agent's Activities

In order to describe how SAMBA controls the activities of all of its constituent agents in order to achieve the behaviour that is required, we will describe one particular use-case of the system and the interactions and exchange of messages that result from the implementation that we have chosen.

Table 1. Tasks in SAMBA Overtime Example

Agent type	Tasks
Intermediary	supplyRosterForRatification produceRoster
Manager	ratifyRoster
TeamAgent	reqOvertime
OSS	supplyWorkRequests

For the overtime scenario described in section 2 the reasoning of the agents in the case that the OSS (or enterprise) requires a resourced overtime schedule was developed as follows.

Table 1 shows the distribution of primitive tasks among four types of agents in the system. The interactions of the agents as they reason about the task is shown in Figure 4.

In order to decide on how overtime should be allocated to the team member the first action required is that all team members should enter their preferences for overtime dates and times into the system. This is achieved using the GUI shown in Figure 6 and discussed in section 3.5. This data is dispatched to the intermediary where it is stored. The OSS (the agent wrapping the operational support system shown as "Enterprise" in Figure 1) may then initiate a scheduling episode by attempting a goal (set from Java and initiated by pressing a button on a simple GUI). The goal set is *"achieve Schedule"*, but this cannot be satisfied by any of the tasks that OSS holds, so its co-ordination engine issues calls to other agents to see if any of them can provide a schedule. The task *produceRoster* held by the Intermediary can satisfy *Schedule*, but has a precondition *WorkItemList()*. The task *supplyWorkRequests* in the OSS can provide a *WorkItemList()*.

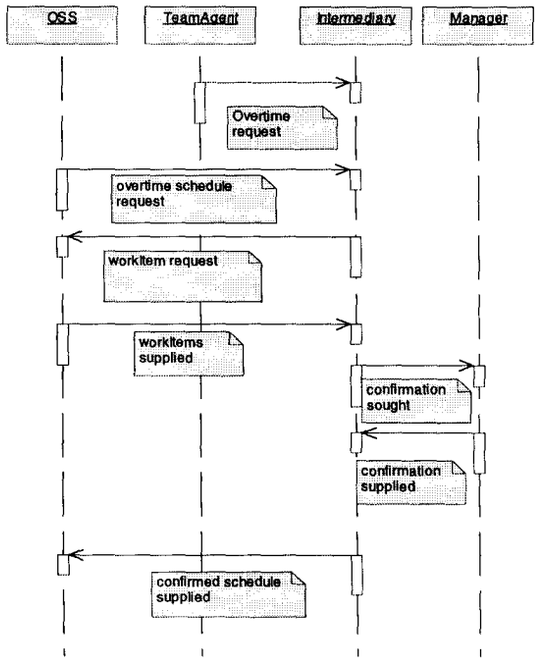

Figure 4. Simplified (contracting not shown) sequence of interactions for overtime scheduling

The OSS's co-ordination engine has now traversed its graph, and has been able to deduce a workable plan for achieving the goal *Schedule*. It will subcontract the goal

to the intermediary and will supply some data so that the intermediary can produce the schedule. The plan is then executed by the schedulers of the Intermediary and the OSS. The task *supplyWorkRequests* is called in the OSS which results in the workItem requests being sent to the intermediary. The task *produceRoster* is then called by the intermediary's scheduler, this task has a sub-goal, which is implemented in its execution script. The sub-goal states that for the task to execute correctly it must *achieve Schedule (ratified true)* . The task *ratifyRoster* in the Manager agent is able to satisfy this goal. The pre-condition of *ratifyRoster* is Schedule(), which can be satisfied by *supplyRosterForRatification*[3]. The co-ordination engine of the Manager agent issues calls and makes a contract with the manager to this effect, and the subgoal is executed before the main task is completed. In Figure 5 we show the plan generated by this process in the context of the agents that execute it.

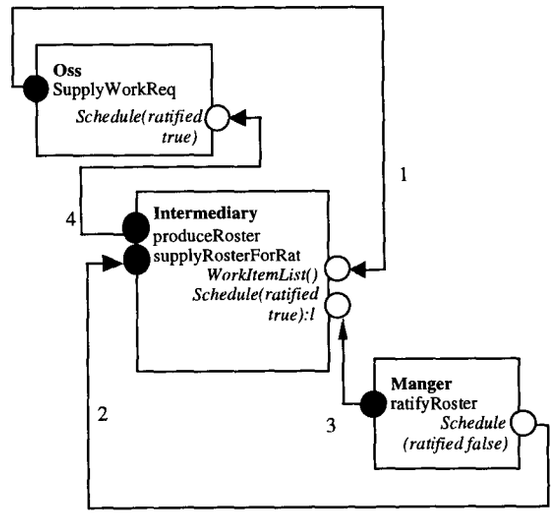

Figure 5. The plan for obtaining an overtime schedule derived by the agents. White circles indicate goals that are being satisfied, black circles indicate task outputs. The arcs are numbered to indicate the order of plan step execution.

Further complications arise if the constraints imposed by the TeamAgents' requests for overtime (not shown in Figure 5) and the Intermediary's requests for workers to complete jobs cannot be resolved, or if the manager fails to approve the schedule. In this case the system must seek a relaxation of some of the constraints. This is done via the Constrained Contract Net protocol described in section 3.2.2.

Similarly, complications arise if one of the agents that has contracted into the schedule is unable to fulfil that commitment. Then, the Extended Request protocol from section 3.2.1 is used to find a suitable replacement.

3.5 Engineer-Agent Interaction

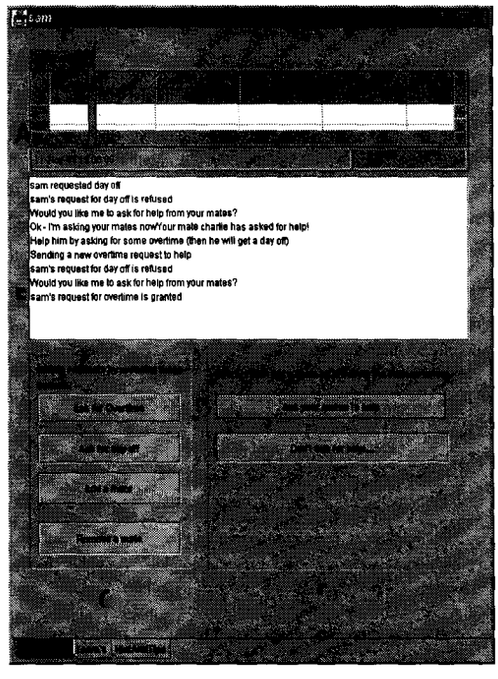

Figure 6. Field engineer to agent interaction screen (letters A to D are labels)

The screen that the field engineers use to interact with their personal agent is shown in Figure 6. It is important that the information needed for the field engineer to interact with their agent is presented graphically. The label A is on the left hand side of the calendar interaction window; the user can use a pointer to select whether they wish to do overtime on a particular date and to view their selections and commitments. Label B is on the left of a "user messages" window, in which the agent displays questions, or statements about the current state of the system. Above label C a set of buttons indicates the actions that the user can take at this time, and above label D two interaction buttons are shown with labels that the agent can change depending on the response that it requires for the interaction at that time.

The interaction of the human operators interfaced with the software agents presented us with an agent level design choice. Either we could capture the interaction

[3] supplyRosterForRatification *(SRFR) has a precondition* Schedule *with a constraint specifing that the* Schedule *fact must be present in the local database . This prevents the intermediary from attempting to satisfy the main goal of the OSS with the SRFR, and also insures that it doesn't attempt to subcontract the provision of the* Schedule *to another agent.*

event and assert a fact in the agent's rule base, and allow the agent's rules to fire as appropriate. Or we could capture the event and instruct the agent to achieve some goal: specifically invoking some reasoning activity by the agent. We chose to use the former approach of asserting a fact like:

```
KeyPressed ("requestOverButton")
```

Which would activate a rule:

```
{condition
  KeyPressed     ("requestOverButton")
FreeDays (Sunday true)}
{action send_message
   (type inform)
   (receiver IntermediaryAgent)
   (content sundayOvertimeRequest)}
```

Our motivation for this design decision was to de-couple the mechanism that invoked an action on the agent from the action being performed, increasing the autonomy of the agent and its flexibility in practice.

3.5.1 Resolving the Schedule

The intermediary is one of the central components of the SAMBA system. Currently we have implemented a static intermediary which uses a rule base fixed at run time to reason over the data that is transmitted to it by the team member's personal agents, as we have described in detail in section 3.4.1.

The intermediary is required to hold a model of the workers that have expressed an interest in overtime and these are combined with the requirements of the enterprise as transmitted from the OSS agent.

Currently we construct the schedule by performing a simple backtracking search in the intermediary. The work items are taken incrementally from the list provided by the OSS, and if there is a worker with the correct qualifications available for overtime in the slot allocated to the work item, they are granted overtime and the work item is regarded as resourced. We track the workers to ensure that they are not allocated work items that have to be performed at the same time.

As we noted in section 3.4.1 if no worker willing to perform overtime can be found, then the scheduling episode has failed and a relaxation of the constraints (normally by negotiating a change in the overtime preference of a worker) must be found.

4. Future Work

One of the extensions to SAMBA that we are considering is the provision of a hints and tips service for sharing knowledge. Engineers are encouraged to add local information to a pre-existing help facility. Due to Quality Management constraints the local tips have to be validated by other engineers before being added to the system. In the framework presented here the intermediary would be in charge of keeping the hints and tips up-to-date and aiding in the interaction between workers in the validation process.

The Hints and Tips service is one possible knowledge management facility we can implement using SAMBA. However, in the overtime allocation and work allocation services that we have been developing the intermediary agent implements the business rules as a static knowledge base. Work in [12] describes our efforts to implement visual interfaces that would allow managers to delete, add and modify rules at run time. Providing the facility for local management practice to be captured in the agent system is another way that we can enhance the corporate learning and memory mechanisms that SAMBA potentially offers.

However, the main drive for this work has been in the production of a set of agent-based services that enable distributed employees to act as a team. So far the focus of the project has been on the provision of services by the agents. But while services have been under development a number of technical issues have become apparent.

One of these issues is the nomadic nature that we would like SAMBA agents to have. By nomadic we mean the agents remain on one device but the device can be moved, switched off etc. This imposes a number of new constraints such as low processor power, vulnerability to interruption and variability in communication cost and bandwidth availability.

These constraints require the classic agent platform design for the use of continuously running agents [3] to be enhanced. For example it now requires a store and forward facility whereby when an agent's communication links are down, the platform stores messages until the agent comes "on-line" and the messages are then forwarded to it.

Nomadic agents are just one of the technical issues that we feel will arise when we make the next step in this project and develop a system suitable for use in a technology field trial, which we plan for later this year. We also plan to conduct further workshops with SAMBA's users, the customer service teams in BT.

5. Conclusions

Traditionally workforce management has been seen as the problem of optimally scheduling the actions of members of the workforce in a dynamic environment. In this paper we have demonstrated that this view of workforce management as a dynamic scheduling problem does not support some elements of the workforce management problem. Specifically SAMBA addresses issues of social interaction, working preference and flexibility that are not handled by previous methods.

We have presented an alternative formulation that views workforce management as a problem of managing the interactions between team members and the interactions of team members and management. The SAMBA architecture of interface agents and intermediaries is one method of implementing this sort of information system.

In order to implement SAMBA we needed to utilise some novel interaction protocols that captured the need of the agents to interact without revealing details of personal preferences, the disclosure of which could damage team dynamics. Our objective was to develop information systems that fit the business processes and practices that we encounter in commercial organisations.

SAMBA represents an application of intelligent agents that enriches the working practices of the workforce while solving an actual business problem. We developed it with reference to its ultimate users (the field workforce at BT) with 10 workshops discussing the shape and use of the technology conducted at sites from Galashields to Bath. Over 88 members of the operational division participated in the workshops with responsibilities ranging from customer service team manager to field engineer.

We believe that SAMBA may prove to be a true "over the shoulder" application that combines, in some way, the properties of collaborative agents, information agents and personal agents. We further believe that the emergence, and ongoing development of agent toolkits like Zeus is the enabler that will release the potential of intelligent agent research into the real world.

6. References

[1] Dourish, P., J. Holmes, A. MacLean, P. Marqvardsen, and A. Zbyslaw (1996, November). Freeflow: *Mediating between representation and action in workflow systems*. In ACM Computer Supported Cooperative Work. ACM.

[2] DTI 1998 *DTI Direct Access to Legislation: Chapter 2 Working Time Limits* http://www.dti.gov.uk/IR/work_time_regs/wtr2.htm

[3] FIPA 1997. FIPA '97 Specification Part 1, *Agent Management* . The Foundation for Physical Agents, Geneva, Switzerland. http://www.fipa.org/spec/FIPA97.html

[4] FIPA 1997. FIPA '97 Specification Part 5, *Personal Assistant* . The Foundation for Physical Agents, Geneva, Switzerland. http://www.fipa.org/spec/FIPA97.html

[5] Forgy, C. L. (1982) *"RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem."* Artificial Intelligence 19, pages 17--37, 1982

[6] Giarratano, J. & Riley, G., (1994) "Expert Systems: Principles and Programming", PWS Publ 1994, ISBN 0-534-93744-6

[7] Ginsberg, M.L., Parks, A.J. & Roy, A. "Supermodels and Robustness", In Proceedings *The Fifteenth National Conference on Artificial Intelligence AAAI'98, AAAI Press.*

[8] Harker, P. T., and Ungar, L. H. 1996. *A market-based approach to workflow automation.* In Proceedings of NSF Workshop on Workflow and Process Automation in Information Systems: State of the Art and Future Directions. Athens GA.

[9] Intelligent Personal Assistant, http://innovate.bt.com/showcase/ipa/index.htm

[10] Jennings, N. R., Faratin, P., Johnson, M. J., O'Brien, P., and Wiegand, M. E. 1996. *Using intelligent agents to manage business processes.* In Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM96), 345-360. London, UK: The Practical Application Company Ltd.

[11] Kohn A., 1993, *Punished by Rewards: The trouble with gold stars, incentive plans, A's, praise and other bribes*, NY: Houghton Mifflin.

[12] Lesaint, D., Azarmi, N., Laithwaite, B., Walker, P., 1998 "Engineering Dynamic Scheduler for Work Manager", *BT Technology Journal,* Vol 16, No. 3, 1998.

[13] Mehandjiev, N. and Odgers, B, 1999. *SAMBA: Agent-supported visual interactive control for distributed team building and empowerment* BT Technical Journal, Vol. 17 No. 4, October 1999

[14] Millard, N, 2000. "The Future of Customer Contact", BT Technical Journal, Vol 18, No.1, January 2000

[15] Nwana, H.S., Ndumu, D.T., Lee, L.C. & Collis, J.C. (1999) "Zeus: A toolkit for Building Distributed Multi-Agent Systems", *In : Applied Artificial Intelligence Journal,* Vol 13 (1), p187-203

[16] PLANET; http://planet.dfki/de

[17] Plexus Floware; http://www.plx.com/html/floware_scaleable_workflow.html

[18] Oracle Workflow; http://www.oracle.fr/services/support/pdf/workflow_setup.pdf

[19] Sarker B.M., Butler B. and Steinfield C. (1995) *Intermediaries and Cybermediaries: A Continuing Role for Mediating Players in the Electronic Marketplace.* Journal of Computer-Mediated

Communication Vol. 1 No. 3. http://jcmc.huji.ac.il/vol1/issue3/sarkar.html

[20] Smith, R. G., 1980. *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem-Solver*, IEEE Transactions on Computers, vol 12, 1980

[21] Smith, R. G. and Davis, R., 1981. *Frameworks for Cooperation in Distributed Problem Solving*, IEEE Transactions on System Man and Cybernetics, 11(1), 1981.

[22] Swenson, K et al. *A business process environment supporting collaborative planning.* Journal of Collaborative Computing, 1(1): 15–34, March 1994.

[23] Tapped In, http://www.tappedin.sri.com/

[24] Tambe, M. 1997. "Toward Flexible Teamwork" *Journal of Artificial Intelligence Research*, **7** , 83-124

[25] The Personal Information Assistant, http://walrus.stanford.edu/diglib/pub/proposal/partII/node30.html

[26] The Virtual Secretary, http://www.vise.cs.uit.no/vise/

[27] WinWin, http://sunset.usc.edu/WinWin/winwin.html

[28] The Zeus Toolkit, http://www.labs.bt.com/projects/agents/zeus/index.htm

Using Planning Formalisms to Reason about Agent Capabilities

Gerhard Wickler
ITC–IRST, via Sommarive 18, 38050 Povo (TN), Italy
gw@itc.it

Abstract

The aim of this paper is to address the problem of capability brokering. For this purpose we will define a new capability description language, CDL, that has two desirable properties: it is expressive, i.e. it has the potential to represent certain circumstances that cannot be represented in a less expressive formalism, and it is highly flexible, i.e. it allows the knowledge engineer to choose a compromise regarding certain trade-offs at the time of knowledge representation.

1 Introduction

One approach to achieving artificial intelligence is the *rational agent approach* (Russell and Norvig, 1995, p. 7). In this approach, the field of AI is viewed as the study and construction of rational agents. Unfortunately there is no agreed definition of what constitutes an agent as yet. One characterisation of what an agent is can be found in Wooldridge and Jennings (1995). They identify four *necessary properties* of an agent which most definitions of agency seem to agree on: autonomy, social ability, reactivity, and pro-activeness.

Social ability, the property we will be most concerned with, means that an agent interacts with other agents (possibly humans) via some kind of agent communication language. Pro-activeness means that an agent should be able to exhibit goal-directed behaviour by taking the initiative. Taken together, pro-activeness and social ability imply that an agent should communicate not with just any other agent, but specifically with those agents that can help it achieve its goals. For an agent to achieve this behaviour, it will be necessary to first *find* these other agents. Finding such agents is part of the problem we are addressing in this paper.

We will assume here that an agent exists in a dynamic environment with other agents. As the environment changes new agents might come into existence or existing agents might disappear. Agent autonomy means that an agent has to operate without the direct intervention of humans, i.e. that it has to find out by itself about other agents that exist, specifically, agents that can help it achieve its goals.

Genesereth and Ketchpel (1994) distinguish two basic approaches to this connection problem: direct communication, in which agents handle their own coordination and *assisted coordination*, in which agents rely on special system programs to achieve coordination. Only the latter approach promises the adaptability required to cope with the dynamic environment we envisage.

Decker et al. (1997) have recently described a solution space to the connection problem based on assisted coordination. The special system programs for coordination are called middle-agents in their analysis. They identify nine different types of middle-agents depending on which agents initially know about capabilities and preferences of agents. In a solution to the connection problem in which capabilities are initially known to the provider and the middle-agent only, and in which preferences are initially known to the requester and the middle-agent only, the middle-agent is what they call a *broker*.

1.1 Defining the Problem

Capability brokering involves communication between different agents. For a specific instance of this problem we shall distinguish three different *types of agents* according to the roles they play for this problem instance:

1. The **Problem-Solving Agents** (PSAs) provide the capabilities that may be called upon by other agents in order to solve their problems.
2. The **Problem-Holding Agents** (PHAs) have a problem that they wish to have solved by utilising the capabilities of the PSAs.
3. The **Broker** matches the problems of the PHA to the advertised capabilities of the PSAs such that the problems can be solved.

The basic protocol for the *exchange of messages* between the different agents that has to take place for capability brokering is illustrated in figure 1. Since capabilities are meant to be known by the PSA and the broker initially, it is necessary that the PSAs advertise their capabilities to the broker. At the time of brokering, problems are meant to be known by the PHA and the broker and thus, the PHA has to inform the broker about its problem as it arises. If a capability has been advertised to the broker that can be used to address the given problem then the

broker should retrieve it and inform the PHA about this capability and the agent that has it. Finally, the PHA can use the information from the broker to ask the PSA with the necessary capability to tackle its problem. Notice that not all of the messages outlined above necessarily have to occur in this order.

Now, *the problem of capability brokering is to achieve the behaviour of the broker outlined in the protocol in figure 1.*

2 Representations for Capabilities

To achieve the behaviour outlined in the protocol in figure 1 it is necessary to *explicitely represent and communicate the capabilities* of the different PSAs. For this purpose we have defined a new capability description language, CDL, that has a number of desirable characteristics outlined below (cf. Wickler (1999)).

2.1 Desiderata for CDL

The two most important properties we want our capability description language to have are *expressiveness* and *flexibility*. By expressiveness we mean the potential to represent certain circumstances that cannot be represented in a less expressive formalism. By flexibility we mean the possibility for the knowledge engineer to choose a compromise regarding certain trade-offs at the time of knowledge representation rather than having to adopt a fixed compromise prescribed and designed into the chosen formalism.

Another characteristic we would like CDL to have is that it is similar to languages which have been *used for capability brokering* successfully, as this would indicate that CDL, too, can be used for brokering. Likewise, since capabilities can be seen as actions one can perform (cf. section 2.2), we would also expect CDL to be similar to representations that have been *used to represent and reason about actions*. In both cases the similarity should only cover properties that contributed to the success of these languages.

As we expect the broker to perform its services autonomously, it is important that the capability representations are in some *formal* language; CDL must have this attribute. Finally, every representation must *have a semantics* to qualify as a representation in the first place (cf. Hayes (1974)), so we shall pay attention to this property as well.

2.2 Capabilities and Actions

Most action representations in AI are representations that describe how the state of the world changes when an action is performed and what needs to be true before that action can be executed. Capability descriptions need to *convey very much the same knowledge*, i.e. what changes a capability can bring about and what needs to be true for that capability to be applicable. There are two major differences between actions and capabilities though:

- **Level of description**: An action is less *abstract* than a capability in the sense that we would expect all its parameters to be instantiated for its execution. However, AI planning systems use operator schemata rather than instantiated actions as input, i.e. they effectively use capability descriptions.
- **Modality**: A capability is an action that *can* be performed (at least in theory), i.e. it has a different modality. But this is implicitly what an AI planner usually assumes when it generates a plan; that the operator schemata it instantiates and inserts into the plan represent capabilities of some agent (cf. (McCarthy and Hayes, 1969, pages 470-477)).

2.3 The Knowledge in Capability Representations

We are now in a position to describe the *knowledge* contained in a CDL capability representation. The core CDL representation for achievable objectives is based on a classical, non-hierarchical operator description and consists of the following parts:

- **Inputs**: This part of the capability representation specifies the objects an agent possessing this capability receives as inputs to this capability. How these inputs will be used is unspecified here.
- **Outputs**: This part of the representation specifies the objects that will be the outputs this capability generates.
- **Input Constraints**: This part defines the constraints that are expected to hold in the situation before this capability can be performed, i.e. the constraints for the capability to be applicable.
- **Output Constraints**: This part defines the constraints that are expected to hold in the situation after this capability has been performed.
- **Input-Output Constraints**: This final part defines the constraints across input and output situations that must hold.

One difference between capability descriptions in CDL and STRIPS-like operator descriptions is that CDL distinguishes *two types of parameters*: inputs and outputs. Parameters are essentially the objects involved in the performance of a capability and must all be instantiated for the execution of a specific action instance. CDL distinguishes input objects, i.e. objects that exist in the situation before the capability is applied, and output objects, i.e. objects that exist only in the situation that results from the application of this capability in the input situation.

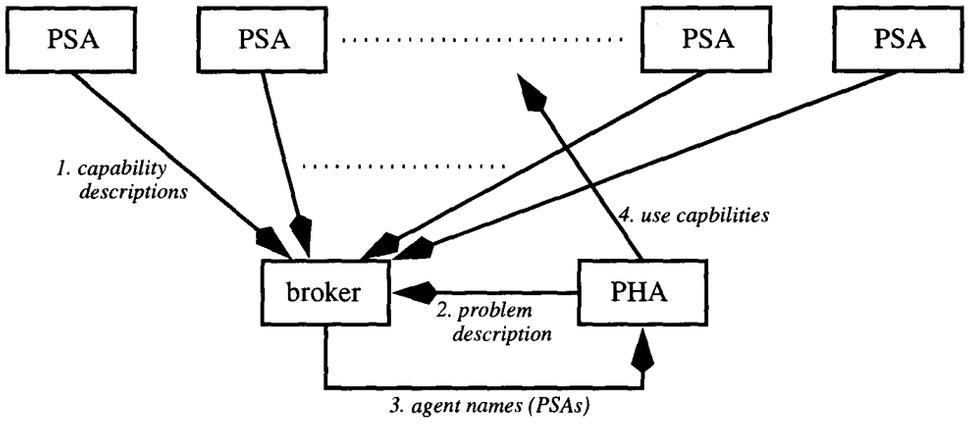

Figure 1: Basic message flow in brokering

Input constraints in CDL *directly correspond to the precondition formula* in classical non-hierarchical action representations. In accordance with modern planning formalisms (cf. Tate et al. (1994)) we prefer to view the precondition formula as a constraint on the situation in which the capability can be applied. Notice that input constraints may only mention objects from the inputs as these are the only objects that exist in this situation. *Output constraints in* CDL *correspond to a combined add and delete list*, i.e. to the effects of an action, and represent constraints on the situation that results from the application of this capability. Finally, output constraints may mention objects that exist in the output situation, i.e. objects from inputs or outputs.

The final set of constraints mentioned above are the *input-output constraints which correspond roughly to secondary preconditions and effects* in ADL (cf. Pednault (1989)). These constraints do not refer to only one situation like the input and output constraints but are constraints across both of these situations. This type of constraint allows one to refer to objects that have different properties in different situations and to expresses a condition on the properties in these different situations.

2.4 Decoupling the Representation

At this point we know what the knowledge is we need to represent in CDL. The next obvious question is *what language* to use to express the different constraints in. Our aim is to inherit the expressiveness and well-defined semantics of logics in CDL, but we also want to retain the flexibility of KQML (cf. Finin et al. (1997)).

Many knowledge representation languages are *state representation languages* at heart, i.e. they implicitly assume the world to be in exactly one state or situation at any given time. That is, unless otherwise stated, a set of sentences in such a language is assumed to refer to the same implicit situation. Knowledge representation languages usually also assume that there exist a number of objects in this implicit situation and that certain relations hold between these objects in this situation.

In most conventional action representation languages such as STRIPS, the state representation language is an integral part of the overall representation language. We shall call such languages *integral action representations*. For example, STRIPS (as described in (Nilsson, 1980, ch.7)) only allowed conjunctions of positive literals in the input and output constraints of its representation. However, it is relatively trivial to extend the state language to allow for more complex formalisms, e.g. horn clauses, full first-order logic, modal logics, etc. With an integral action representation we have to commit to one of these languages and every new state representation language defines a new action representation. It is this inflexibility that we seek to avoid in CDL as it is not clear which would be the right state language for describing arbitrary agent capabilities.

To allow the arbitrary combination of action and state representation we will define the action representation language independent from the state representation language. We shall call this a *decoupled action representation*, i.e. a full action representation consists of a decoupled action representation combined with a state representation language. Syntactically, decoupling will be achieved using an approach similar to the way KQML allows content expressions to be in some independent content language, i.e. by having a field that names the content language and one that holds exactly one expression in this language as a sub-expression of the wrapper. CDL will also allow the nomination of a state language in which the different types of constraints are to be expressed, except that there will be several sub-expressions in the named content language. By decoupling the action from the state representation, CDL will achieve the same, high flexibility that KQML provides.

Finally, the following capability advertisement for a hospital agent (cf. Wickler (1999)) illustrates the flexibility and expressiveness that can be used in CDL:

```
(capability
  :state-language fopl
```

```
:input ((InjuredPerson ?person))
:input-constraints (
  (elt ?person Person)
  (Is ?person Injured)
  (or (Has Location ?person Abyss)
    (Has Location ?person Barnacle)
    (Has Location ?person Exodus))
  (implies (or(on Road Ice)(on Road Snow))
    (have Ambulance SnowChains)))
:output-constraints (
  (not(Is ?person Injured))))
```

In this capability description the sole input parameter is the injured person. Constraints on the situation before the capability can be applied include a typing constraint, the fact that the given person must actually be injured, that s/he must be in one of the three given places, and finally that if there is ice or snow on the road the ambulance will need to have snow chains. Finally, the output constraint states that the given person will no longer be injured after this capability has been applied. Flexibility through decoupledness is illustrated here by the named state language: `fopl`. Expressiveness is illustrated in the input constraints by formulae that go beyond simple conjunctions of literals.

2.5 Performable Actions

Every capability can be described as *achieving an objective or as performing an action*. The former description can be regarded as an objective-centred description and the latter is an action-centred description. Natural language allows us to describe every capability in both ways, although some descriptions might sound awkward to us. Performing an action can be described as achieving a state in which the action has been performed. Achieving an objective can be described as performing an action of type achieving for the given objective. Thus, both descriptions are effectively equivalent.

To think of capabilities in terms of performable actions as opposed to achievable objectives has one major advantage: one can *define a new capability* in terms of other, more primitive capabilities. For example, suppose the broker knew the description of a general sorting action. If a new agent now wants to advertise the capability that it can sort lists of integers, and this new agent is aware of the broker already knowing about the description of a sorting action, then the new agent could advertise its integer sorting capability based on the description of the sorting action already known to the broker. All the new agent needs to do in this case is refer to the broker's existing description of a sorting action and modify it by stating the additional constraint that the elements of the given list must all be integers.

The knowledge the broker would need to achieve this kind of behaviour is effectively an *ontology of actions*. It is conceivable that a broker knowing about a number of primitive actions in an ontology would be much easier to communicate with, as it would not be necessary to represent every new capability completely from scratch.

Thus, we shall briefly describe an extension of CDL to allow for the representation of performable actions. If the broker has an ontology of actions and another agent wants to define a new capability in terms of an action in this ontology, it needs to be able to refer this action in the ontology in some way. For this purpose we need to add the following to CDL:

- **a capability identifier**: this field allows the specification of a unique action name for a capability; and
- **a capability inheritance link**: this field allows the naming of an action from which this capability will inherit the description.

When a new capability description inherits from an action description in the broker's action ontology, the description of the new capability is effectively a description of how to modify the inherited action description inherited from to obtain the new capability description. We shall call a CDL expression that describes a capability by inheriting from some action a *modification description*. Three principal types of modification possible are:

- **New parameters**: The modification description can specify additional parameters for input and output in the inheriting capability description.
- **Instantiated parameters**: The modification description can give values for parameters defined in the description inherited from, i.e. these parameters are instantiated in the inheriting description.
- **New constraints**: The modification description can specify additional input, output, or input-output constraints involving all the new parameters as well as inherited parameters.

This extension allows the representation of capabilities as performable actions. The syntax of the core of CDL is given in figure 2. For a detailed description of the complete syntax of CDL including this extension we again have to refer to Wickler (1999), which also contains several examples that illustrate the various aspects of CDL.

To illustrate modification descriptions and the inheritance mechanism outlined above we shall now look at a *simple example*. The first thing we need is an ontology of actions known to the broker. For simplicity, we shall describe only one action in this ontology: a moving action. This action will be described as follows:

```
(capability
  :action move
  :state-language fopl
  :input ((Thing ?thing)(From ?p1)(To ?p2))
  :input-constraints (
    (Has Location ?thing ?p1))
```

```
<cdl-descr> ::= ( <ctype>
                    :state-language <name>
                    :action <name>
                    :isa <name>
                    :properties ( <name>+ )
                    :input ( <param-spec>+ )
                    :output ( <param-spec>+ )
                    :input-constraints ( <constraint>+ )
                    :output-constraints ( <constraint>+ )
                    :io-constraints ( <constraint>+ )

<ctype> ::= capability | task

<param-spec> ::= ( <name> <term> )
<term>       ::= <constant> | <variable> |
                 ( <constant> <term>+ ) |
<variable>   ::= ?<name>
<constant>   ::= <name>

<constraint> ::= << expression in state-language >>
```

Figure 2: Syntax of core CDL in BNF

```
:output-constraints (
  (not (Has Location ?thing ?p1))
  (Has Location ?thing ?p2)))
```

The three parameters are the object that is to be moved (`?thing`), the place from where it is to be moved (`?p1`), and the place to which it is to be moved (`?p1`). The sole constraint on the input situation is that the thing to be moved is at the place from where it is to be moved: `(Has Location ?thing ?p1)`. The output constraints state that `?thing` will not be at the initial location anymore after the action has been performed: `(not (Has Location ?thing ?p1))`; and that it will be at the location it was to be moved to: `(Has Location ?thing ?p2)`. The name of this action is given as `move`. We shall now assume that this action description is known to the broker before it receives any capability advertisements.

Now, suppose a hospital also wants to *advertise* the capability that it can move patients to the hospital. Of course, this could be done by simply defining a new capability, but it can also be described as a modification of the moving action already known to the broker. Thus, the hospital-agent could send a capability advertisement message to the broker with the following content:

```
(capability
  :isa move
  :state-language fopl
  :input ((To Hospital2)(Ambulance ?a))
  :input-constraints (
    (elt ?thing Person)
    (Is ?thing Injured)))
```

This CDL description first states that it inherits from the move action in the broker's action ontology. This action is modified by instantiating the input parameter `(To ?p2)` to `Hospital2`, i.e. the capability can only move objects to this hospital. The description also adds one more input parameter, the `Ambulance` that is to be used in the application of this capability. Thus, the three input parameters of the new capability described here are the object to be moved (i.e. the patient) and the place it is to be moved from, both inherited from the move action, and the ambulance with which the patient is to be moved. The capability description also extends the input constraints, specifying that the object to be moved must be a person: `(elt ?thing Person)`; and that this person must be injured: `(Is ?thing Injured)`. It also inherits the input constraint, `(Has Location ?thing ?p1)`, and the first output constraint, `(not (Has Location ?thing ?p1))`, from the move action. The second output constraint, however, is modified to `(Has Location ?thing Hospital2)` because the input parameter `To`, which is represented by the variable `?p2` in the description of move, has been instantiated to `Hospital2` in the input of the modification description.

3 Related Work

KQML (cf. Finin et al. (1997)), the de-facto standard for agent communication languages, effectively defines an interface to a broker by providing a set of performatives that can be used for capability brokering. In fact, the broker we have implemented uses these KQML performatives with CDL as the inner language. KQML does not define an inner language to be used for brokering and thus can-

not not be compared with CDL. Similar comments apply to the FIPA[1] standard, although this is still under development.

The ABSI facilitator (cf. Singh (1993)) was one of the earliest brokers and is based on some of the KQML brokering performatives. The only content language supported is KIF (cf. Genesereth et al. (1992)). While this appears to be a powerful formalism, the limitation comes with the matching performed by the broker which is essentially a simple unification procedure. In addition, a few simple constraints on the variables bound during the unification can be specified. This behavior can easily be emulated in CDL by our broker and thus, CDL is effectively a more expressive language. Furthermore, the ABSI facilitator provides no flexibility at all.

Similar limitations apply to the SHADE/COINS matchmakers (cf. Kuokka and Harada (1995)) and the brokers in the InfoSleuth architecture (cf. Nodine et al. (1998)). Again, capabilities are to be described in KQML. The primary envisaged content language is KIF, although one alternative exists for each broker. Thus, the formalism is expressive, but the matching is again a simple unification procedure limiting the usable expressiveness. By providing two content languages these brokers provide at least a very small amount of flexibility though.

A more interesting language is LARKS (cf. Sycara et al. (1999)). Like CDL it consists of parameter descriptions and constraints on input and output situation. While it does not provide cross-situational constraints, it does allow for concept definitions within the capability description. In CDL all used concepts must be defined in the ontology which is referenced in the wrapping KQML layer. LARKS also provides some interesting types of matchmaking between capabilities and tasks. What it does not provide, however, is flexibility: constraints must be Horn clauses.

In fact the only other languages we are aware of that do provide flexibility are complex planning formalisms such as O-Plan TF (cf. Tate et al. (1998)) or SPAR (cf. Tate (1998)), and KQML. However, the latter cannot be considered a full capability description language as its content language is undefined. The problem with O-Plan TF and SPAR is that they do not provide reasoning mechanisms that can adequately treat constraints in unknown languages. CDL deals with this problem by using reflective reasoning and loading unknown languages from the Internet, both only made possible by its implementation in Java, which supports these features.

4 Conclusions

In this paper we presented a new capability description language, CDL. Such a language is essential for the explicit representation of capabilities as required for the brokering performed by middle-agents. This is the approach generally considered most promising for agent coordination in an open environment.

By adopting the structure of action representations used in AI planning CDL inherits the experience gained with such formalisms in three decades of planning research. Thus, it can be considered a well-founded language. Building on such formalisms, CDL has two further important properties: it is highly flexible which has been achieved by implementing it in a KQML-like fashion as a decoupled language, and it is expressive which has been achieved by allowing for first-order logic as one possible state description language within CDL.

The core of CDL as described in this paper as well as several important extensions to this core have been implemented in the programming language Java. As content languages, first-order logic and a restricted version that just permits conjunctions of literals have been implemented to illustrate flexibility. Furthermore, there is a broker based on CDL that uses a reflective matching algorithm to pair tasks and capabilities. Finally, a number of agents and scenarios including the very simple one described in this paper have been implemented to evaluate the the whole framework.

The code and the thesis that describes this project (Wickler (1999)) in detail will be distributed on the O-Plan 3.3 CD in the near future.[2]

References

James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning*. Morgan Kaufmann, San Mateo, CA, 1990.

Ronald J. Brachman and Hector J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, Los Altos, CA, 1985.

Keith Decker, Katia Sycara, and Mike Williamson. Middle-agents for the internet. In *Proc. 15th IJCAI*, pages 578–583, Nagoya, Japan, August 1997. Morgan Kaufmann.

Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In Jeffrey M. Bredshaw, editor, *Software Agents*, chapter 14, pages 291–316. AAAI Press/MIT Press, Menlo Park, CA/Cambridge, MA, 1997.

Michael R. Genesereth, Richard E. Fikes, Daniel Bobrow, Ronald Brachman, Thomas Gruber, Patrick Hayes, Reed Letsinger, Vladimir Lifschitz, Robert MacGregor, John McCarthy, Peter Norvig, Ramesh Patil, and Len Schubert. Knowledge interchange format version 3.0 reference manual. Report Logic-92-1, Stanford University, Stanford, CA, June 1992.

[1] see http://www.fipa.org/

[2] see http://www.aiai.ed.ac.uk/~oplan/cdl/

Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 147, July 1994.

Patrick J. Hayes. Some problems and non-problems in representation theory. In *Proc. AISB Summer Conference*, pages 63–79, University of Sussex, 1974. Also in: (Brachman and Levesque, 1985, pages 4–22).

Daniel Kuokka and Larry Harada. Matchmaking for information agents. In *Proc. 14th IJCAI*, pages 672–678, Montréal, Canada, August 1995. Morgan Kaufmann.

John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In Bernhard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969. Also in: (Allen et al., 1990, pages 393–435).

Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA, 1980.

Marian Nodine, Brad Perry, and Amy Unruh. Experience with the InfoSleuth agent architecture. In Brian Logan and Jeremy Baxter, editors, *Proc. AAAI Workshop on Software Tools for Developing Agents*, Madison, WI, January 1998. AAAI Press.

Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proc. 1st KR*, pages 324–332, Toronto, Canada, 1989. Morgan Kaufmann.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.

Narinder Singh. A Common Lisp API and facilitator for ABSI. Report Logic-93-4, Stanford University, Stanford, CA, January 1993.

Katia Sycara, Jianguo Lu, Matthias Klusch, and Seth Widoff. Matchmaking among heterogenious agents on the internet. In *Proc. AAAI Spring Symposium on Intelligent Agents in Cyberspace*, Standford, CA, 1999.

Austin Tate. Roots of SPAR—shared planning and activity representation. *The Knowledge Engineering Review*, 13(1):121–128, March 1998.

Austin Tate, Brian Drabble, and Richard Kirby. O-Plan2: An open architecture for command, planning and control. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*, chapter 7, pages 213–239. Morgan Kaufmann, San Francisco, 1994.

Austin Tate, Stephen T. Polyak, and Peter Jarvis. TF method: An initial framework for modelling and analysing planning domains. In *Proc. Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice*, Pittsburgh, PA, June 1998. Carnegie-Mellon University, AAAI Press.

Gerhard Wickler. *Using Expressive and Flexible Action Representations to Reason about Capabilities for Intelligent Agent Cooperation*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, April 1999.

Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theories and practice. *The Knowledge Engineering Review*, 10(2):115–152, June 1995.